Learning from the Environment Based on Percepts and Actions

Wei-Min Shen

June 26, 1989

CMU-CS-89-184

Submitted to Carnegie Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, OHIO 45433-6543, under Contract F33615-87-C-1499. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the United States Government.

To the people, especially those in China.

Abstract

This thesis is a study of "learning from the environment." As machine learning moves from toy environments towards the real world, the problem of learning autonomously from environments whose structure is not completely defined *a priori* becomes ever more critical. Three of the major challenges are: (1) how to integrate various high level AI techniques such as exploration, problem solving, learning, and experimentation with low level perceptions and actions so that learning can be accomplished through interactions with the environment; (2) how to acquire new knowledge of the environment and to learn from mistakes autonomously, permitting incorrect information to be identified and corrected; (3) how to create features in such a way that knowledge acquisition is not limited by the initial concept description language provided by the designers of a system.

The thesis defines learning from the environment as the process of inferring the laws of the environment that enable the learner to solve problems. The inputs to the learning are *goals* to be achieved, *percepts* and *actions* sensed and effected by the learner, and *constructors* for organizing the percepts and the actions. The output is a set of *prediction rules* that correlate actions and percepts for achieving the given goals. The thesis develops a framework called LIVE that unifies problem solving with rule induction. It creates rules by noticing the changes in the environment when actions are taken during exploration. It uses a new learning method called *complementary discrimination* to learn both disjunctive and conjunctive rules incrementally and less biasedly from feedback and experiments. Furthermore, it constructs "new terms" to describe newly discovered hidden features in the environment that are beyond the scope of the initial perception description language.

Theoretically, if the environment is a n state finite deterministic automaton and each of the learner's m actions has a known inverse, then the learning method can learn with accuracy $1-\epsilon$ and confidence $1-\xi$ the correct model of the environment within $O(n^2\, m\, \frac{1}{\epsilon} \log \frac{1}{\xi})$ steps. Empirically, in comparison with existing learning systems, this approach has shown the advantages of creating problem spaces by interacting with environments, eliminating the bias posed by the initial concept hierarchy, and constructing new vocabulary through actions. The generality of this approach is illustrated by experiments conducted in various domains, including child learning, puzzle solving, robot simulation and scientific discovery.

Acknowledgements

A Chinese proverb says: "Among every two people you walk with in the streets, one of them must be your teacher." In these six productive years in CMU, I have had many teachers both inside and outside computer science.

I owe a great deal of gratitude to my advisor Herbert A. Simon, who opened for me the doors to western science and guided me in the maze of scientific research. To me, he is both my academic advisor and a noble man who has dedicated himself to the whole of mankind. This thesis is only a by-product of what I have learned from him.

Professor Jaime G. Carbonell, my co-advisor for this thesis, has shown great support for me. In our many pleasant meetings, he always understood the ideas that I was trying to tell him better than I did myself. Professors Tom Mitchell and Chuck Thorp, as members of my thesis committee, have given me many valuable comments to make this thesis better.

Throughout these years, I have received enormous help from my friends in CMU. I have had valuable advice from Allan Newell and Balas Natarajan. I have also overcome my Confucianess and learned how to discuss ideas with people even if I don't know what I am saying. Among them are Jill Fain, Murray Campbell, Angela Hickman, Jeffrey Schlimmer, David Long, Peter Jansen, Ken McMillan, Jerry Burch, Yolanda Gil, Klaus Gross, Han Tallis, Peter Shell, Allan Fisher, Peter Highnam, Peng Si Ow and many others.

Finally, I think no single man can stand for long. My lovely wife Sai Ying has made me complete and my life more meaningful. My parents and sister in China have supported me conditionlessly and sacrificed a great deal to make me who I am now. Although my son Theodore Tong Shen is about the same age as this thesis, he has contributed so much that it is very hard to clarify which ideas in the thesis were mine and which were his.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This thesis is a study of machine learning from the environment. As we know, machines like computers perform certain tasks, such as calculating and memorizing, better than humans, but they do poorly on others, such as learning and discovering. Aside from the philosophic question whether computers can eventually exceed humans, two of the reasons are manifest: computers have not had the ability to actively interact with the external world, and their internal knowledge representation languages are planted by brain surgery rather than based on their own ability to see and act. In the light of this philosophy, this thesis has attempted to find the principles of active learning based on the learner's own physical abilities and implement them on computers. Such attempts have led to some interesting new results, including an architecture for autonomous learning, a new learning algorithm that learns from mistakes, and a linkage from daily-life learning to scientific discovery.

Learning from the environment is a very common phenomenon in real life. Imagine yourself just beginning to learn how to swim. You may ask your friends for advice or even read books about it, but when you are actually in the water, the first thing you will probably do is to move your arms and legs to see what really happens. After noticing that rowing your arms backward causes your body to move forward, you may deliberately move your arms in order to move your body. Meanwhile, you may choke once or twice when you accidentally breathe water. You thus learn that you should never breathe under water. After a while, you gradually become familiar with the environment of the water and may be able to swim one or two meters. This process, the explorations, the mistakes, and the

interactions, is the process of learning from the environment.

Learning from the environment is a developmental process of a learner, whether it is a human or a machine, that combines exploration, learning and problem solving. The goal of the learner is to use its percepts and actions to infer the laws of the environment, and then utilize the laws to solve problems. Since the consequences of actions are unknown in a new environment, the learner must explore the environment to identify the correlations between its percepts and its actions. These correlations then become the basic laws, and are used immediately to solve problems. When a mistake is detected, the learner must refine the laws that cause the error. When the current laws are not sufficient to make any progress in problem solving, the learner must explore again to gather more information. These activities are repeated by the learner until the given problem is solved.

Learning from the environment is also an interesting and important task in Artificial Intelligence. Three reasons are worth mentioning. First, learning from the environment provides an opportunity to study human problem solving more realistically because the task requires a system to create problem spaces during problem solving instead of being fed at the outset. Psychological evidence [17, 20] has shown that when given an unfamiliar problem, human problem solvers do not start with the correct problem space. Instead, they create a suitable search space during the process of problem solving. This explains why two isomorphic problems that have the same problem space for a computer can have different difficulties for human problem solvers. Previous studies have mainly concerned how problem spaces are created when problems are presented as natural language instructions [17, 35]; this thesis will concentrate on creating problem spaces through interactions with an environment.

Second, learning from the environment can have direct applications to building learning robots. Such robots can be very useful to learn from mistakes and operate in those environments in which circumstances cannot be foreseen and humans have little or no controls. To realize this dream, high-level problem solving and reasoning techniques must be combined with physical level perception, planning, and control. Since learning from the environment requires both problem solving and interaction with the environment, we hope the solutions developed here can contribute some useful ideas towards that direction.

Third, we believe that building autonomous learning systems that can perceive and act in their environments is one of the best ways to explore the mystery of intelligence.

Current research and technology have pushed us to the frontiers for exploring such integrated intelligent systems. Although previous researchers have shown us how to decompose the problem and studied the parts separately, integrating all the components together is a very exciting challenge.

Research in Artificial Intelligence has studied learning from the environment from different aspects. They are commonly referred to as: *prediction, explanation, problem solving* (or *planning*), *learning* and *discovery*. The task of prediction is to predict the next state of the environment given the current state and an action [44, 45]. For example, given a closed circuit consisting of a battery, a light, a switch, and some metal wires, the task is to predict what will happen if the switch is turned on (i.e. the light will be on). The task of explanation, which is roughly the reverse of prediction, is to find the reasons why an action causes certain results [7, 14, 24]. For example, after observing that the switch turns the light on, the task is to explain whether it is because the battery and the light are connected by the wire, or because today is Friday. The task of problem solving, which can be viewed as a sequence of predictions, is to plan a sequence of actions which will achieve a given goal state [13, **?**, 21, 30, 32]. For example, given the parts of the circuit, the question is how to put them together so the light will be on. Finally, the task of learning and discovery, as an extension of explanation, is to identify or discover the correct conditions (which may or may not be directly observable) so the prediction can be more accurate and problem solving can be more efficient [11, 25, 27]. For example, when plastic wires are also available the conductivity of materials should be discovered by noticing that with plastic wires the light does not shine but with the metal ones it does. Many problem solving systems have incorporated such learning ability [**?**, 21, 30, 32].

From a theoretical point of view, learning from the environment bears a close relation to the problem of system identification or *inductive inference* [3, 4, 16, **?**]. However, in contrast to the pure theoretical work, learning from the environment does not look for a perfect model of the environment, but rather for laws that are good enough for solving the given problem. This characteristic provides a more practical criterion for machine learning because it is closer to human learning.

Although great progress has been made in research in each individual area, studying learning from the environment as an integrated problem is still challenging and necessary. This is not only because an autonomous AI system is more than simply putting all the parts

together, but also because the close relationship between the four subtasks suggests that they can benefit each other only when studied together. For example, prediction can be used as an evaluation criterion for explanation; explanation provides a means for improving prediction; problem solving detects when and where further prediction and explanation are necessary; and learning and discovery provide more building blocks for prediction, explanation and problem solving.

The thesis defines learning from the environment as the process of inferring the laws of the environment that enable the learner to solve problems. It develops a framework called LIVE that unifies problem solving with rule induction. LIVE creates rules by noticing the changes in the environment when actions are taken during exploration. It uses a new learning method called *complementary discrimination* to learn both disjunctive and conjunctive rules incrementally and less biasedly from feedback and experiments. Furthermore, it constructs "new terms" to describe newly discovered hidden features in the environment that are beyond the scope of the initial perception description language.

Theoretically, if the environment is a n state finite deterministic automaton and each of the learner's m actions has a known inverse, then LIVE can learn with accuracy $1-\epsilon$ and confidence $1-\xi$ the correct model of the environment within $O(n^2\, m\, \frac{1}{\epsilon} \log \frac{1}{\xi})$ steps. Empirically, in comparison with existing learning systems, this approach has shown the advantages of creating problem spaces by interacting with environments, eliminating the bias posed by the initial concept hierarchy, and constructing new vocabulary through actions.

The generality of this approach is illustrated by experiments conducted in various domains, including child learning, puzzle solving, robot simulation and scientific discovery. In the hidden-rule Tower of Hanoi environment with a simulated robot arm and a hand, LIVE learned the correct rules for the puzzle and solved all the given problems. In the Balance Beam environment [46], LIVE discovered the concept of torque when learning to predict the balance of the beam in terms of objects' weights and distances. Finally, in an environment derived from Mendel's pea hybridization experiments [28], LIVE discovered the hidden feature of genes when predicting the colors of peas in the experiments.

1.2 The Methodology

The approach for learning from the environment developed in this thesis has four major components: integrating problem solving with rule induction; creating rules through ex-

ploration; revising rules by complementary discrimination; and discovering hidden features from the environment. Since all of these methods are developed within the same framework, they work together to accomplish the task of learning from the environment.

1.2.1 Integrating Problem Solving with Rule Induction

The thesis has specified a formal definition of learning from the environment. In the definition, the input to the learning includes the *goals* to be achieved, the learner's *percepts* and *actions*, and *constructors* for organizing the percepts and the actions. The output of the learning is a set of rules that correlate actions and percepts for solving the given problems. For example, in learning how to use a vending machine, your goal is to get the desired food; your actions include, among others, pushing buttons and depositing coins; your percepts include, among others, the position of the food and the numbers on the panel; and your constructors include the relation "=" for relating the numbers on the panel to the numbers of the food. All these elements are integrated to form the rules for operating the machine.

Based on the definition, this thesis develops a theory of learning from environment as a special case of Simon and Lea's unified theory [51] for problem solving and rule induction. Unlike dual spaces models investigated previously, the instance space here is a sequence of environmental states linked by the learner's actions along the time line. The rule space is a combination of two subspaces, the percept subspace and the action subspace; the rules to be learned are connections between these two subspaces.

This thesis also constructs the LIVE architecture to implement the theory. In this architecture, actions are always associated with predictions, and the learner can alternate its attention between problem solving and rule induction. In particular, when no rule is available for the current goals, the learner explores the environment to create new rules; when faulty rules are detected during planning, the learner designs experiments to find information about how to correct the rules; when predictions fail during plan execution, the learner revises the relevant faulty rules by analyzing the failure, defining new relations, and discovering hidden features when necessary.

1.2.2 Rule Creation through Exploration

Learning from the environment is assumed to be knowledge poor. Therefore, exploration is necessary to gather more information from the environment when the learner has no idea

how to achieve the given goals. For example, faced with a mysterious vending machine, you will probably give it a kick or push some buttons. In this thesis, an exploration is a sequence of actions. New rules are created by noticing the changes in the environment as actions are taken. For example, after you have kicked the vending machine, a rule saying "kicking makes a sound" may be created. After you have pushed a button (saying it is 3), a rule saying "pushing a button causes a number to appear on the panel" may be created. Since exploration involves more than just random actions, a set of heuristics is proposed to make the exploration more fruitful.

1.2.3 Rule Revision by Complementary Discrimination

In this thesis, rules created during exploration are as general as possible. For example, when the rule about pushing a button is created, it assumes that pushing *any* button will show a number. It may be, however, that some buttons cause English letters to be displayed. Therefore, newly created rules are often over-general and incomplete. However, since these rules will make predictions, they provide a springboard for further learning from the environment. Because of their over-generality, the learner will have chances to be surprised, to make mistakes, and hence to increase its knowledge about the environment. To revise incomplete rules, this thesis has developed a method called *complementary discrimination* that discriminates the rules' failures from their successes. For example, using the rule for pushing a button, you may push some particular button, say B, and predict that a number will show up. After seeing B instead of 3, you may compare this rule application with the last and notice that the button pushed previously was a number, while the button just pushed now was not. Therefore, the old rule will be split into two rules: one says that if a numbered button is pushed then the panel will show a number, and the other says that if a lettered button is pushed then the panel will show a letter.

Although the learning method comes from discrimination, it performs both generalization and specialization because both the concept and its complement are learned simultaneously. Generalization of the concept is achieved by discriminating (specializing) its complement, and vice versa. Thus, the method is more complete than pure discrimination methods [7, 15, 24] or pure generalization methods [60]. It has some advantages over other "two-way" concept learning methods such as Version Space [**?**] and Decision Trees [38] because it can learn both conjunctive and disjunctive concepts and because its learning is

incremental.

1.2.4 Discovering Hidden Features in the Environment

Although complementary discrimination is a powerful learning method, perhaps the most novel part of this thesis is the extension of this method for discovering hidden features in the environment.

For most existing learning programs, the ability to learn concepts is limited by the concept description language. Hidden features pose a problem for such programs. These are features that are beyond the learner's perceptual ability but crucial for learning the correct rules. One example is the concept of "torque" that must be defined in terms of weight and distance. Another example involves finding the genes that determine children's characteristics. To discover the hidden features, this thesis has decomposed the task into three subproblems: to know when such hidden features are necessary, to define these features in terms of existing percepts, actions and constructors, and to assimilate them so that they can be useful in the future. Solutions for each of them have been developed.

Since our primary learning method is discrimination, it is relatively easy to know when new features must be discovered. If the learner is ever in a situation where no difference can be found between a success and a failure, as when two pairs of green peas look exactly the same but produce different offspring, then it is certain that the environment has hidden features. The reason is simple: although two states may appear identical, there must be something unobservable that distinguishes the states; otherwise the same action should produce the same results.

What distinguishes the discovery method presented here from previous results in this area, such as BACON [25] and stabb [**?**], is that in addition to discovering features that are expressible in the existing terms and constructors, this method is also capable of discovering features that must be defined recursively through actions (such features are normally inexpressible in the learner's concept description language alone). One such example is the discovery of genes through hybridization.

1.3 Scientific Contributions

Following is a list of the major contributions this thesis makes to the existing scientific knowledge on AI:

1. *Creation of problem spaces by interacting with the environment* (in contrast to understanding written problem instructions [17]). One primary result is that problem spaces are not created from nothing, but are extracted from an immense "raw" space determined by the learner's innate physical abilities and prior knowledge, including its actions, percepts and the relations and functions that it already knows. This extraction process is guided by the information gathered during problem solving and can be classified as a special case, with some interesting new properties, of Simon and Lea's dual space theory [51] for unifying problem solving and rule induction.

2. *Rule induction with complementary discrimination.* By learning both the concept and its complement simultaneously, this new learning method is more complete than any "one-way" discrimination [7, 15, 24] or generalization [60] methods because it performs both generalization and specialization. It has advantages over many "two-way" learning methods [38, **?**] because it can learn both conjunctive and disjunctive concepts, and its learning is incremental. Moreover, the method provides a view to unify learning from similarity and learning from difference because similarities to a concept are the same as differences to the concept's complement.

3. *A plausible architecture for integrating exploration, learning, problem solving, experimentation, and discovery.* Based on interaction with the environment and the ability to detect errors, this architecture alternates its activities between problem solving and rule induction, including experimentation. Compared to most existing integrated systems [30, 52], the architecture emphasizes the correlations between percepts and actions and incorporates exploration and discovery.

4. *A method for discovering hidden features in the environment.* As a natural extension of complementary discrimination, the discovery method developed here has provided an approach for overcoming the problem of incomplete concept description languages [**?**] and for discovering new terms [25] (also known as constructive induction [10]). Furthermore, by classifying environments as time-dependent and time-independent, this thesis has identified a class of hidden features that must be defined recursively through actions. Such features could yield some new insights for the discussion of theoretical terms [48] in the philosophy of science.

1.4 Thesis Organization

There are eight chapters following this introduction. Chapter 2 defines learning from the environment and analyzes its assumptions and complexity. Chapter 3 develops a theory for learning from the environment and points out its interesting properties. Chapters 4 through 7 present the LIVE system designed for learning from the environment. In particular, Chapter 4 surveys LIVE's architecture and describes its internal knowledge representation and its problem solver. Chapter 5 discusses how LIVE creates new rules through exploration. Chapter 6 explains the complementary discrimination method and how it is used to revise LIVE's incorrect rules. Chapter 7 illustrates LIVE's method for discovering hidden features in the environment. Chapter 8 presents some experimental results of applying LIVE to the various environments. Finally, Chapter 9 summarizes the thesis and outlines some directions for future work.

Chapter 2

The Problem of Learning from the Environment

In this chapter, we will define the task of learning from the environment and give a set of assumptions that constrain the work in this thesis. We will also analyze the complexity of the task from the viewpoint of learning.

2.1 The Definition and Two Examples

To study learning from the environment is to study how a system learns to solve problems in an unfamiliar environment. In this thesis, we assume the task is given in the form of four components: an environment, a learner, an interface, and some goals. The objective of learning from the environment is for the learner to build an internal model of the environment and use it to achieve the goals. The internal model must be good enough to solve the problem, but need not be a perfect model of the environment.

Table 2.1 gives the definition of learning from the environment. The *environment* here is what Simon called the "outer environment" [47]. It provides a mold for the learner, the "inner environment," to adapt. The function of the environment is to take actions from the learner, determine the effects of the actions, and then return the effects (not necessarily all the effects are visible to the learner) in a form that can be perceived by the learner. The function of the learner, which is driven by the given goals, is to select actions based on its internal model and send the actions to the environment in the hope that progress toward the goals will be made. At the outset, the learner's internal model is empty, or at least inadequate, for the given environment and the goals.

Given

- Environment: A process that takes an action from the learner, determines the effects, and returns the effects in terms of percepts;
- Actions: Commands that may cause changes in the environment;
- Percepts: Information about the environment that is perceivable by the learner;
- Constructors: A set of primitive relations (e.g. $=$, $>$), functions (e.g.$+$, $*$), and logical connectives (e.g. $\wedge, \neg$, and $\exists$);
- Goal: A description of a state of the environment in terms of percepts;

Learn

- An internal model of the environment sufficient to accomplish the given goals.

Table 2.1: The definition of learning from the environment.

The environment and the learner are linked by the interface, which consists of *percepts* and *actions*. The percepts constrain the way the learner sees the environment (which may have more states than are seen by the learner), and the actions constrain the way the learner affects the environment (which may be changed by means other than the learner's actions). Since the interface shapes the task of learning from the environment to a large extent, we shall define the meaning of actions and percepts precisely.

An action is an internal change of the learner; it is independent from its consequences in the environment. For example, if the learner is a robot with a switchable magnetic hand, then "turn on the magnetic field of the hand" is an internal change, an action. But whether anything will be picked up by the hand depends on the current environment, for example, whether there are any movable metal objects near the hand. The reason we make this distinction between actions and their consequences is that we can separate the learner from its environment. In this way, our solution will be useful in the future for constructing an autonomous system to learn from a wide range of environments.

A percept is a piece of information about the environment that can be perceived by the learner. Percepts can be relations of objects, such as block A is on block B, or features of objects, such as the color and the shape of block A, or a representation about the surrounding space, such as Moravec's Certainty Grid [33]. In the definition, we do not

restrict the form of percepts. In the rest of this thesis, however, we will only use the first two kinds of percepts. Relations of objects will be specified as predicates, and features of objects will be represented as feature vectors.

Besides the functionality of interface, the actions and the percepts also provide the building material for constructing the internal model. For example, if "nearby," "in-hand" and "metal" are the percepts, and "turn on the magnetic field of the hand" is the action, then we can partially model the law of the hand's pick-up phenomena by saying that if nearby(hand,x) and metal(x) are perceived, then the action will cause the object x to be in the hand.

To build the internal model from the percepts and actions, we also need *constructors*, or linking material, such as basic relations ($=$, $>$), primitive functions ($+$, $\times$), and logical connectives ($\wedge$, $\neg$). In the example above, $\wedge$ is the constructor that links nearby(hand,x) and metal(x) together. The functionality of constructors are like that of nails, glue, bolts and nuts when one is building a house from bricks and wood. An interesting fact about the constructors is that they can also be used as constraints to focus the attention of the learner. Suppose, for example, that "mass" and "acceleration" are given as percepts, and "force" must be learned in order to master the environment. Providing $\times$ (multiplication) as a constructor with few other functions will be much better than giving the learner a hundred functions to consider.

In principle, the constructors for linking actions are also needed. For example, to learn macro actions like "move the hand to an object and then pick up the object," the constructor "composition" must be given so that a sequence of actions can be treated as a single action. Similarly, to learn parallel actions like "rotate the arm and at the same time open the fingers," the constructor "parallel execution" must be given. However, since learning compound actions is left open in this thesis, we omit these constructors from the definition.

The last component of the learning task is the *goals*, the driving force of learning from the environment. In this thesis, the objective of learning is not a perfect model of the environment, as in most theoretical work [3, 4, 16, **?**], but a model that is adequate for reaching the goals. For example, in solving the hidden-rule Tower of Hanoi problem (to be defined shortly), once all the disks are on the goal peg, learning stops unless new goals are given. It is possible that not all the rules of the Tower of Hanoi are correctly inferred when the problem is solved. Compared to learning a perfect model, an immediate advantage

is that it provides a natural termination criterion for the learning. When the given goals are reached, the learner believes its internal model is a good one for the environment and learning is stopped until new goals are given or generated. If the goals are excluded from the learning, i.e., a perfect model of the environment must be learned, then finding a termination criterion can be very difficult [**?**].

Figure 2.1: The hidden-rule Tower of Hanoi puzzle.

To illustrate the definition of learning from the environment, let us consider two specific examples, both derived from the familiar Tower of Hanoi puzzle (Figure 2.1). The puzzle has three pegs and n disks, and the goal is to move all the disks from one peg to another without violating the following rules:

- Only one disk may be moved at a time;
- If two or more disks are on one peg, only the smallest disk can be moved;
- A disk cannot be put on a peg that has a smaller disk.

Unlike previous studies of this problem, we require a stronger condition: the rules of the game are not given to the learner at the outset but are enforced by the environment by refusing to carry out any illegal actions. For example, if the learner tries to put a disk on a peg that has smaller disks, it will find the disk remains in its hand regardless of its put action. Although the environment behaves like a teacher, the task differs from supervised learning in one aspect: the environment does not provide the instance and answers automatically. Instead, the learner has to choose the instances and then ask the environment for the

answers.[1]

From this hidden-rule Tower of Hanoi environment, the first example of learning from the environment is defined in Table 2.2. In this case, the learner is given three relational predicates, ON, GREATER and IN-HAND, as percepts; and two macro actions, Move-Pick and Move-Put, as actions to pick up disks from pegs and to put down disks on pegs. Since this environment will be used extensively in the thesis, we will call it the Relational Predicate Macro Action Tower of Hanoi environment, or the RPMA environment for short.

The RPMA Environment

- **Percepts** Three relational predicates ON(*disk, peg*), GREATER(*diskx, disky*), and IN-HAND(*disk*).
- **Actions** Move-Pick(*disk, peg*) and Move-Put(*disk, peg*).
- **Constructors** Three logical connectives: $\wedge$, $\neg$ and $\exists$.
- **Goal** Move all the disks from their initial position to the target peg and construct the rules of the game in terms of the given percepts, actions and constructors.

Table 2.2: The Tower of Hanoi with relational predicates and macro actions.

The second example of learning from the environment is defined in Table 2.3, where the learner is more like a hand-eye robot. Compared to the RPMA environment, the learner here cannot perceive any relations between objects, but only features of individual objects, such as shape, size and location. The actions are more primitive. They control the movement of learner's arm and hand. Two relations, $=$ and $>$, in addition to the logical connectives, are given as constructors. Since the environment is more like a hand-eye version of Tower of Hanoi, we will refer to it as the Hand-Eye environment.

Learning from the environment as defined can be viewed as the automatic creation of a problem space through interacting with an environment. When the goals are reached, the learner has created a problem space in which the laws of the environment are the operators,

[1]The difference is equivalent to the difference between *Experimentation* and *Examples*, which is still an open problem in the theoretical machine learning literature [1].

The Hand-Eye Environment

- **Percepts** Features of the disks and the pegs, including shape, size, direction, distance, and altitude. (Note: The last three features determine objects' locations in the environment, and their values are relative to the location of the robot.)
- **Actions** Rotating the arm, sliding the hand, picking up an object under the hand, and dropping an object in the hand. (Assume the learner is a hand-eye system with a rotatable arm and slideable hand.)
- **Constructors** Two primitive relations, $=$ and $>$ (no relational predicates are given), and three logical connectives, $\wedge$, $\neg$ and $\exists$.
- **Goal** The same as the RPMA environment.

Table 2.3: The hand-eye version of Tower of Hanoi.

and the percepts, including those constructed in the process, are the state description language. A unique character of this creation process is that it is interleaved with the problem solving; the space is created during problem solving, not before. Of course, once actions and percepts are given, a "raw" problem space is already there, but that space is far from relevant to the problem at hand and its size is enormous (even infinite if the environment is continuous). Unless a concise and relevant space is abstracted, solving a problem in that raw space can be extremely difficult, if not impossible.

The process of learning from the environment can also be viewed as inferring the properties of surrounding objects so that objects can be used as "tools" to reach the given goals. Figure 2.2 is an environment adapted from a child development experiment [19]. In this environment, a bar is placed on a rotatable circular platform that is mounted on a table. An interesting toy is fastened at one end of the bar. The bar and table are arranged so that a child cannot reach the toy at his/her side of the table without rotating the bar. The goal of the experiment is to reach the toy. As we can see, the key to the problem is the rotatability of the bar. Object properties like this can only be learned if one interacts with the environment.

Finally, we should point out that learning from an environment does not require the learner to start with complete knowledge about the environment. If there is knowledge, the

Figure 2.2: Learning to use a lever.

learner can certainly take advantage of it by incorporating the knowledge into its internal model. If there is little or no knowledge, the learner should still be able to pursue its goals. After all, learning from an environment is an incremental and continuous process; solving one problem does not mean the learner has mastered the environment. When new problems come along, the learner may or may not be able to solve them with its current knowledge; learning will be required when mistakes are made and deficiency of knowledge is detected.

2.2 The Assumptions

This section states five assumptions that constrain the task of learning from the environment. First, we assume that the environment can only be changed by the learner's actions. This assumption excludes both the interference of other agents, such as other robots or humans, and the existence of natural forces, such as gravity, wind, or current. For example, when a boat is on a river, we assume it will stay motionless unless we row it, even though in the real world it might drift away with the current. Although this assumption is too strong for learning from the real world, it is a first step in that direction. When a good solution is found for this simple environment, we can then relax the assumption as follows. To include other agents in the environment, we can have agents in the environment communicate to each other whenever actions are taken. To include natural forces, we can define a NIL action and then learn the laws about this special action. For example, we can specify the drifting

motion of the boat by saying "if the NIL action is taken, the boat will move downstream."

The second assumption is that the environment is deterministically controlled by an oracle. In other words, once the learner takes an action, the response to that action is uniquely defined by the current state of the environment seen by the oracle. From the learner's point of view, the environment may behave nondeterministically because the learner's percepts may not be powerful enough to perceive complete state information (the oracle may keep some state variables hidden from the learner). For example, parents' genes determine their children's characteristics. But if we do not know anything about genes, the grandchildren's characteristics seem nondeterministic. Although the final answer to this assumption, i.e., whether the real world is deterministic or not, is philosophical, we adopt the deterministic point of view because we want to emphasize learning. If we assume that the world is nondeterministic, then there is no point in asking why predictions about an action fail because the unpredicted outcome can be just another non-deterministic result of the action.

The third assumption concerns the learner's carefulness. We assume that all the changes in the world that can be detected by the learner's percepts will be detected. Note that this is different from the famous closed-world assumption[2] because we do not assume that all the things that cannot be detected are not changing. For example, suppose in a Tower of Hanoi environment we can sense the relations (ON *disk peg*) and (GREATER *diskx, disky*). We will sense every possible ON and GREATER relation in the world (which might be impossible in the real world), but we do not assume that relations we cannot see, such as (IN-HAND *disk*), are never changing.

This assumption is expensive because sensing all the changes within one's sensing ability can be computationally intractable. For example, to describe the effects of striking a key on my terminal, we do not check if the Moon is full in Beijing today, even if we could. (It is within our sensing ability because of overseas communication.) However, this is a useful assumption, especially when environments possess hidden features to be discovered. Without it, the existing features are beyond one's examining ability, and discovering new features is out of the question.

Although this third assumption is weaker than the closed-world assumption, there are still many things we are unable to learn. In particular, we cannot learn laws that depend

[2]The "closed-world assumption" [6] says that all the relationships not explicitly stated to hold do not hold. In STRIPS's language, actions change none of the program's beliefs about the world except those explicitly listed in the description of the action.

on some facts that are totally irrelevant to our percepts and actions or that lie outside of the constructible space that is determined by the given constructors. For example, if the predicate "full moon" is neither changeable by our actions (e.g., picking a disk up causes the moon to become full, putting the disk down causes the moon otherwise) nor constructible in terms of our given percepts and constructors (e.g. "full moon" is true iff 3 * 2 = 5 where * and = are given), then we cannot learn a law that says "if the moon is full, then we cannot pick up any disk." In other words, this assumption enforces the learner to rely heavily on the fact that the effects of an action are caused by the objects that are related to the action. In the Tower of Hanoi environment, a bigger disk cannot be put on a smaller one because of the features of the disks, not because today is Friday.

The fourth assumption is that the recognition process is made easy. In other words, if the learner sees a cup at time t, we assume it can recognize the cup in the future whenever it sees the cup again. This assumption enables us to bypass the difficult recognition problems.

The fifth and the last assumption is that the environment is noise free. Although we allow small measurement errors (e.g., we could consider 3.45000 equal to 3.45003 in the Hand-Eye environment if the threshold is 0.0005), we assume that the environment oracle always tells the truth. For example, if the learner is told at time t that x belongs to a concept C but at time $t + i$ $(i > 0)$ x does not belong to C, then it will not assume that the environment has made a mistake but assume the environment knows something it does not know. From a concept learning point of view, this requires that the learner always get the correct answers from the environment even though such answers may seem very wrong according to the learner's current knowledge.

2.3 The Complexity

Although we have argued that learning from the environment should not be separated into isolated pieces and have given a definition that manifests our belief, it is helpful, if not necessary, to define the complexity of learning from the environment because finding a complete solution to the problem would involve enormous work and we must move from simple to complex.

In a sense, learning from the environment is like solving a jigsaw puzzle. Its complexity depends on the relative difference between the size of the picture and the sizes of the pieces. For example, from the same picture, it is easier to assemble a puzzle if the pieces are big (only

a few pieces), and harder if the pieces are small (many pieces). Similarly, given pieces of the same size, a puzzle is easier if the picture is small and harder if the picture is big. Based on these intuitions, the first two factors of the complexity measurement (Table 2.4) are the complexity of the given percepts and actions relative to the complexity of the environment. We call these the *primitiveness* of percepts and actions, relative to the environment.

- The *primitiveness* of the percepts relative to the laws of the environment;
- The *primitiveness* of the actions relative to the laws of the environment;
- The number of *hidden features* in the environment.

Table 2.4: The complexity of learning from the environment.

The third complexity factor concerns the sufficiency of percepts and actions for constructing the laws. This is analogous to whether there are enough pieces to complete the jigsaw puzzle. Although solving a jigsaw puzzle with missing pieces is no fun, natural environments do have features that are hidden. For example, two pairs of green peas may look exactly the same, but they could produce different offspring (because they have different genes) even when they are fertilized under the same conditions. Naturally, the more hidden features an environment has, the more difficult learning is in that environment.

The three complexity factors are best illustrated by examples. To see how the primitiveness of percepts influence the complexity of learning from environment, consider a simple version of hidden-rule Tower of Hanoi puzzle in which the learner is given one action: Move(*disk pega pegb*), for moving *disk* from *pega* to *pegb*, and two percepts: On(*disk peg*), indicating *disk* is on *peg*, and Free-to-Move(*disk pega pegb*), indicating *disk* is free to be moved from *pega* to *pegb*. Based on these action and percepts, one of the laws of the environment can be written as follows:

Condition: On(*disk pega***) and Free-to-Move(***disk pega pegb***) Action: Move(***disk pega pegb***)** **[Rule-A] Prediction: On(***disk pegb***)**

The complexity of learning this rule increase when the learner's percepts become more primitive than the two above. For example, if Free-to-Move is replaced by Top(*disk peg*) and Free(*disk peg*), then learning becomes more complex because Free-to-Move must be built from Top and Free (see Figure 2.3). Moreover, if On(*disk peg*) and Greater(*diskx disky*)

are the only two percepts given to the learner, then the complexity increases even further because both Top and Free must be built. The matter can be more complicated if the learner is not given any relational predicates. For example, if all that can be sensed are objects' features, like shape, size and color, then learning is more difficult because relational predicates, such as On and Greater, must be built from these features.

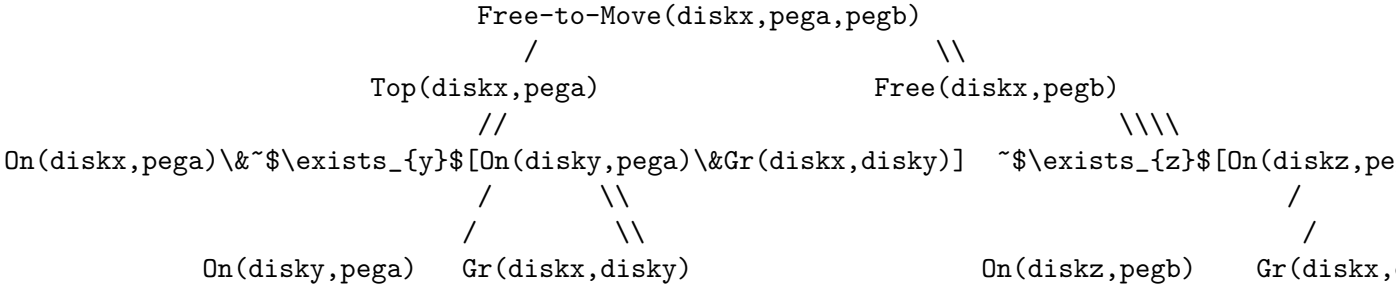


Figure 2.3: The primitiveness of percepts and the complexity.

Just as the primitiveness of percepts increases the complexity, so does the primitiveness of actions. In the example above, if the action Move(*disk pega pegb*) is replaced by two more primitive actions Move-Put(*disk peg*) and Move-Pick(*disk peg*), then learning becomes more complex because more laws must be learned and more actions must be taken in order to achieve the same effects. For example, Rule-A must be learned in terms of two rules, Rule-B and Rule-C:

Condition: On(*disk peg***) and Top(***disk peg***) Action: Move-Pick(***disk peg***) [Rule-B]**
Prediction: In-Hand(*disk***) and ¬On(***disk peg***)**

Condition: In-Hand(*disk***) and Free(***disk peg***) Action: Move-Put(***disk peg***) [Rule-C]**
Prediction: On(*disk peg***) and ¬In-Hand(***disk***)**

In order to move a disk from one peg to another, two actions, Move-Pick and Move-Put, must be executed instead of one. Clearly, when actions become more primitive, (e.g., Move-Pick and Move-Put are further replaced by actions Rotate(θ), Slide(d), Pick(), and Drop(), as specified in the Hand-Eye environment), learning becomes more difficult.

Interestingly, the more primitive the actions, the more independent they are of the environment. For example, Move(*disk pega pegb*) can only be executed in the Tower of

	Move(x, a, b)	Move-Pick(x, a) Move-Put(x, b)	Rotate(θ), Pick() Slide(d), Drop()	Real Mobile Robots
On(x, a), Free-to-Move(x, a, b)	TOH-1			
On(x, a), In-Hand(x), Top(x, a), Free(x, b)		TOH-2		
On(x, a), In-Hand(x), Greater(x, y)		TOH-3		
Shape, Size, Direction Distance, Altitude			TOH-4	
..... Real Camera Images				TOH-n

Table 2.5: The Tower of Hanoi with different complexity.

Hanoi environment, but Rotate(θ) and Slide(d) can be executed in any environment for the robot. This fact is very useful for separating the learner's actions from the environment. If two learners have the same learning ability, then we will prefer the one with more primitive actions and percepts because it is more environment independent.

Finally, as an example of how the third complexity factor, the hidden features of an environment, affects the complexity of learning, suppose that In-Hand(*disk*) is not given when learning Rule-B and Rule-C. This will increase the complexity in two senses. First, if there is a hidden feature, the environment will behave nondeterministically from the learner's point of view. For example, without In-Hand(*disk*), the robot has no way to tell whether a disk, when it is not on a peg, is in the hand or on the table. As a consequence, Move-Put(*disk peg*) will sometimes success and sometimes fail depending on where the disk is. Second, if the hidden features of an environment are necessary components for constructing correct laws, the learner will be forced to discover the hidden features in addition to learning the correct rules. For example, to infer Rule-B and Rule-C, In-Hand(*disk*) must be discovered as well. (This is an artificial example because the In-Hand relation is very easy to detect.)

The three complexity factors discussed so far constitute a complexity space for learning from the environment. If we list all the versions of Tower of Hanoi problems we have talked about so far in the Table 2.5, their relative positions in the space can be shown in Figure 2.4. Note that TOH-3 is the RPMA environment and TOH-4 is the Hand-Eye environment, as defined in Section 2.1.

Figure 2.4: The complexity space for learning from the environment.

Chapter 3

A Theory of Learning from the Environment

This chapter develops a theory for learning from the environment by arguing that such learning is both creating problem representations and solving problems with incomplete knowledge. Since the theory is a special case of Simon and Lea's dual space theory [51], we will point out those interesting properties that differ our theory from most dual spaces that have been investigated.

3.1 Creating Problem Representations

According to the definition of learning from environment, the learner's task is to create an internal model of the environment adequate for solving the given problem. But what is this internal model? Why should it be created instead of given? If it is to be created, where does it come from? These are the three questions to be answered in this section.

3.1.1 Problem Spaces Must be Created

Since the purpose of learning from the environment is to solve problems, the internal model to be created from the environment is equivalent to the *problem space*. As first defined by Newell and Simon [**?**], a problem space is a graph-like problem representation which consists of states and operators. The nodes of the graph are states of the environment, and the edges of the graph are operators available to the problem solver. Each problem is defined by designating an initial node and a set of goal nodes, and solving a problem is finding a path from the initial to the goals. For the Tower of Hanoi problem, if we assume

the only action available to the problem solver is Move(x,A,B), which stands for moving a disk from one peg to another, then Figure 3.1 shows its problem space [**?**]. In this space,[1] the nodes are the legal states of the Tower of Hanoi, and there exists at least one path between any two nodes. Once such representation has been identified for a problem, solving the problem can be described formally as a search for paths in the problem space from the initial node to the goal node.

Figure 3.1: The Tower of Hanoi problem space.

Most prior research in problem solving and learning has assumed that the problem space is given to the system at the outset. However, experiments in psychology [17, 20, 50] have revealed some evidence that a good problem space, like the one shown in Figure 3.1, does not exist in human problem solver's head when he/she is given the Tower of Hanoi for the first time. Such a mature space is created by the solver as a part of the problem solving activity. Simon, Hayes and Kotovsky [20] have shown that some isomorphic problems (problems whose mature spaces are structurally equivalent) have different levels of difficulty for human problem solvers. For example, if the Tower of Hanoi puzzle is given in a different form, say that the pegs are monsters, the disks are crystal globes, and globes can be transferred from one monster to another, then a human problem solver will take much longer time to solve

[1] The numbers indicate the pegs, and the positions indicate the disks. For example, (3 2 2) means disk1 is on peg3, and disk2 and disk3 are on peg2.

it. Moreover, when solving the monster-globe problem, human problem solvers will more frequently run into illegal states (nodes not shown in Figure 3.1). One explanation for this phenomena is that the mature space for the monster-globe problem is more difficult to construct than the mature space for the Tower of Hanoi. Otherwise, the monster-globe problem should take the same time to solve as the Tower of Hanoi puzzle because their mature spaces are identical.

In the example above, problems are presented to the solver in the form of written instructions. If no problem instruction is given, as in learning from environment, then creating problem spaces becomes more important and carries more weight in the process of problem solving. Imagine yourself given the Tower of Hanoi puzzle for the first time without being told the rules. The space you start with is much greater than the one in Figure 3.1. You probably have many more actions other than Move(x,A,B). For example, you may pick up a disk and put it on table, you may switch the positions of pegs instead of moving disks, or even more ridiculously you may turn the whole puzzle 180 degrees so that the goal state is reached in one step (assume you are asked to move all the disks from the left peg to the right peg). Besides the freedom of many possible actions, there are many more states of the environment than those shown in Figure 3.1. For example, disks can be in your hand and your hand can be in many positions; disks can be on the table instead of on a peg, and larger disks can be on smaller disks. Obviously, solving problems in such situations requires creation of a problem space.

3.1.2 Where Do Problem Spaces Come from?

To create problem spaces, we must understand where these problem spaces come from. Two possible sources are manifest: to understand written problem instructions, or to exploring the environment. In both cases, the solver's prior knowledge plays an important role.

When an unfamiliar problem is given in written instructions, a human problem solver will read the instructions and extract information to construct an internal problem representation. When the solver thinks the representation is adequate for solving the problem, he/she will start to search for solutions in that representation. Sometimes, the person will go back to read the instructions again when the problem solving process reveals insufficiency in the representation. Hayes and Simon [17, 50] first studied these phenomena and constructed a computer program, called UNDERSTAND, that simulates the processes that

people use to generate an internal problem representation from written instructions. In that program, prior knowledge is cast as list structures and list operations, and these are the building blocks for an internal representation. The output of the program is list representations of problem states, operators that change states, and tests for the presence or absence of particular features. The criteria for whether the representation is adequate or not is whether it is sufficient for a general problem solver GPS to start running and solve the problem. The UNDERSTAND program works like a compiler, for it restates the given problem in terms of his own internal assembly language (the list structure and list operators) that runs on its own problem solving engine.

If no instruction is given, then interacting with environment becomes the primary source for creating problem spaces. Since the problem solver may not know the consequences of his actions, he relies on the interaction with environment to correlate his senses and actions and create the states-operator representation. In this case, as in the example of children learning the lever (see Figure 2-2), the building blocks are the learner's actions and percepts.

Theoretically, the learner's actions and senses can be viewed as a very primitive problem space (senses determine the possible nodes in the state graph and actions determine the links). Therefore, strictly speaking, the problem space for solving a given problem is not "created" but extracted from a raw, usually much bigger, space that is determined by the learner's physical abilities.

3.2 Problem Solving with Incomplete Knowledge

Since learning from the environment is to solve given problems, the learner has to search for solutions once a problem space is created. However, since there is no guarantee that the created problem space is adequate for the given problem, the learner may lack of some important knowledge for finding a solution and he must cope with incomplete knowledge during his problem solving. For example, when learning how to use a lever, in Figure 2-2, a child knows his goal is to get the toy, but he does not know the fact that rotating the bar will change the location of the toy, a piece of information that is crucial for solving the problem.

3.2.1 The Completeness of Knowledge

In order to see why learning from the environment is to solve problem with incomplete knowledge, it is necessary to define the completeness of knowledge.

Whether knowledge is complete or not varies with the purpose of the knowledge. For example, for an explanation-based learning system, we say its knowledge is complete if it can successfully construct an explanation tree when a training example is given. For learning from the environment, since the purpose is to solve problems, its knowledge completeness has to do with finding solutions. Furthermore, solving a problem can be by "rote" or by "reasoning", and their completeness of knowledge is different. For example, to solve the lever-toy problem, a robot might remember, by rote, a sequence of actions, such as rotate the arm to left for 40 degree, forward the arm 0.8 meters, rotate the arm to right for 90 degree, withdraw the arm for 0.5 meters, low the hand, and close the fingers. In this case, remembering the action sequence can be considered as complete knowledge because it solves the problem. On the other hand, if the problem is to be solved by reasoning, the robot must have reasons for each of its actions. For example, the purpose of rotating the arm to left for 40 degree is to rotate the bar so that the toy will move closer. In this kind of problem solving, simply remembering actions is not complete knowledge. Simon [49] has studied the different strategies used by human in solving the Tower of Hanoi puzzle, and discovered that all the reasoned strategies are isomorphic to means-ends analysis.

Since means-ends analysis will be the main problem solving method in this thesis, we define completeness of knowledge as the necessities of this method [17]:

1. State representation for the problem situations, including the initial and the goal states. In the Tower of Hanoi puzzle, these can be list structures like ((1 2 3) () ()) (the numbers are disks, and the pegs are the sublists), or a set of predicates like (ON 1 1), (ON 2 1) and (ON 3 1), where (ON x y) means disk x is on peg y.

2. Actions with their applicable conditions for changing one state into another. In Tower of Hanoi, an example of these will be "If the goal is to have disk x on peg B and x is the smallest disk on peg A and there is no smaller disks than x on peg B, then Move(x,A,B)".

3. Differences for describing states and comparing pairs of states. In Tower of Hanoi, an example of differences is Blocked-by(x,y), meaning disk x is blocked by the disk y.

4. A table of connections between differences and actions, which prescribes the possible actions that may be taken when each kind of difference is encountered. For the Tower of Hanoi problem, an example of an entry in the connection table is that "if Blocked-by(x,y), then do the action Move(y, peg-of(y), other-peg(peg-of(y),goal-peg)."

With completeness of knowledge so defined, we can see clearly why the knowledge in learning from the environment is incomplete at the outset. For instance, the actions and the percepts for learning from the environment are given separately. The solver does not know when an action can be applied and what the consequences will be (there is no predefined operators and no connection tables). Further more, even though the solver can perceive information from the environment and represent the state using the percepts, it does not know what differences are important or relevant when comparing two states. For example, in the definition of RPMA environment, predicates like Blocked-by are not given.

3.2.2 Detecting and Refining the Incomplete Knowledge

Because of the incompleteness of knowledge, problem solving in learning from environment has some unique characteristics. Besides finding a solution for the given goals, the problem solver must also detect the deficiencies and errors in its knowledge and correct them accordingly. Thus, for learning from the environment, a problem solver must be able to utilize whatever knowledge it has to search for solutions, and at the same time detect the incompleteness of existing knowledge.

One symptom of incomplete knowledge is not knowing what to do in order to achieve the given goals. If this happens when we are solving problems by reading instructions, we can go back to read the instructions again in order to get clues to pursue further. If this happens when we are learning from environment, we must explore the environment in order to get more knowledge. For example, in the child learning lever situation, when he discovers that the toy is too far to be reached, he plays with objects on the table. Of course, such exploration is far from random. A child will not just wave his hands in air, or stamp his feet on the floor. The exploration, innate or not, is guided by his previous knowledge so that he will interact with objects that are interesting to him.

Besides total lack of ideas, incomplete knowledge has another symptom, namely, actions (under certain conditions) produce surprising results (something unexpected). One way to detect such incompleteness is to make predictions whenever actions are to be taken and

check the fulfillment of predictions after actions are executed. This requirement casts some light on the format of the knowledge for learning from the environment. It is better to associate with each action both its conditions and its predictions. In fact, that is the reason we will choose our representation (see Chapter 4) as three component STRIPS-like rules (with conditions, actions and predictions packed together), instead of two component productions[2].

Once the incompleteness of knowledge is detected in the problem solving, it is obvious that the problem space must be further developed. Thus, one should switch activities to refining the problem space before continuing to search for problem solutions.

3.3 Can We Do Them Separately?

We have said that learning from the environment is both creating problem spaces through interacting with the environment and solving problems with incomplete knowledge. However, can these two activities be done separately?

If problems are given as instructions, then it might be possible to create a correct problem representation before solving the problem. This is because the written instructions provide both information and termination criteria for the construction, and the solver can create an adequate representation by making sure he understands all the given instructions. However, even in this case, as Simon and Hayes [17] have shown, there are some evidences that human problem solvers use the two processes alternately. They start to solve the problem without understanding all the instructions, and later come back to read again when they make mistakes or have troubles finding solutions.

When learning from the environment, it is much harder to separate these two processes. One reason is that there is no predefined termination criteria similar to understanding instructions. The other reason is that there are some intrinsic relations between these two processes. Creating a problem space is an information gathering process, and solving a problem is an information application process, neither of which alone is capable of learning from the environment. Without the former, on the one hand, the later can only be a blind search in the raw problem space. Without the latter, on the other hand, the former would not know what to look for and what is relevant. Moreover, since there is no given criteria

[2]Production rule can make and check predictions too, except it will need two rules, one for making prediction and the other for checking the prediction.

for the construction, the former relies on the latter to tell whether the problem space has been adequate or not (a solution is found or a mistake is made). When problem solving finds itself in a position where it does not know what to do, it must turn to the creation process for more information.

3.4 The Theory

Based on the analysis carried out so far, we are now ready to present a theory for learning from the environment:

> Learning from the environment is a process of creating internal problem representation from an initial problem space determined by the learner's percepts and actions, and such creation process must be integrated with the process of problem solving through interacting with the environment.

This theory is an interesting special case of Simon and Lea's dual space theory [51] that unifies rule induction and problem solving. In Figure 3.2, we have viewed learning from the environment as such two spaces. Comparing with most dual spaces that have been investigated so far, both instance space and rule space here have some interesting characteristics. Here, the instance space is *time dependent*, i.e. the space is a sequence of environmental states that linked by the learner's actions instead of a non-ordered collection of objects. In a time dependent environment, any instance depends not only on the action that is taken but also the history of the environment. For the rule space, the rules are not just classifiers, they are rules that correlate learner's actions and percepts. Thus, the space can be viewed as connections of two subspaces, the percepts subspace **P**, and the action subspace **A**.

The interactions between these two spaces are in both direction. On the one hand, the learner, based on the search status in the rule space, can apply its actions to produce instances in the instance space. Such actions can have several purposes. It may be for gathering information from the environment (exploration), for achieving certain environment state (problem solving), or for falsifying some incorrect rules (experiments). On the other hand, the environment provides the state information (not just a yes or no answer) to the learner after an action is taken. Such feedback is used by the learner to either construct new rules (when the action is explorative), or check the fulfillment of its prediction. If the

Figure 3.2: The dual spaces for learning from the environment.

prediction failed, the learner will use this information to analyze the failure and revise the incorrect rules that made the prediction.

To correlate actions and percepts, each of the rules that constitute the rule space (or the internal problem space) has three components (see Chapter 4 for detail): the condition (in terms of percepts), the action (in terms of actions), and the prediction (in terms of percepts). The action subspace $\mathbf{A}$ is for constructing compound actions (e.g. macro actions), the percept subspace $\mathbf{P}$ is for constructing compound percepts (e.g. logical expressions or concepts). The primitives in the action subspace are the actions that available to the learner, and the generator in that space are the constructors that combines actions in meaning groups. For example, if sequential composition is a constructor, then all the possible macro actions will be in the $\mathbf{A}$ subspace. If "parallel execution" is a constructor, then all the parallel actions will be in the $\mathbf{A}$ subspace too (see some related work in [43]). Similarly, the primitives in the percept (or concept) subspace $\mathbf{P}$ are the percepts that are available to the learner, and the generators are the given constructors, such as logical $\wedge, \neg$, relations $=, >$, and functions $+, *$. In principle, any concepts that are expressible by the given percepts and constructors can be constructed in this subspace.

Because actions and percepts are correlated by the rules and the instances in the environment are time dependent, this special dual space has a unique characteristic: it is capable of constructing, or discovering, hidden features in the environment that are not expressible in the percept subspace $\mathbf{P}$ alone. As we have mentioned in the definition of learning from the environment, it is possible that some information about the environment is not perceivable by the learner. If that is the case, then the learner may find itself in a

situation where applying the same action in two "identical" states will produce different outcomes. Therefore, hidden features must be discovered, otherwise the learner will never be able to predict the outcomes correctly. Fortunately, since the instance space is time dependent, the learner can search back into history to find out the reasons in the states preceding the two indistinguishable states and hypothesize the hidden features that inherited through actions. Since such features are not definable in the percept subspace alone, they must be constructed by a cooperation between actions and percepts. Chapter 7 presents this kind of learning in detail.

Therefore, the theory of learning from the environment unifies many learning phenomena in a single framework. It includes not only problem solving and rule induction, but also exploration, experimentation and discovery. Since the learning has a sense of time and emphasizes the correlation between percepts and actions, hidden features that require both actions and percepts can be discovered.

Finally, since the criteria of learning from the environment is to solve problems rather than to infer the whole world, the theory presented here may provide some insights on how to link theoretical machine learning research to learning in real-life. Previous theories on inductive inference [4] emphasize learning the whole environment, and thus most interesting problems are classified as NP completete or intractable. The need for some new criteria for machine learning have been realized recently by many theoreticians. Along this line, Valiant [57] proposes the *probably approximately correct* as a more practical criteria, and Rivest and Sloan propose the reliability and the usefulness [39]. Our theory can be viewed as another possible practical criterion for machine learning, although no theorems have been proved in this thesis.

3.5 Related Work

Besides the work we have mentioned in this chapter, there is much work, both theories and systems, that are closely related to the theory presented here.

In psychology, two famous theories related to ours are Piaget's child development theory [36, 37] and Gibson's perception with actions [**?**]. Piaget's insightful and detailed experiments on child development have revealed many interesting results, some of them are surprising. For instance, object permanency is learned instead of innate. In some sense, our decision that learning from the environment is based on percepts and actions is motivated

by his theory. Gibson, studying how the world is perceived by humans, pointed out that perception must cooperate with actions. Objects are most often perceived in terms of their affordance (what they can do for the perceiver), not just what they look like. This supports our beliefs that some features must be learned in terms of both percepts and actions, a factor that is often ignored by most AI concept learning systems.

In Artificial Intelligence, Alan Newell [35] has recently proposed a unified theory for cognition, in which the major learning method is "chunking" and the long term memory is assumed to be non-penetrable, i.e. new knowledge can be added to the memory but the existing knowledge cannot be changed. For learning problem strategies from the environment, Anzai and Simon [5] have constructed a program that simulates closely the behavior of an intelligent human subject in acquiring sophisticated strategies for solving the Tower of Hanoi problem, using information gathered during solution attempts made without the help of these strategies. Mitchell *et al* have constructed the LEX system [32] which learns problem solving heuristics by experimentation based on Version Spaces [**?**], an algorithm well known for learning conjunctive rules with a given concept hierarchy. In the PRODIGY project [30], Minton [29] has applied explanation-based learning on learning search control knowledge, and Carbonell and Gil [9] have proposed a method to learn better domain knowledge by experimentation.

In his thesis, Shrager [45] has studied the problem of instructionless learning, a phenomenon that is closely related to learning from the environment in nature. However, his study is more concerned about how a learner changes its current theory by applying his *views*, i.e. the existing knowledge, to the current problem, and his learning method emphasizes, when failures occur, more on what the environment did rather than why the existing hypothesis failed.

Learning by discrimination was first used by Feigenbaum and Simon [15] in their EPAM program. Vere [58] also uses it as counterfactuals for learning relational production, including concepts. Langley [24] utilizes it in his SAGE system and develops a theory of learning by discrimination. More recently, Falkenhainer [14] uses discrimination as disanalogy, and Newell and Flynn [35] use it to modify incorrect productions in SOAR.

With respect to automatic creation of a problem space, Hayes and Simon [17] have built a system capable of constructing problem spaces from written English, which is extended by Yost and Newell [35] in the SOAR system. However, few attempts have been made to

create problem spaces from interactions with an environment, although Drescher [11] did some interesting work on implementing early Piagetian learning.

As for discovering hidden features, the BACON system [25] creates new terms to infer laws from given data. Mitchell and Utgoff [32, **?**] present methods for detecting the insufficiency of a concept description language in problem solving and define new terms. Dietterich and Michalski [10] give a survey on some of the existing *constructive induction* systems for discovering hidden features in time-independent environments, and present their solutions.

Finally, learning from the environment as defined here is closely related to the theoretical problem called system identification or inductive inference (see [4] for an excellent survey) where the environment is viewed as a black box from which inputs and outputs can be observed. Gold [16] has proved the complexity for identifying languages in limit. Angluin develops an algorithm [2] that learns regular sets by asking an oracle for counterexamples. Valiant [57] has proposed a new criteria for what is learnable and presented his algorithms. Recently, Rivest and Schapire have developed a new representation, called *diversity*, for finite automata and an algorithm that learns from some deterministic environments very efficiently. In fact, the algorithms presented in this thesis are developed from an earlier paper by the author [42] on learning from finite deterministic environments by experiments.

Chapter 4

The LIVE System

LIVE is a system that implements the theory of learning from environments. In this chapter, we will survey the system's architecture, and describe in detail the system's internal knowledge representation and one of its subsystems, the problem solver. Other components of the system will be discussed in the chapters that follow.

4.1 System Architecture

The LIVE system is an extension of the GPS problem solving framework to integrate a learning component that creates and learns an internal model from the environment. The entire system can be divided into six components: the internal model (the knowledge base), the problem solver (including planning), the exploration module, the rule creation module, the rule revision module, and the experiment module. Their relationship is illustrated in figure 4.1. In order to work in a particular environment, as we defined in Chapter 2, LIVE is also given a set of goals, actions, percepts, and constructors.

The architecture works under the control of the algorithm outlined in Figure 4.2. The main idea is to create new knowledge by exploring the environment and to correct any faulty knowledge by analyzing errors which arise in problem solving. When the goals are given, LIVE will immediately attempt to construct a solution plan (by Means-Ends Analysis) using the knowledge in the current internal model (which is empty at the outset). If no knowledge can be found for constructing a solution, LIVE will call its exploration module to generate an *exploration plan* (a sequence of actions) which will be executed to gather new knowledge from the environment. (This knowledge, as we will describe shortly, is represented as rules with

Figure 4.1: The LIVE architecture.

Figure 4.2: The outline of LIVE's algorithm.

conditions, actions, and predictions, and is designed for correlating actions and percepts.) If a solution plan is constructed successfully, then LIVE will execute it with learning. During plan execution, subgoals will be proposed when the condition for a solution step is not satisfied. Since the correctness of a solution plan cannot be guaranteed, plan execution is monitored to detect errors. When a prediction in the solution dose not match the real

1. Constructing a solution plan: Find the differences between the goals (or an *experiment*) and the current state; Find a rule for each difference; Order the differences by their rules; If *goal contradiction* is detected, design an *experiment* and go to 1;
2. Exploring the environment: If the first difference in the solution plan has no rule, then Call the Exploration module to propose an *explorative plan*; While the explorative plan is not empty, do: Execute an explorative action; If the action has no prediction, then call the Rule Creation module to create a new rule; If the action violates its prediction, then call the Rule Revision module to revise the faulty rule; Go to 1;
3. Proposing Subgoals: Select the first difference and its rule from the solution plan; If the rule is not executable in the current state, then Identify new differences; Select a rule for each new difference; Order the differences by their rules; If *subgoal loop* is detected, then design an *Experiment* and go to 1, else push the differences into the solution plan and go to 3;
4. Execution with Learning: Make a prediction from the rule and execute its action; If all the given goals are accomplished, then exit; If the prediction is fulfilled by the action's outcome then go to 3; else (a *surprise*) Call the Rule Revision module to revise the faulty rule; Go to 1;

outcome perceived from the environment, LIVE will call the rule revision module to revise the faulty rule. After the faulty rule is corrected, LIVE will construct a new solution plan and then continue its problem solving until all the given goals are solved.

Experiments (the precise definition of an experiment and its usage will be given in Chapter 6) will be designed in two cases. One case is when the current knowledge causes a *Goal Contradiction* error (goals destroy each other) at planning time. The other case is when a *Subgoal Loop* error (the same goal is proposed repeatly) is detected at the subgoal proposing time. Since all experiments are carried out as if solving some new problems in the environment, the mechanism for problem solving and learning is also used to execute and learn from experiments.

4.2 The Internal Model: Knowledge Representation

As we can see from the architecture, the internal model is LIVE's knowledge base. It reflects how much the system knows about the environment and provides the backbone for all the other components in the system. For instance, the problem solver uses it to plan solutions for achieving the goals, the exploration module uses it to propose explorative plans, the experiment module uses it to design experiments, and the rule revision module accesses it

to detect knowledge errors and rectify them. Therefore, the representation for the internal model must be competent for multi purposes, including planning, execution, error detection and knowledge modification.

Given these constraints, LIVE's knowledge representation is chosen to be a set of C-A-P production rules with three components: *conditions*, *actions*, and *predictions*. The actions in C-A-P rules are those given to the learner at the outset. The conditions and predictions of C-A-P rules are well-formed formulas in LIVE's description language L (see section 4.3). The action contained in a rule can be executed whenever the condition of that rule is satisfied, and the prediction of the rule predicts the effects of the action. The C-A-P rules are very much like STRIPS operators but have one important difference: The postconditions of STRIPS operators modify the perceived world by deleting and adding elements, while the predictions of C-A-P rules serve only as templates for checking and matching the perceived world. The C-A-P rules are also like the operators used in Carbonell and Gil's learning from experimentation [9]. However, LIVE represents the inference rules and the operators uniformly and appears more natural. For example, since only ALUMINIZE(obj), among others actions in their program, can possibly be the last action for making a mirror, their Operator-3 and Inference-Rule-1 can be combined into one C-A-P rule in our representation.

As examples, let us take a close look at some of the C-A-P rules learned by LIVE in the RPMA environment (see Chapter 2 for the definition of the environment). In these rules (and for all the rules in this thesis), the constants and predicates are shown in upper case, and variables are in italics. Elements in a list are conjuncts. Negations are represented by the word NOT. Variables are quantified as described in Section 4.3.

Condition: ((**IN-HAND** *x*) (**NOT** ((**ON** *y p*)))) **Action:** **Move-Put**(*x p*) [**Rule-0**]
Prediction: ((**ON** *x p*) (**NOT** ((**IN-HAND** *x*))))

Condition: ((**IN-HAND** *x*) (**ON** *y p*)) **Action:** **Move-Put**(*x p*) [**Rule-1**]
Prediction: ((**IN-HAND** *x*))

Rule-0 says that if there is a disk x in the hand *and* there is no disk y on the peg p, then the action Move-Put(x p) will leave disk x on peg p and disk x will not be in the hand any more. Similarly, Rule-1 says that if disk x is in hand and there is a disk y on peg p, the action will have no effect on disk x, i.e. disk x will be still in the hand.

In addition to relational predicates like ON and IN-HAND, the conditions and the predictions of C-A-P rules can also contain the feature vectors of objects. For example, (OBJECT-345 DISK RED 10CM) represents a red disk with a diameter of 10cm. Sometimes, certain features of an object are unimportant to a rule, and those features will be represented as a "don't care" symbol, #, which can be matched to any value. If object feature vectors are also included, then Rule-0 will look like the following:

Condition: ((**IN-HAND** x) (**NOT** ((**On** y p))) (x **DISK RED 10**) (y **DISK # #**) (p **PEG # #**)) **Action:** **Move-Put**(x p) [**Rule-0'**] **Prediction:** ((**ON** x p) (**NOT** ((**IN-HAND** x)) (x **DISK RED 10**) (y **DISK # #**) (p **PEG # #**))

The C-A-P rules have some interesting and useful characteristics. Like productions, C-A-P rules have a simple and uniform structure. This makes learning easier than modifying programs (as demonstrated in many adaptive production systems [59]). In particular, since C-A-P rules have predictions, it is straightforward to detect incorrect rules. Every time after a rule's action is applied, the actual outcome will be compared to the rule's prediction, and if the comparison results in some differences then the rule is wrong and must be revised. For example, if Rule-1 is applied when disk x is smaller than disk y, then the prediction (In-Hand x) will be wrong because disk x will be put on the peg p.

The C-A-P rules can also be used as problem reduction rules when applied backwards. Given some goals to reach, the actions of the rule whose predictions match the goals become the means to achieve the goals, and the conditions of that rule, if not true already, are the subgoals that must be achieved beforehand. For example, in order to put a disk x on a peg p, Rule-0 tells us that action Move-Put must be performed and (IN-HAND x) and (NOT ((ON y p))) must be satisfied before the action can apply.

Finally, and most interestingly, C-A-P rules have sibling rules. This is because whenever a rule is found to be incorrect, LIVE's learning method will split the rule into two rules with a discriminating condition. (see Chapter 6.) Sibling rules share the same action, but they have some opposite conditions and different predictions. For instance, Rule-0 and Rule-1 are sibling rules; They have the same action (Move-Put x p), but their predictions are different and their conditions differ at (ON y p). As we will see later, sibling rules plays an important role for designing experiments.

4.3 LIVE's Description Language

As we pointed out in the last section, the conditions and predictions of C-A-P rules are expressions in LIVE's description language. In fact, the same language also describes the state of the environment (a state is perceived as a set of ground atomic formulas in this language). In this section, we will describe this language's syntax and semantics, and give details of how a rule's conditions and predictions are matched to the states perceived from the environment.

4.3.1 The Syntax

The description language L has the following alphabet of symbols once the percepts and constructors are given:

- Variables, $x_1, x_2, ...$, (including the "don't care" symbol #),
- Individual constants $a_1, a_2, ...$, such as DISK, RED, and SPHERE,
- Predicates A_i^n, such as ON, GREATER, $=$, $>$,
- Identifiers of objects O_i, such as OBJECT-345,
- Functions f_i^n, such as $+$, $\times$,
- The punctuation symbols (and),
- The logical connectives $\wedge$ (represented as lists) and $\neg$ (represented as NOT),
- The existential quantifier $\exists$ (occurs wherever there are free variables).

Note that for different environments, L may have different alphabets because the percepts might be given differently. Also, the number of relational predicates is not fixed because new ones can be defined by LIVE during the learning process. For example, if the relational predicate "=" is given as a constructor, then new predicates such as DISTANCE= and WEIGHT= might be defined. Although the alphabet does not include the logical connective OR ($\vee$), LIVE is capable of learning disjunctive conditions and predictions because a disjunct like $P_1 \vee P_2$ can be learned as $\neg(\neg P_1 \wedge \neg P_2)$.

To define what are the well-formed formulas in L, we first define a *term* in L as follows: (Terms are those expressions in L that will be interpreted as objects or features of objects.)

(i) Variables and individual constants are terms.

(ii) If f_i^n is a function, and $t_1, \ldots, t_n$ are terms in L, then $f_i^n(t_1, \ldots, t_n)$ is a term in L.

(iii) The set of all terms is generated as in (i) and (ii).

A *atomic formula* in L is defined as follows: if X is a predicate or an identifier in L and $t_1, \ldots, t_n$ are terms in L, then $(X\ t_1 \ldots t_n)$ is an atomic formula of L. An atomic formula is *ground* if all $t_1, \ldots, t_n$ are constants. Atomic formulas are the simplest expressions in L. They are to be interpreted as relationships among objects or as individual objects with features. For example (ON x y) means object x is on y, (OBJ3 DISK RED) means the object OBJ3 is a red disk.

Based on atomic formulas, the *well-formed formulas* of L are defined as follows:

(i) Every atomic formula of L is a wf of L.

(ii) If A and B are wfs of L, so are $(\neg A)$, $(A \wedge B)$ and $(\exists_{x_i})(A)$ where x_i is a variable.

(iii) All the wfs of L are generated as (i) and (ii).

4.3.2 The Interpretation

The expressions of L are interpreted on the objects and their relations perceived from the external world by the learner. For instance, the predicate symbols A_i^k are interpreted as relations among objects; the object identifiers O_i are the names assigned to the external objects; and the function symbols f_i^n are interpreated as functions. Variables in L, if they appear in a relational atomic formula, are interpreted as objects. If they appear in an object's feature vector, they are interpreted as the object's features.

The following defines what it means for an expression in L to be true, or matched, in a state perceived from the external world:

- An atomic formula is true iff there is a ground relation or an object feature list in the state that unifies with the formula.
- A conjunctive expression is true iff all of its conjuncts match the state.
- The negation of an expression is true iff the expression does not match the state.
- An exsitentially quantified expression is true iff there exists an assignment of values to each of the variables such that the substituted expression is true in the state. For example, $\exists_x$((ON x PEG1) (GREATER x DISK1)) is true iff there is a disk on PEG1 and its size is greater than DISK1.

In the rest of the thesis, since all variables will be interpreated as exsitentially quantified, we will omit the symbol $\exists$ in future expressions, but use the following convention (note that

this may restrict the set of logical relations expressible):

$$
\begin{aligned}
P(x) &\implies \exists_x P(x) \\
\neg P(x) &\implies \neg\exists_x P(x) \\
\neg(P(x) \wedge Q(x)) &\implies \neg\exists_x (P(x) \wedge Q(x)) \\
\neg(P(x) \wedge \neg Q(x)) &\implies \exists_x (\neg P(x) \vee Q(x)) \\
\neg(\neg P(x) \wedge Q(x)) &\implies \exists_x (P(x) \vee \neg Q(x)) \\
\neg(\neg P(x) \wedge \neg Q(x)) &\implies \exists_x (P(x) \vee Q(x)) \\
(P(x) \wedge Q(x\ y)) &\implies \exists_x (P(x) \wedge (Q(x\ y)))
\end{aligned}
$$

Therefore, the following expression: is equivelant to

((IN-HAND x) ($\neg$ ((ON y PEG3) (GREATER x y))) (x DISK *color size*) (PEG3 PEG # #))

$\exists_x \exists_{color} \exists_{size}$((IN-HAND x) ($\neg$ $\exists_y$((ON y PEG3) (GREATER x y))) (x DISK *color size*) (PEG3 PEG # #))

4.3.3 Matching an Expression to a State

In order to understand LIVE's description language, it is necessary to know how the system matches conditions and predictions to a state that perceived from the external environment.

When an expression is given, LIVE matches all the positive relational predicates to the state before matching any negative relations, and it matches all the negative relations before matching object feature lists. Matching is done from left to right in each of these three submatching processes. Every time a subexpression is matched successfully, the bindings from that match are immediately used to substitute all occurences of the bound variables in the whole expression. If all the subexpressions are true, the match succeeds. If any of the subexpression is false, the matching process will backtrack, and continue searching for a new match in a depth-first fashion. The final result of a match, if successful, is the set of bindings established in the process.

To illustrate the matching process, let us look at an example of how the matcher works on the following expression and state:

Expression	State
((IN-HAND *x*) (NOT ((ON *y* PEG3) (GREATER *x* *y*))) (*x* DISK *color size*) (PEG3 PEG # #))	(ON DISK3 PEG2) (ON DISK2 PEG3) (GREATER DISK2 DISK1) (GREATER DISK3 DISK1) (GREATER DISK3 DISK2) (IN-HAND DISK1) (PEG1 PEG BLUE 5) (PEG2 PEG BLUE 5) (PEG3 PEG BLUE 5) (DISK1 DISK RED 1) (DISK2 DISK RED 2) (DISK3 DISK RED 3)

The first positive relational predicate (IN-HAND *x*) in the expression matches (IN-HAND DISK1) because there is only one IN-HAND relation in the state. Thus *x* is bound to DISK1, and all the *x* in the expression are substituted by DISK1. Since (IN-HAND *x*) is the only positive relation in the expression, the negative relation (NOT ((ON *y* PEG3) (GREATER DISK1 *y*))) is now in consideration. (Note that *x* in this negative expression has been replaced by DISK1.) There is a free variable *y* in this negative expression, so we have to make sure all the possible values are tried. First, *y* is bound to DISK3 because the state contains (ON DISK3 PEG3) which matches (ON *y* PEG3). The matcher cannot find (GREATER DISK1 DISK3) so this binding fails. Next, *y* is bound to DISK2 since there is a relation (ON DISK2 PEG3) in the state, but this binding also fails because there is no (GREATER DISK1 DISK2). After these two failures, the negative expression returns true (because the matcher cannot find any other value in the state that can be bound to *y*) and the match continues. The last two subexpressions are true in the state, *x* is already bound to DISK1, so (*x* DISK *color size*) will be matched to (DISK1 DISK RED 1), and (PEG3 PEG # #) will be matched to (PEG3 PEG BLUE 5). Thus the expression is true in the state and the matcher returns the set of bindings that have been made: ((*x* . DISK1) (*color* . RED) (*size* . 1)).

To see how the matcher backtracks, suppose the state also contains the following facts: (IN-HAND DISK4), (GREATER DISK4 DISK3), (GREATER DISK4 DISK2), (GREATER DISK4 DISK1), (DISK4 DISK RED 4). In other words, the state contains a fourth and largest disk, and this disk is in the hand. Suppose LIVE first matches (IN-HAND *x*) to (IN-HAND DISK4) instead of (IN-HAND DISK1). Then the match will eventually fail because there exist DISK2 and DISK3 that are on PEG3 and they are smaller than DISK4. After the failure, the matcher will look for the next possible value for *x* and will match (IN-HAND

x) to (IN-HAND DISK1). Then the match will succeed as before.

4.4 The Problem Solver Module

We now turn our attention to the problem solver module (including planning).

4.4.1 The Problem Solver's Functions

LIVE's problem solver undertakes three tasks: to plan solutions for the goals that are given, to execute solutions in the environment, and to detect errors both in planning and in execution.

The problem solver plans solutions using means-ends analysis. It first finds the differences between the goals and the current state, then for each difference it selects, from the current internal model, a rule whose prediction indicates that the difference will be eliminated. If more than one such rule is found, a set of heuristics will be used to select the "best" one. After the differences are associated with rules, a plan is constructed by ordering these differences according to the conditions of their rules. The ordering criteria is similar to the methods used in HACKER [54] and the *goal regression* in [**?**]. The rules whose conditions interfere with the other differences in the plan will be executed later, while the rules whose conditions have no or little interference with the other differences will be executed earlier.

Once a plan is constructed, the problem solver will execute the actions in the plan in the order. The execution is monitored in the sense that whenever an action is taken, the actual outcome in the environment will be compared to the prediction that made by the rule that contains the action. If all the predictions in the plan are fulfilled and all the given goals are achieved, the problem solver will terminate.

Error detection occurs at both planning time and execution time. When planning, two kinds of errors are possible. The *No Rule* error happens when there is no rule to achieve the given goals. In this case, the exploration module will be called to gather more information from the environment. The *Goal Contradiction* error occurs when the goals always conflict with each other (according to some rules) no matter how they are ordered. In this case, the experiment module will be called to find out how to revise these rules that indicate such contradictions. Errors are also possible at execution time; a rule's prediction may fail to match the actual outcome of its action. In this case, the error is called a *surprise*, and the rule revision module will be called to analyze the error and rectify the faulty rule.

The most important feature of LIVE's problem solver is it is interleaved with learning. Here, one cannot assume that the results of actions will always be what is expected. The execution of a planned solution is not the end of the problem solving, it is an experiment for testing the correctness of the knowledge that has been learned so far. In LIVE, the cooperation of learning and problem solving is made possible by associating actions with predictions, which is one of the important advantages of the C-A-P rule representation.

4.4.2 An Example of Problem Solving

To illustrate the problem solver, let us go through a concrete and detailed example. Suppose LIVE is learning from the RPMA environment and currently in the external state shown as the picture in Figure 4.3. Suppose also it is given the following goals to achieve (note that goals are also written in the language L described in the section 4.3.):

GOALS: ((ON DISK1 PEG3) (ON DISK2 PEG3) (ON DISK3 PEG3)).

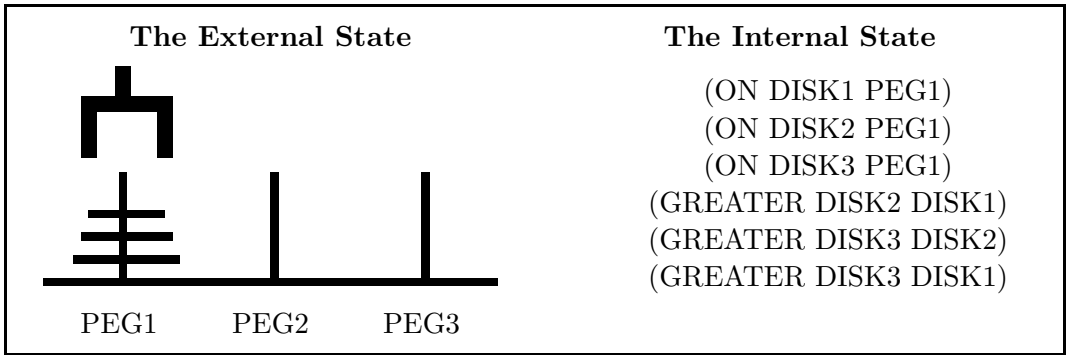


Figure 4.3: The initial state of a problem.

LIVE first perceives the environment and constructs the internal state shown in Figure 4.3. Then the problem solver compares the state with the given goals and finds the differences: (ON DISK1 PEG3), (ON DISK2 PEG3), and (ON DISK3 PEG3). Since the internal model is empty at the beginning, the problem solver fails to find any rule to eliminate these differences, hence a *No Rule* error is detected and the exploration module is called. After exploring the environment (the details will be discussed in chapter 5), two rules are created:

Condition: ((ON *disk peg*)) **Action: Move-Pick(***disk peg***)** [**Rule0**] **Prediction: ((IN-HAND** *disk*) **(NOT ((ON** *disk peg*))))

Condition: ((IN-HAND *disk*)) **Action: Move-Put(***disk peg***)** [**Rule1**] **Prediction: ((ON** *disk peg*) **(NOT ((IN-HAND** *disk*))))

Rule0 is a general rule about the Move-Pick action; it says that the action can pick up a disk from a peg if the disk is on that peg. Rule1 is a rule about the Move-Put action; it says that the action can move to a peg and put a disk on that peg if the disk is in the hand. With these two rules, the problem solver can now plan its actions for the goals because Rule1 has a prediction, (ON *disk peg*), that can be matched to each of the given goals. A new plan is then constructed as follows:

Goal	Rule	Prediction	Conditions
(ON DISK1 PEG3)	Rule1	(ON *disk peg*) with bindings ((*disk* . DISK1) (*peg* . PEG3))	(IN-HAND *disk*)
(ON DISK2 PEG3)	Rule1	(ON *disk peg*) with bindings ((*disk* . DISK2) (*peg* . PEG3))	(IN-HAND *disk*)
(ON DISK3 PEG3)	Rule1	(ON *disk peg*) with bindings ((*disk* . DISK3) (*peg* . PEG3))	(IN-HAND *disk*)

This plan is constructed by associating with each of the goals a rule that predicts achieving that goal. For example, the goal (ON DISK1 PEG3) is associated with Rule1 because Rule1's prediction (ON *disk peg*) can reach the goal. This association is done by searching all the rules in the knowledge base and finding the rules whose prediction matches the goal. Sometimes, more than one rule will be found. In this case, a set of heuristics (see section 4.4.3) is used to decide which one to use. After all the goals are associated with rules, the problem solver reorder the goals according to their rule's conditions. In our example, since none of the conditions (IN-HAND *disk*) interferes with any of the goals (ON DISK1 PEG3), (ON DISK2 PEG3), or (ON DISK3 PEG3), the reordering does not change the order of the goals.

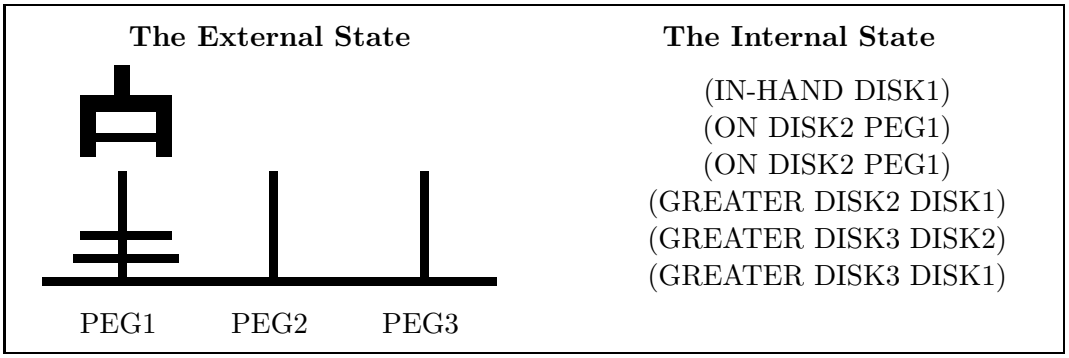
Given this plan, LIVE now starts executing the rules one by one in the means-ends analysis fashion. Working on the first goal (ON DISK1 PEG3), the problem solver realizes that Rule1 has a condition, (IN-HAND *disk*), which must be satisfied in order to apply its action. Here the solver proposes a subgoal (IN-HAND DISK1) by substituting the variable

using the bindings of the rule. (If the subgoals contains free variables that are not bound by the bindings of the rule, then a set of heuristics, see section 4.4.3, will be used to bind them to concrete objects.)

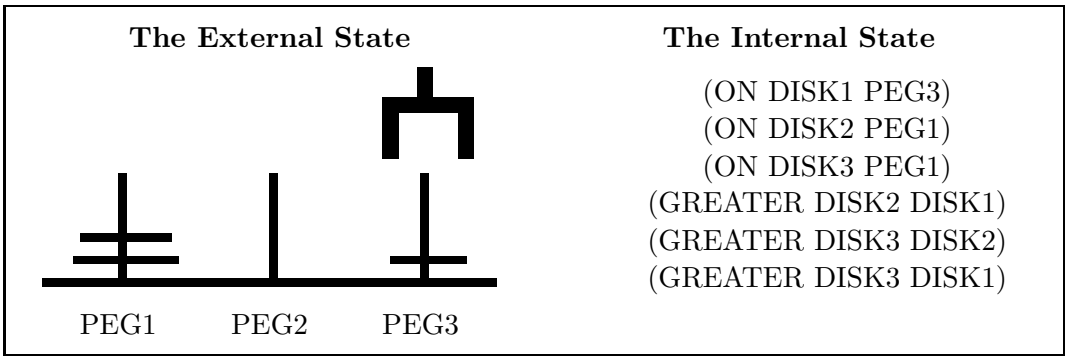
As before, to reach the new goal (IN-HAND DISK1), LIVE searches all the rules in the knowledge base and finds that Rule0 predicts the goal. So Rule0 is associated with (IN-HAND DISK1), and a new entry is pushed on the top of the plan as shown in the following table:

GOAL	RULE	PREDICTION	CONDITIONS
(IN-HAND DISK1)	Rule0	(IN-HAND *disk*) with bindings (*disk* . DISK1)	(ON DISK1 *peg*)
(ON DISK1 PEG3)	Rule1	...	...
(ON DISK2 PEG3)	Rule1	...	...
(ON DISK3 PEG3)	Rule1	...	...

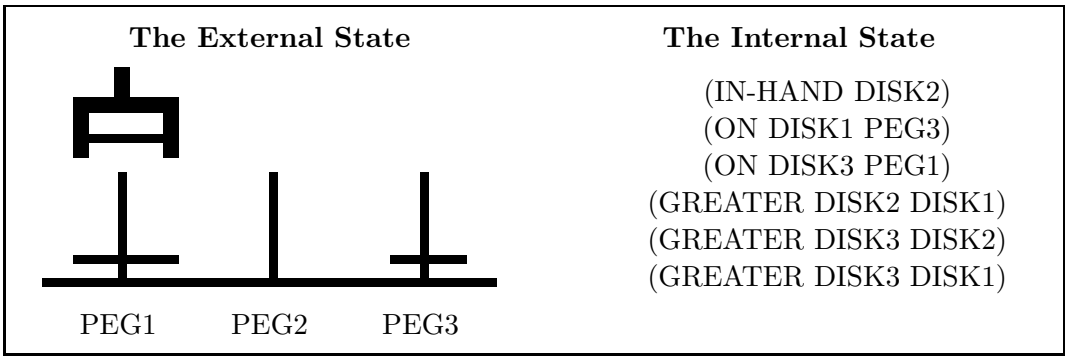
LIVE continues to execute the plan. This time Rule0's action Move-Pick(DISK1 *peg*) can be executed because Rule0's condition (ON DISK1 *peg*) is satisfied in the current state. (ON DISK1 *peg*) is matched to (ON DISK1 PEG1) and *peg* is bound to PEG1. LIVE then executes Move-Pick(DISK1 PEG1) and predicts that (IN-HAND DISK1). After the action is taken, LIVE senses the external world, and builds a new internal state as follows:



LIVE compares the prediction (IN-HAND DISK1) with the state and finds it is fulfilled, so the plan execution continues. Now, Rule1's action Move-Put(DISK1 PEG3) becomes executable because the condition (IN-HAND DISK1) is true in the current state. The action Move-Put(DISK1 PEG3) is taken, and LIVE senses the external world:



Again, the prediction of Rule1 (ON DISK1 PEG3) is fulfilled and LIVE is now working on the next goal in the plan: (ON DISK2 PEG3). As before, (IN-HAND DISK2) is proposed as a subgoal and Rule0 is chosen to achieve it. After the action Move-Pick(DISK2 PEG1) is executed successfully, the state is now as follows:



Since Rule1's condition (IN-HAND DISK2) is satisfied now, LIVE executes the action Move-Put(DISK2 PEG3) and expect (ON DISK2 PEG3). After sensing the external world, it finds that the prediction is false: DISK2 is not on PEG3 but still in the hand (because DISK1 is on PEG3 and DISK1 is smaller than DISK2).

This is a *surprise*. At this point, LIVE suspends its plan and calls the rule revision module which will analyze the error and split the faulty Rule1 into the following two new rules (details are in Chapter 6):

Condition: ((IN-HAND *disk*) **(NOT ((ON** *y peg*)))) **Action: Move-Put(***disk peg***)** [**Rule1**] **Prediction: ((ON** *disk peg*) **(NOT ((IN-HAND** *disk*))))

The new Rule1 says that in order to successfully put a disk on a peg, the peg must be empty (no y is on the peg), otherwise, as Rule2 predicts, the disk will be still in the hand.

Condition: ((IN-HAND *disk*) **(ON** *y peg*)) **Action: Move-Put(***disk peg***)** [**Rule2**] **Prediction: ((IN-HAND** *disk*))

With these new rules, a new plan is constructed as follows (note that (ON DISK1 PEG3) is already satisfied in the current state, so it does not appear in the new plan):

GOAL	RULE	PREDICTION	CONDITIONS
(ON DISK2 PEG3)	Rule1	(ON *disk peg*) with bindings ((*disk* . DISK2) (*peg* . PEG3))	(IN-HAND *disk*) (NOT ((ON *y peg*))
(ON DISK3 PEG3)	Rule1	(ON *disk peg*) with bindings ((*disk* . DISK3) (*peg* . PEG3))	(IN-HAND *disk*) (NOT ((ON *y peg*))

Unfortunately, when reordering these two goals, LIVE finds that no matter which goal is to be achieved first, it will always be destroyed by the later goals. For example, if (ON DISK2 PEG3) is achieved first, it will be destroyed by the condition of Rule1, (NOT ((ON *y peg*))), for achieving the goal (ON DISK3 PEG3). Here, a *Goal Contradiction* error is found, and Rule1 is the troublesome rule. The problem solver will stop its planning and call the experiment module to gather information for revising Rule1 (the details are in Chapter 6).

Up to this point, we have illustrated all the basic activities of LIVE's problem solver. This kind of alternation of planning, execution and learning is repeated until finally a set of correct rules are learned. To complete this example, one of final rules is the following: and LIVE's final plan is the following:

Condition: ((IN-HAND *disk*) **(NOT ((ON** *y peg*) **(GREATER** *disk y*)))) **Action: Move-Put(***disk peg***)** [**Rule3**] **Prediction: ((ON** *disk peg*) **(NOT ((IN-HAND** *disk*)))

GOAL	RULE	PREDICTION	CONDITIONS
(ON DISK3 PEG3)	Rule1	(ON *disk peg*) with bindings ((*disk* . DISK3) (*peg* . PEG3))	(IN-HAND *disk*) (NOT ((ON *y peg*) (GREATER *disk y*))
(ON DISK2 PEG3)	Rule1	(ON *disk peg*) with bindings ((*disk* . DISK2) (*peg* . PEG3))	(IN-HAND *disk*) (NOT ((ON *y peg*)) (GREATER *disk y*))

Note that this time DISK3 is forced to be put on PEG3 before DISK2 because if DISK2 were put on PEG3 first, (ON DISK2 PEG3) will violate the condition (NOT ((ON *y* PEG3) (GREATER DISK3 *y*))) for putting DISK3 on PEG3. With this new plan, LIVE first moves DISK1 away from PEG3 then successfully puts all the disks on PEG3 and solves all the given goals.

4.4.3 Some Built-in Search Control Knowledge

At the present, LIVE connot learn all the necessary search control knowledge [5, 29]. For example, when LIVE wants to put down a disk to empty its hand, it cannot learn that the "other" peg is a good place to put the disk down. (The "other" peg concept is a part of the sophisticated perceptual strategy outlined in [49].)

Thus, two sets of domain-dependent heuristics are imployed by the problem solver to guide some of the decision making in planning and subgoal proposing. Although LIVE will still learn the correct rules from the environment without these heuristics, problem solving will take much longer time because whenever a strategic decision is necessary the current system will make a random choice.

The first set of heuristics are for binding free variables in a rule when the rule is selected to propose subgoals. In the RPMA environment, these heuristics implement exactly the "other" peg concept. For example, in order to move DISK3 from PEG1 to PEG3 when DISK2 is currently in the hand, LIVE must first achieve the subgoal (NOT ((IN-HAND DISK2))). Rule3 will be selected because its prediction contains (NOT ((IN-HAND *disk*))) (so *disk* is bound to DISK2). However, when applying Rule3's action Move-Put(*disk peg*), LIVE will notice that *peg* is still a free variable. In this case, the heuristics will bind *peg* to PEG2 because PEG2 is the "other" peg when DISK3 is to be moved from PEG1 to PEG3.

The second set of heuristics are for selecting which rule to use when more than one rule is found to predict the same goal. Although no such situations have been encountered in the RPMA environment, the heuristics are necessary in the Hand-Eye version of Tower of Hanoi (see Chapter 2 for the definition) because the number of rules is larger. For example, in order to pick up DISK2 when DISK1 is on the top, LIVE will generate a subgoal to move DISK1 away. This goal can be accomplished by picking up DISK1 and turning the hand away from DISK2. However, to turn the hand back to DISK2, two rules can be used. One rule turns the hand back when the hand is empty, and the other turns the hand back when

the hand is full. Although the later can applied immediately (because DISK1 is in the hand), it is the former rule that is needed. At present, the heuristics for the Hand-Eye environment are designed to help LIVE to choose the right rule in this kind of situations.

4.5 Summary

This chapter has introduced the system LIVE as a whole. Although we have described in detail the internal knowledge representation and the problem solver, the architecture and the overall interactions between different modules are more important for understanding the system. In the following chapters, when other individual modules are presented, the reader is encouraged to refer to LIVE's architecture (Figure 4.1) and its algorithm (Figure 4.2).

Chapter 5

Rule Creation through Explorations

In the last chapter, we outlined LIVE's architecture and introduced its internal knowledge representation. In particular, we described in some detail the functions of the Problem Solver and indicated that it will invoke other modules if planning or execution have confronted exceptions. In this chapter, we will have a closer look at LIVE's Rule Creation module and Exploration module, and explain how the rules like Rule0 and Rule1 in section 4.4 were created and how LIVE explores its environment.

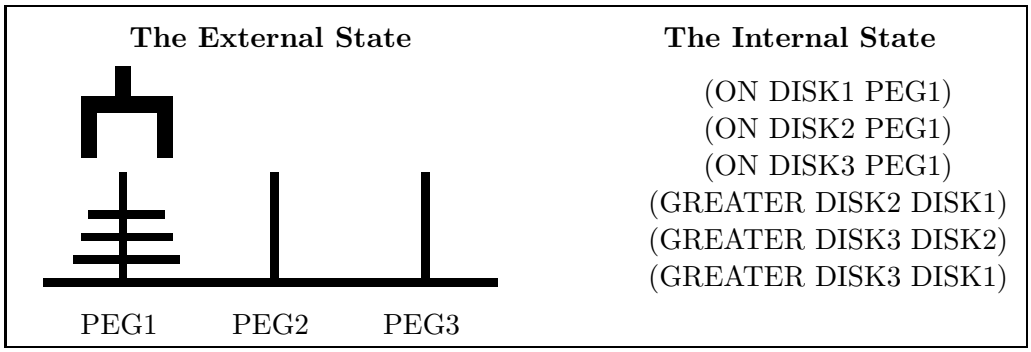
5.1 How LIVE Creates Rules from Objects' Relations

In a new environment, since LIVE does not know the consequences of its actions at the outset, it has no idea how to achieve the given goals at the very beginning. For example, if the learner never moved any disk before, it will not know which actions to use to move the disks. Thus, when the current knowledge is insufficient to construct any plan for certain goals, it is necessary for the learner to explore the environment and create new rules to correlate its actions and its percepts. This is the main task of the Rule Creation module. In this section, we will focus our attention on creating new rules from objects' relations, while leave the more complex case, creating new rules from objects' features, to the next section.

To illustrate the idea, let us first consider an explorative scenario in the RPMA environment. Suppose that LIVE has decided to explore the action Move-Pick(DISK1, PEG1) in the state shown in Figure 5.1, and observed the outcome from the environment after the

action is taken. What rule should LIVE create from this scenario?

Before the Action:



The Action: Move-Pick(DISK1, PEG1)
After the Action:

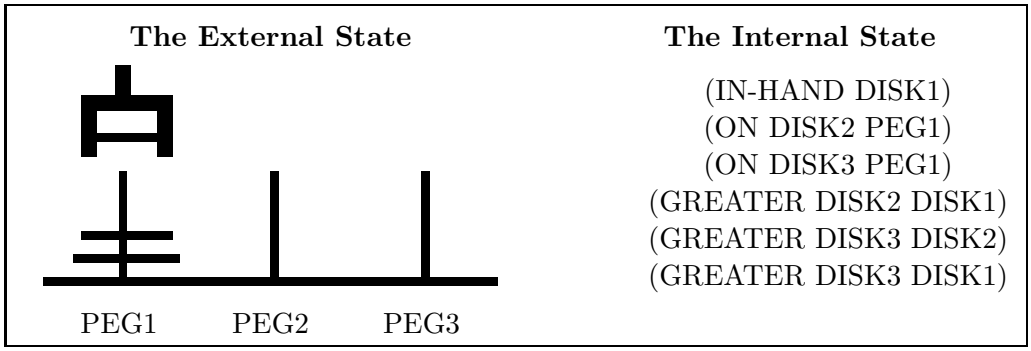


Figure 5.1: Creating new rules from the relations that have been changed.

Since our intention here is to create a new C-A-P rule and the action part is already known as Move-Pick(DISK1 PEG1), what we need is to construct the conditions and the predictions under the following constraints. First, since we cannot take the whole state as conditions or predictions (the rule would be useless), we must decide which relations from the initial state should be used as conditions, and which from the final state should be used as predictions. Next, since the new rule is to correlate percepts and actions, the conditions and prediction must be related to the action. Finally, since the rule should be useful in the future to predict similar events, it must be generalized from this particular scenario.

To decide what must be included in the conditions and in the predictions, LIVE's strategy is to focus on changes in the relations. In this particular scenario, LIVE will select the relation (ON DISK1 PEG1) from the initial state, and select (IN-HAND DISK1) from the final state

because the former was true in the initial state but not in the final state, while the later emerges in the final state but was not true in the initial state. Thus the condition will be (ON DISK1 PEG1) and (NOT ((IN-HAND DISK1))); the prediction will be (IN-HAND DISK1) and (NOT ((ON DISK1 PEG1))). Note that the negations of the relations are also included in the condition and the prediction. These are to reflect the changes of the relations in the rule and to make the rule more useful in problem solving. Since both the conditions and the predictions already contain the elements mentioned in the action, namely, DISK1 and PEG1, the second constraint is satisfied automatically. Finally, to generalize the new rule, LIVE will replace DISK1 with a variable *disk*, and PEG1 with a variable *peg*, so that they can be matched to any disk or peg in the future. Thus, a new rule is created as follows:

Condition: ((**ON** *disk peg*) (**NOT** ((**IN-HAND** *disk*)))) **Action:** **Move-Pick**(*disk peg*) **Prediction:** ((**IN-HAND** *disk*) (**NOT** ((**ON** *disk peg*))))

From this simple example, we can see that LIVE creates new rules in three steps:

1. Compare the states before and after the action and find a set of *vanished* relations and a set of *emerged* relations.

2. Relate the condition and prediction with the action that is taken.

3. Create a C-A-P rule as follows:

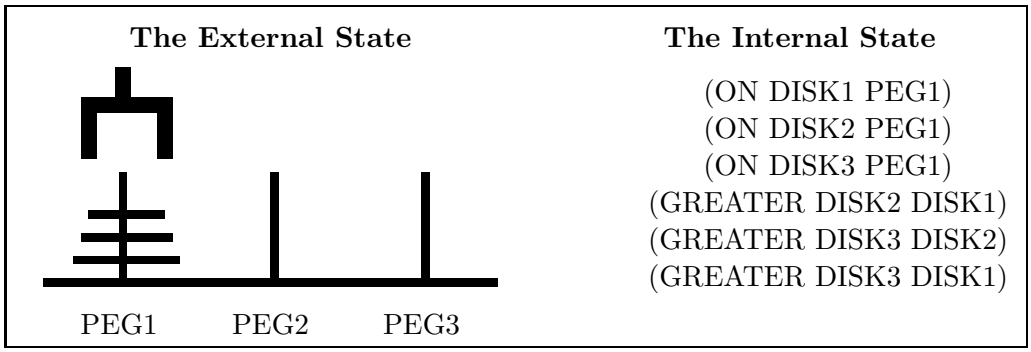
   Condition: the vanished relations plus the negations of each individual emerged relation;
   Action: the action just taken;
   Prediction: the emerged relations plus the negations of each individual vanished relation;

   and then generalize all constants in the new rule to variables.

One important criterion is the usefulness of the newly created rule, that is, whether it can be applied to many similar cases to predict the outcomes of LIVE's actions. To ensure this, the method we just outlined emphasizes that a new rule should be as general as possible. For example, if the set of percepts are more realistic, then there are many relations, such as shadow locations and hand atitude, that could be changed by the action.

In that case, we will ignore these relations in the new rule. If any of them is proved to be relevant in the future, it will be brought into the rule by our learning algorithm (see Chapter 6). In the process of creating rules, we prefer to keep the new rule overly general rather than overly special. Thus, we will not only change the constants to variables, but also put as few elements as possible in the condition. In the example above, the rule is indeed most general because the action happens to make one and only one relation change. Now the question is how do we choose conditions and predictions when an explorative action does not change any relations, or changes many relations?

Before the Action:



The Explorative Action: Move-Put(DISK1, PEG2)
After the Action:

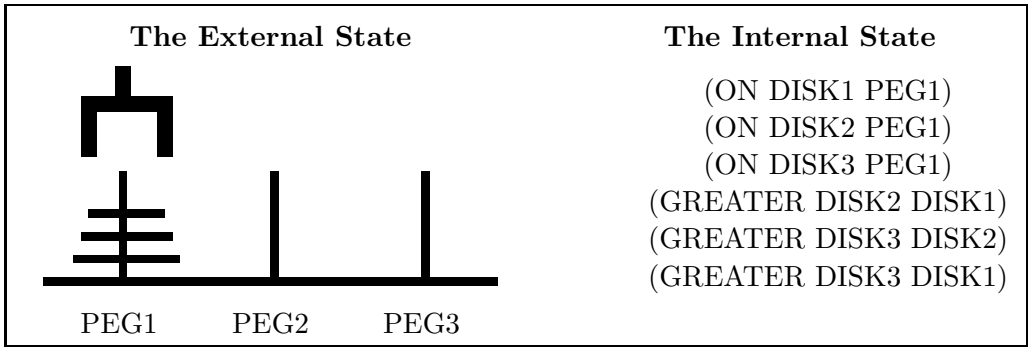


Figure 5.2: Explorative action causes no relation change.

For the case of no relation change, consider the scenario in Figure 5.2, where LIVE explores the action Move-Put(DISK1, PEG2) instead of Move-Pick(DISK1, PEG1). Since there is nothing in LIVE's hand to be put down, nothing will be changed by Move-Put(DISK1, PEG2) and comparing states will return no relation changes at all. In this case, a new rule

will still be created except that its condition and prediction will be the same. To make the new rule as general as possible, LIVE will only select those relations that are *relevant*: those share common relation type with the given goals and those share common objects with the taken action. In Figure 5.2, LIVE will use (ON DISK1 PEG1) as both condition and prediction because ON is a relation type mentioned in the given goals (see the goal specifications in Section 4.4.2) and DISK1 is in the action Move-Put(DISK1, PEG2). (PEG2 is in the action too, but it does not have any ON relation in the current state.) Thus, LIVE creates a new rule as follows:

Condition: ((**ON** *disk pegx*)) **Action:** **Move-Put**(*disk pegy*) **Prediction:** ((**ON** *disk pegx*))

If an explorative action causes many relation changes in the environment, then LIVE will select as few of them as possible so long as the selected ones are adequate to describe the action. In addition to the relevance criterion outlined above, this is accomplished by detecting overlaps among relation changes. Consider the scenario in Figure 5.3, where LIVE is given an additional relational percept (ABOVE *diskx disky*).

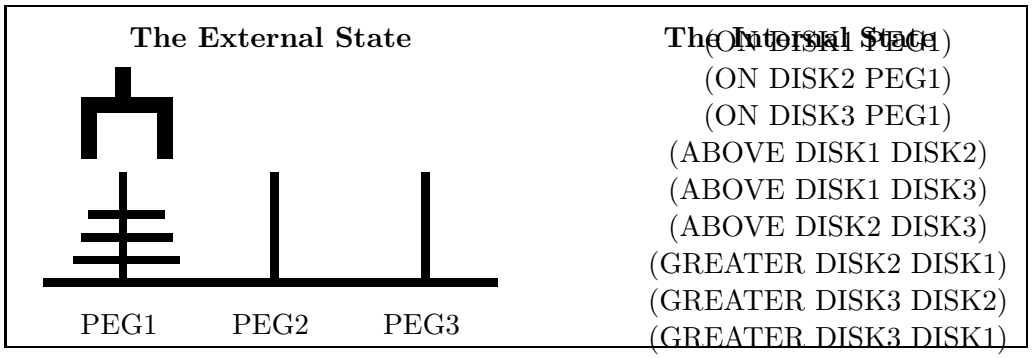
Thus, in addition to the state information perceived previously, LIVE also finds that two ABOVE relations, (ABOVE DISK1 DISK2) and (ABOVE DISK1 DISK3), have vanished after the action Move-Pick(DISK1, PEG1). In this case, before the vanished relations are used as conditions, LIVE will detect that these two ABOVE relations are overlapping, so that only one of them will be chosen. At present, a simple technique seems sufficient to detect the overlaps of relations: two relations have the same relational predicate and at least one common parameter.[1] Thus the new rule that will be created is the following:

Condition: ((**ON** *diskx peg*) (**ABOVE** *diskx disky*) (**NOT** ((**IN-HAND** *diskx*)))) **Action:** **Move-Pick**(*diskx peg*) **Prediction:** ((**IN-HAND** *diskx*) (**NOT** ((**ABOVE** *diskx disky*))) (**NOT** ((**ON** *diskx peg*))))

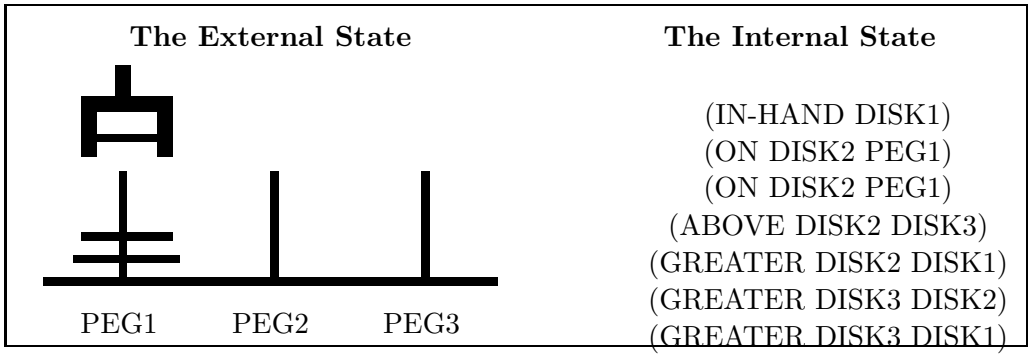
In real life, actions always make changes in the world. The reason we divide the case into both "no change" and "many changes" is that the learner's scope of attention is limited and sometimes to say an action causes no change is reasonable, even though there might

[1] An extreme but possible solution for making new rules as general as possible is to set the new rule's condiction as a don't-care symbol, and later revise the rule by pulling in relevant conditions when the rule makes mistakes. The method for revising rules is described in Chapter 6.

Before the Action:



The Action: Move-Pick(DISK1, PEG1)
After the Action:




```
Disappeared Relations          Emerged Relations
(ON DISK1 PEG1)                (IN-HAND DISK1)
(ABOVE DISK1 DISK2)
(ABOVE DISK1 DISK3)
```

Figure 5.3: Explorative action with overlapping relation changes.

be changes in a larger scale. For example, tapping your fingers on a stable desk can be characterized as no change, but if you look carefully there might be some changes on the surface of the desk and your fingers may not be at the same postions as before.

One possible difficulty in LIVE's rule creation method is that comparison of two states may be impossible in the real world because there may be an infinite number of relations that can be perceived. (This is a version of the frame problem.) For example, when comparing the two states in Figure 5.1, there may be many other irrelevant relations such as "today is

Friday" and "the Moon is full". When the action Move-Pick(DISK1, PEG1) is taken, the moon may become non-full, say it happens to be covered by some clouds. How do we know whether or not the Full-Moon relation should be included in our new rule?

To respond to this challenge, we have to back up a little to see how LIVE actually does the comparison. When information is perceived from the external world, it is represented in the system as a graph-like structure, where nodes are objects and edges are relations. After an action is taken, the comparison begins from a small set of nodes corresponding to the objects that are mentioned in the action. For example, the comparison begins from DISK1 and PEG1 in Figure 5.1. (If an action does not have any parameter, the comparison begins from the nodes corresponding to the learner's hand or body.) To look for relation changes, LIVE does incremental checking on the subgraph that is currently under consideration. For example, in Figure 5.1, LIVE compares the relations involving DISK1 and PEG1 first. If any relation differences are found there, no other objects or relations will be considered. Therefore, when hundreds or thousands objects are given, LIVE will not spread its attention if relation differences can be found among the objects that are involved in the current action.

Of course, this does not mean LIVE has overcome this difficulty completely. (After all, this is the famous frame problem in AI.) In particular, when an action does not cause any changes in the world, LIVE cannot make a "no change" conclusion unless it checks all the objects and relations in the world. This is why we have assumed, in section 2.2, that LIVE will use all the percepts that are given and that it is possible to check all the information that has been perceived. The assumption is especially important for discovering hidden features in environments for if the existing features are already beyond one's examining ability, what on earth we need to discover new, invisible features.

5.2 How LIVE Creates Rules from Objects' Features

In the last section, we described LIVE's methods for creating new rules from observed relational changes. However, when the environment becomes more realistic, the high level relations, such as ON and GREATER, may not available to the learner. What if the environment does not provide these relational percepts but features of objects instead? Can LIVE's methods still create useful rules without any given relational percepts?

5.2.1 Constructing Relations from Features

The answer to the question is yes, provided that the system is given a set of constructors which include the necessary primitive relations, such as "=" and ">". In addition to the three steps outlined above, LIVE must construct new relational predicates (using the given constructors) when comparison of states does not yield any relational changes. Let us take a close look at this problem through the hand-eye version of Tower of Hanoi.

Figure 5.4: The implementation of the hand-eye version of Tower of Hanoi.

The hand-eye environment, which we have outlined in the section 2.1, is shown in Figure 5.4. It consists of a symbolic robot with a rotatable arm along which a hand H can slide back and forth. The location of objects in this environment is represented by three elements, the degree of rotation from the arm's zero position, the distance from the base of the robot, and the altitude from the ground. For example, the peg B's location is 70.0, 38.0 and 0.0; peg C's location is 0.0, 43.3 and 0.0; the hand H's location is 45.0, 35.0 and 23.0; and the smallest disk, disk1's location is -60.0, 40.0 and 6.0. (the other two disks and peg A are at the same direction and distance as disk1, but with different altitudes.)

The robot, or the LIVE system, has been given percepts to sense object shape, size, direction, distance, and altitude, so that in this environment each object is represented as a feature vector: (*objID, shape, size, direction, distance, altitude*). *objID* is an unique identifier assigned to each object by the learning system when the object is first time seen. (Features of objects are assumed to be independent.) For example, the hand H is represented as (obj1 HAND 10.0 45.0 35.0 23.0); the three disks, from the smaller to the larger, are (obj2 DISK 5.0 -60.0 40.0 6.0), (obj3 DISK 7.0 -60.0 40.0 3.0) and (obj4 DISK 9.0 -60.0 40.0 0.0) respectively; and the peg A is (obj5 PEG 12.0 -60.0 40.0 0.0), where 12.0 is the height of the peg.

Although no relational predicates are given as percepts in this environment, LIVE has two primitive relations, = and >, as constructors. These two relational constructors will be used to construct relational predicates in the process of learning.

The robot can *rotate* the arm from -180.0 to 180.0 degrees and *slide* the hand from 0.0 to 70.0 units of distance, and it can pick up or drop objects. Thus there are four kinds of actions that can be performed by LIVE: RotateArm(o), SlideHand(d), Pick() and Drop(), where o and d are real numbers within the range of the environment. For example, in the configuration of Figure 5.4, if RotateArm(-45.0) is executed, then the arm will be at 0.0 degrees. That means the hand will be perceived in the new state as (obj1 HAND 10.0 0.0 35.0 23.0).

The robot is given the following goals to achieve: which means in English that all the

((goal-obj2 DISK 5.0 70.0 38.0 (¡ 23.0)) (goal-obj3 DISK 7.0 70.0 38.0 (¡ 23.0)) (goal-obj4 DISK 9.0 70.0 38.0 (¡ 23.0))),

disks must be located at direction 70.0 and distance of 38.0 (this is where peg B is located),

and the altitude all these disks must be lower than 23.0, the altitude of the hand. In other words, LIVE's goal is to move all three disks from peg A to peg B.

Since no relational percepts are given in this kind of environment, to create new rules as before LIVE must construct new relational predicates after comparing two states. For example, suppose LIVE decided to explore the action RotateArm(25.0) in Figure 5.4. Executing the action will give LIVE the information listed in Table 5.1 (only internal states have been shown). Note that all that LIVE can perceive from the external environment are individual objects with features. How do we proceed from there?

	Action: RotateArm(25.0)	
	The State Before	The State After
hand	(OBJ1 HAND 10.0 45.0 35.0 23.0)	(OBJ1 HAND 10.0 70.0 35.0 23.0)
disk1	(OBJ2 DISK 5.0 -60.0 40.0 6.0)	(OBJ2 DISK 5.0 -60.0 40.0 6.0)
disk2	(OBJ3 DISK 7.0 -60.0 40.0 3.0)	(OBJ3 DISK 7.0 -60.0 40.0 3.0)
disk3	(OBJ4 DISK 9.0 -60.0 40.0 0.0)	(OBJ4 DISK 9.0 -60.0 40.0 0.0)
pegA	(OBJ5 PEG 12.0 -60.0 40.0 0.0)	(OBJ5 PEG 12.0 -60.0 40.0 0.0)
pegB	(OBJ6 PEG 12.0 70.0 38.0 0.0)	(OBJ6 PEG 12.0 70.0 38.0 0.0)
pegC	(OBJ7 PEG 12.0 0.0 45.0 0.0)	(OBJ7 PEG 12.0 0.0 45.0 0.0)

Table 5.1: The states before and after RotateArm(25.0).

Comparing these two states, LIVE finds no relation changes, but some feature changes of the hand. (the feature with index 3 has been changed from 45.0 to 70.0.) With the given primitive relations = and >, LIVE begins to examine the changed feature closely. It computes the = and > relations between this particular feature and the same feature of other objects, and then compares those relations before and after the action, see Table 5.2. For example, the relation = between the third feature of HAND and the third feature of pegB has been changed from "no", ($70.0 \neq 45.0$), to "yes", (70.0=70.0). Similarly, the relation > also has some changes on the third feature between the hand and peg B.

Upon finding these relational changes, LIVE concludes that the action RotateArm(o) can change the relationships = and > among objects on the feature of index 3. Since these relation changes must be built into the new rule, LIVE will construct two new relational predicates as follows: (**elt** is a LISP function that fetches the value of indexed element in a vector.)

	Before			After		
	Index 3	=45.0?	>45.0?	Index 3	=70.0?	>70.0?
hand	45.0			70.0		
disk1	-60.0	no	no	-60.0	no	no
disk2	-60.0	no	no	-60.0	no	no
disk3	-60.0	no	no	-60.0	no	no
pegA	-60.0	no	no	-60.0	no	no
pegB	70.0	no	yes	70.0	yes	no
pegC	0.0	no	no	0.0	no	no

Table 5.2: Identify relational changes with constructors.

REL234(*objx objy*): (the meaningful name is direction=.) **TRUE $\Longleftrightarrow$ (elt objx 3) = (elt objy 3).**

REL235(*objx objy*): (the meaningful name is direction>.) **TRUE $\Longleftrightarrow$ (elt objx 3) ¿ (elt objy 3).**

With these two new relational predicates, LIVE now has the vocabulary to express the relation changes caused by the action. (This is one place where new terms are constructed for the language L, see section 4.2.) From now on, every time LIVE perceives an external state, it will compute these newly constructed relational predicates on the objects that have been perceived. Thus, LIVE's internal state will not only contain the individual objects but also some of their relations. For example, after the internal states in Table 5.1 are perceived, LIVE will compute REL234 and REL235 for the objects to get the new internal states shown in Table 5.3.

Another interesting and useful phenomenon after new relational predicates are constructed is that the goals of the system will be viewed differently. To see this, consider again the goals that are given in Figure 5.4: Since the goal objects are virtual objects (vs.

((goal-obj2 DISK 5.0 70.0 38.0 (¡ 23.0)) (goal-obj3 DISK 7.0 70.0 38.0 (¡ 23.0)) (goal-obj4 DISK 9.0 70.0 38.0 (¡ 23.0)))

the real objects in the current state), once REL234 and REL235 are defined, we can view the goal objects through the relations to see how they are related to the real objects. In order to accomplish the goals, the real objects and the goal objects must correspond. In this case, the above goals will be augmented to give the following more useful goal expression. As

Action: RotateArm(25.0)	
The State Before	The State After
(REL234 OBJ2 OBJ3)(REL234 OBJ2 OBJ4) (REL234 OBJ2 OBJ5)(REL234 OBJ3 OBJ4) (REL234 OBJ3 OBJ5)(REL234 OBJ4 OBJ5) (REL235 OBJ7 OBJ2)(REL235 OBJ7 OBJ3) (REL235 OBJ7 OBJ4)(REL235 OBJ7 OBJ5) (REL235 OBJ1 OBJ2)(REL235 OBJ1 OBJ3) (REL235 OBJ1 OBJ4)(REL235 OBJ1 OBJ5) (REL235 OBJ1 OBJ7)(REL235 OBJ6 OBJ7) (REL235 OBJ6 OBJ2)(REL235 OBJ6 OBJ3) (REL235 OBJ6 OBJ4)(REL235 OBJ6 OBJ5) (REL235 OBJ6 OBJ1)	(REL234 OBJ2 OBJ3)(REL234 OBJ2 OBJ4) (REL234 OBJ2 OBJ5)(REL234 OBJ3 OBJ4) (REL234 OBJ3 OBJ5)(REL234 OBJ4 OBJ5) (REL234 OBJ1 OBJ6) (REL235 OBJ7 OBJ2)(REL235 OBJ7 OBJ3) (REL235 OBJ7 OBJ4)(REL235 OBJ7 OBJ5) (REL235 OBJ1 OBJ2)(REL235 OBJ1 OBJ3) (REL235 OBJ1 OBJ4)(REL235 OBJ1 OBJ5) (REL235 OBJ1 OBJ7)(REL235 OBJ6 OBJ7) (REL235 OBJ6 OBJ2)(REL235 OBJ6 OBJ3) (REL235 OBJ6 OBJ4)(REL235 OBJ6 OBJ5)
hand: (OBJ1 HAND 10.0 45.0 35.0 23.0) disk1: (OBJ2 DISK 5.0 -60.0 40.0 6.0) disk2: (OBJ3 DISK 7.0 -60.0 40.0 3.0) disk3: (OBJ4 DISK 9.0 -60.0 40.0 0.0) pegA: (OBJ5 PEG 12.0 -60.0 40.0 0.0) pegB: (OBJ6 PEG 12.0 70.0 38.0 0.0) pegC: (OBJ7 PEG 12.0 0.0 45.0 0.0)	(OBJ1 HAND 10.0 70.0 35.0 23.0) (OBJ2 DISK 5.0 -60.0 40.0 6.0) (OBJ3 DISK 7.0 -60.0 40.0 3.0) (OBJ4 DISK 9.0 -60.0 40.0 0.0) (OBJ5 PEG 12.0 -60.0 40.0 0.0) (OBJ6 PEG 12.0 70.0 38.0 0.0) (OBJ7 PEG 12.0 0.0 45.0 0.0)

Table 5.3: The states before and after RotateArm(25.0) (includes relations).

more relations are constructed, it will become more and more explicit how to achieve these goals (namely, by achieving the relations one by one as we did in the RPMA environment).

((REL234 goal-obj2 OBJ2) (goal-obj2 DISK 5.0 70.0 38.0 (¡ 23.0)) (REL234 goal-obj3 OBJ3) (goal-obj3 DISK 7.0 70.0 38.0 (¡ 23.0)) (REL234 goal-obj4 OBJ4) (goal-obj4 DISK 9.0 70.0 38.0 (¡ 23.0)))

With the states expressed in terms of both relations and object feature vectors, it is now straightforward to find the relation differences between the state before the action and the state after the action. For example, in Table 5.3, it is easy to see that the relation (REL235 OBJ6 OBJ1) has vanished, while the relation (REL234 OBJ6 OBJ1) has emerged.

Just as in the RPMA environment, comparing two states here will also have the situations of "no change" or "too many changes", but the solutions we gave earlier also work

here. For example, if in Figure 5.4 LIVE did RotateArm(5.0) instead of RotateArm(25.0), then no relation change will be detected (or even constructed). As before, when no change is detected, we will only use the relations or the objects that are mentioned in the action or physically part of the learner. Thus the rule to be created will only use the feature vector of HAND as the condition and the prediction.

If an action causes many changes in the environment, for example, if LIVE did RotateArm(-105.0) instead of RotateArm(25.0), then many relation differences will be found after REL234 and REL235 are constructed. (For example, all the relation REL234 between HAND and the disks will become true after the action is taken because they all have -60.0 as their 3rd feature.) In this case, LIVE will not use all the disks in the rule, instead, it will delete those overlapped relations and chooses only one, any one, of the following relations to be included in the rule:

(REL234 OBJ1 OBJ2) *; hand is direction= to disk1.* (REL234 OBJ1 OBJ3) *; hand is direction= to disk2.* (REL234 OBJ1 OBJ4) *; hand is direction= to disk3.* (REL234 OBJ1 OBJ5) *; hand is direction= to pegA.*

5.2.2 Correlating Actions with Features

Once relations are included in the internal states, creating new rules can be pursued as before. In particular, LIVE will first compare the states in Table 5.3 to find the relation differences caused by the action, which yields (REL235 OBJ6 OBJ1) as a vanished relation and (REL234 OBJ6 OBJ1) as an emerged relation. Then, LIVE will correlate these relations with the action RotateArm(25.0) before it puts all the information together to form a new C-A-P rule.

However, in this environment, correlation between actions and percepts is not automatically satisfied as in the RPMA environment because RotateArm(25.0) does not mention any particular objects as Move-Pick(DISK1, PEG1) did. In order to correlate the action with the percepts, LIVE must find a relation between the number 25.0 and the changes of features in the states, namely, 70.0 and 45.0, which are the values of the third feature of the hand before and after the action.

To find the relation between 25.0 and (70.0 45.0), LIVE again depends on its constructors. It will search through all the given constructors and find one that can fit in this example.

The search is simple in this case because among all the given constructors (=, >, and -), only - can fit in, that is, 25.0 = 70.0 - 45.0. (Of cause, LIVE could make a mistake here if some other constructors, f, also fit in this single example, say 25.0 = f(70.0, 45.0). But LIVE's learning is incremental. If f is not the right correlation between RotateArm's parameter and the feature changes of the hand, it will make a wrong prediction in the future, and - will replace f eventually.)

Since the action's parameter is related to the 3rd feature of the HAND, the condition and the prediction of the new rule will have to include the HAND object and the values of its third feature before and after the action. In particular the correlation will look like this:

- (HAND # # 45.0 # #)
- RotateArm((70.0 − 45.0))
- (HAND # # 70.0 # #)

After finding the relation differences and correlation between the action and the features that have been changed, a new C-A-P is ready to be created. Just as in the RPMA environment, LIVE will use the vanished relation and the negations of emerged relations as the condition, use emerged relations and the negations of the vanished relations as prediction, and then generalize all the constants to variables (different constants will be generalized to difference variables). In addition to that, since individual objects are perceivable in the hand-eye environment, they will also be included in the rule if their identifiers are mentioned in the relevant relations. For example, we will include both the HAND and pegB because they are involved in the relations. (HAND is also involved in the correlation between the action and the percepts.) Moreover, since HAND is physically a part of the learner itself and future applications of this rule will always refer to the hand as a special object, it will not be generalized into variable but keeps its identity. Thus, a new rule is finally created as follows:

Condition: ((REL235 *objx* **HAND) (NOT ((REL234** *objx* **HAND))) (HAND # # x_1 # #) (*objx* # # # # #)) Action: RotateArm**($x_2 - x_1$) **Prediction: ((REL234** *objx* **HAND) (NOT ((REL235** *objx* **HAND))) (HAND # # x_2 # #) (*objx* # # # # #))**

At this point, we have given some long and detailed examples of LIVE's rule creation method; we now complete the description. When an explorative action (an action whose

outcomes cannot be predicated by any of the existing knowledge) is performed, the system LIVE will create a new rule following these steps:

1. Compare the states before and after the action and find the vanished and emerged relations;

2. If no relation differences are found, but objects' feature are changed, then use the given constructors to construct new relational predicates, and find the relation changes in terms of these new relational predicates;

3. Correlate the changes of the state with the action that is taken;

4. Create a C-A-P rule as follows:

   Condition: the vanished relations, plus the negations of the emerged relations, plus the objects involved in these relations or in the action;
   Action: the action that is taken with parameters expressed in terms of objects' identities, or in terms of changed features with a proper constructor;
   Prediction: the emerged relations, plus the negations of the vanished relations, plus the objects that are involved in these relations or in the action.

   and then generalize all objects in the rule into variables except objects belonging to the learner itself, such as its hand, arm or body.

5.3 How LIVE Explores the Environment

5.3.1 The Explorative Plan

In the description of rule creation, we have been assuming that LIVE always chooses the right action to explore, which of course is not the case in reality. When exploring an unfamiliar environment, there are many things you can do; choosing the relevant explorative action among a large number of possibilities is not a easy task. For example, in exploring the RPMA environment in Figure 5.1, LIVE has many actions to choose from. Why should it explore Move-Pick(DISK1, PEG1) rather than Move-Pick(DISK2, PEG1) or Move-Put(DISK3, PEG2)? In the hand-eye environment, Figure 5.4, the situation is even worse because some actions contain real numbers as parameters, like o in RotateArm(o)

and d in SlideHand(d), and in principle these parameters can have an infinite number of values.

This section describes the exploration module, a module that is designed to deal with such tasks. In particular, we will describe how it generates *explorative plans* and execute them to provoke and observe novel phenomena from which new rules can be created.

An explorative plan is a sequence of actions, usually without predictions. For example, when you step into a complex building for the first time, you might take a walk without knowing for sure what you will see after turning a corner. As you can imagine, an explorative plan is different from a solution plan because each of its steps may not be associated with any particular prediction. However, LIVE executes an explorative plan in the same manner as it executes a solution plan. Given an explorative plan, LIVE will execute the actions in the plan from the beginning to the end. Whenever it is about to take an action, LIVE will search its internal model to see if any existing rules can be used to make any predictions about the consequences of this action. If predictions are made, LIVE will execute the action and compare the prediction to the actual outcome of the action. (In this case, before LIVE will execute the next explorative action, the learning module might be called if the prediction does not match the actual outcomes, as we described in section 4.4.) If no prediction is found, LIVE will simply execute the action and observe the outcomes and use the observation to create a new rule (see Section 5.1 and 5.2.).

Although explorative plans have some accidental character, they are far from random actions. For example, a random action in the RPMA environment means any one of the following possible actions:

Move-Pick(disk1, pegA), Move-Pick(disk2, pegA), Move-Pick(disk3, pegA),
Move-Pick(disk1, pegB), Move-Pick(disk2, pegB), Move-Pick(disk3, pegB),
Move-Pick(disk1, pegC), Move-Pick(disk2, pegC), Move-Pick(disk3, pegC),
Move-Put(disk1, pegA), Move-Put(disk2, pegA), Move-Put(disk3, pegA),
Move-Put(disk1, pegB), Move-Put(disk2, pegB), Move-Put(disk3, pegB),
Move-Put(disk1, pegC), Move-Put(disk2, pegC), Move-Put(disk3, pegC).

But a good explorative plan will be Move-Pick(DISK1, PEG1) followed by Move-Put(DISK1, *anypeg*) because all we want to know is how to move a disk from one peg to another. (Move-Pick(DISK1, PEG1) is the only action that can make changes in the initial environment, shown in Figure 5.1.)

Similarly, if one explores the hand-eye environment randomly then he might spend his life time in that environment without learning anything interesting. A successful learner should certainly do better than random explorations.

5.3.2 Heuristics for Generating Explorative Plans

LIVE generates its explorative plan in a unfamiliar environment with a set of domain independent heuristics. These heuristics are executed in the order as listed in Table 5.4. The general idea is that exploration should be guided by the existing knowledge and the desired goal. Thus, all these heuristics have access to LIVE's internal model and are aimed to suggest more concise and productive explorative plans. Let us illustrate these heuristics through a set of examples.

- If the internal model is empty, then propose each action once with its parameters replaced by a randomly generated value.
- If there exists a rule that can change an object's feature F_i, then use the rule to propose an action that will cause the learner to have the same value of F_i as the current F_i of objects mentioned in the goal.
- Always prefer actions that have never been observed to cause any changes in the environment.

Table 5.4: Heuristics for generating explorative plans.

To see how the first heuristic works, suppose the exploration module is called in the hand-eye environment, see Figure 5.4, when LIVE's internal model is empty, then the following explorative plan is an example that can be proposed by the heuristic: This is because the

RotateArm(-30.0), SlideHand(10.0), Pick(), Drop().

four actions are all that available to LIVE and -30.0 and 10.0 are random values between -180.0 to 180.0 and 0.0 to 70.0 respectively. With this explorative plan, four rules will be learned. Two of them are about RotateArm and SlideHand which indicate that these actions will change Arm's rotating degree and hand's distance respectively. The other two learned rules are about Pick and Drop although they indicate no change at all in the current environment because there is no object under or in the hand at this moment. In

this particular explorative plan, the order of these four actions does not make it any better or worse.

After these four rules are learned, the second and third heuristics will be useful. If the current active goal is (goal-obj2 DISK 1 +70.0 38.0 (¡ 23.0)), then the following explorative plan will be generated by these two heuristics: In this explorative plan, the first two actions

RotateArm(-75.0), SlideHand(-5.0), Drop(), Pick().

are proposed by the second heuristic because LIVE has learned rules that can change objects' direction and distance, and the hand, whose current location is (+15.0, 45.0, 23.0), should be manipulated to have the same direction and distance values as the goal object DISK1, whose current location is (-60.0, 40.0, #). The actions Pick() and Drop() are proposed by the third heuristic because these two actions have not produced any observed changes in the environment so far. As we know from LIVE's rule creation method, these four actions will give the system opportunities to construct new relational predicates and learn rules that move its hand to a desired location and pick up and drop objects.

In the RPMA environment, these heuristics can also direct the exploration into a more productive path. When learning has just begun (no rule has been learned), the first heuristic will force LIVE to repeatly try the two actions with randomly generated values for the disk and the peg parameters until Move-Pick(DISK1, PEG1) is selected. (which might take long time.) This is because in the initial state (Figure 5.1), where all three disks are on PEG1, Move-Pick(DISK1, PEG1) is the only action that can cause some changes. Once DISK1 is picked up, LIVE will repeatedly to explore the action Move-Put(*disk, peg*), as suggested by the third heuristic, until this action has made some changes in the environment, i.e. DISK1 is put down on some peg.

The heuristics are designed with two motivations. First, exploration is to provoke and observe new phenomena. The heuristics encourage the learner to interact with the objects that are relevant to the goals and try the actions that are less unfamiliar. For example, to solve the goal (ON DISK1 PEG1), we should interact with DISK1 or PEG1; to learn how to swim, one should interact with water; to get the toy (see Figure 2.2), the child should play with the level instead of waving his/her hands in the air or stumping on the floor. The second motivation is that we believe exploration should depend on the existing knowledge; that is why all the heuristics have access to LIVE's internal model. Finally, we point out

that these heuristics represent only the beginning of the research on the exploration issue; they are by no means complete.

5.4 Discussion

5.4.1 The Pros and Cons of LIVE's Rule Creation Method

While describing the examples, we have mentioned briefly the advantages of LIVE's rule creation method, let us summarize here. First, this method has provided a way to correlate actions with percepts, which is a very important step in learning from environments. Unless actions and percepts are correlated, one cannot act purposefully in an environment. Next, the method looks for conditions and effects of an action around the action itself, which gives a way for focusing the learner's attention if the environment is vast. Although it is not a complete solution for the frame problem, it does work well in some cases. Finally, this method overcomes, to some extent, the incompleteness of the concept description language. When new relational predicates are necessary to describe the environment, it will construct them in the process of learning. In doing so, the method abstracts a continuous environment into discrete relations which are essential for solving problems.

Compared to some existing rule learning mechanisms, such as Carbonell and Gil's learning from experimentation [9] in the domain of making telescopes, our method shows that some of the learnings in their program can be saved at rule creation time. For example, the initial operator for ALUMINIZE in their program has assumed to have a missing prediction ¬(IS-CLEAN *obj*). In fact, such a prediction will be included here at rule creation time because the fact of (IS-CLEAN *obj*) is changed when the action ALUMINIZE is applied.

As the reader may have already noticed, there are several drawbacks of this method. First, the method cannot use objects' properties alone as building blocks; everything must be said in terms of relations. As a consequence it always requires a "reference" to express the values of features. For example, LIVE cannot learn the rule that says "If a disk is RED, then do this," but instead creates a rule says "If (SAME-COLOR disk X), then do this," which requires that there must exist an object X whose color is red. Second, as we pointed out earlier, in order to conclude that an action does not cause any changes in the environment, this method must check all the relations in the environment, which can be quite expensive. For example, if there are n objects and m binary relational predicates,

then there will be $m(n(n-1))$ potential relation links because each predicate could have $n(n-1)$ links and there are m relations. Finally, in order to construct useful relational predicates, the system must be given correct constructors. For example, if = is not given, REL234 cannot be learned. But while this is a drawback, it also has some good points. We can provide our knowledge to the system through these constructors to focus its learning.

5.4.2 Can We Use More Primitive Percepts and Actions?

As a closing statement of this rule creation chapter, we ask ourselves this question: will this approach be feasible if lower level percepts and actions, like those used on a mobile robot, are given? Can the method really scale up? We have already shown that rules can be constructed from relational percepts and macro actions, and from features of objects and movements of a robot arm and a set of primitive relational constructors. But if the aim of this approach is to construct a real mobile robot that learns from environments, this is the question that must be answered, even though not necessarily in this particular thesis. In the literature of qualitive reasoning, there are already some positive evidence to show that complexity may not explore if one keeps in mind that most reasonings are at the qualitive level. Kuipers's map learning at a qualitive level [**?**] provides a very good example.

Although it is still too early to predict the outcome, we think this approach is promising because it contains some viable ingredients. First of all, the target is fixed in this approach. No matter how you implement it, the final product should be the same: knowledge that can help the robot reason about its actions and solve problems. The C-A-P is only one implementation (though it is a convenient one). Second, the studies so far have revealed a hierarchy of percepts and actions. From top down, this hierarchy contains the C-A-P rules, the relations of objects, and the features of objects. Going further down in this hierarchy, it is clear that the next building blocks must be some kind of spatial representations from which features of objects can be constructed. Finally, since this approach constructs new terms when necessary, the learner's initial description language does not have to be complete. We think this is a very important point in machine learning in general because we believe that, in a very primitive sense, all the knowledge we have is constructed from what we can see and what we can do.

Chapter 6

Rule Revision by Complementary Discrimination

In this chapter, we will analyze the challenges of revising rules while learning from environments and introduce the complementary discrimination learning method based on some examples both from concept learning and rule revision. We will also explain how experiments are designed to correct the faulty rules detected at the planning time. At the end of the chapter, we will compare LIVE's learning method with some existing techniques and argue that the method has provided a new approach for incrementally learning both conjunctive and disjunctive rules from environments with considerably less representational biases.

6.1 The Challenges

In Chapter 3 and Chapter 4, we have defined learning from environments as a problem of inferring rules from environments and argued that such learning must be interleaved with problem solving. Although many authors have made significant contributions to the area of rule learning [8], the requirements of learning from environments have brought out some distinctive challenges.

First, since learning from environments starts with very little knowledge, techniques that rely on concept hierarchies that given at the outset, like Version Spaces [32] and Concept Clustering [**?**], are difficult to apply here because one does not know beforehand what hiearachy is necessary.

Second, since learning from the environment requires the learner to learn incrementally

both conjunctive and disjunctive rules, the Version Space algorithm may fail to converge in its predefined concept space [8], and Classification methods like ID3 [38] may not get enough training instances to learn the correct rules because they require all the training instances to be given beforehand. Although there are many incremental methods for learning concepts from examples [31, 40, 60], only a few solved problems [32] and learned disjunctive rules [58].

The third and the most fundamental challenge is that the rule description language may not be adequate for learning the correct rules, as Mitchell and Utgoff addressed in [32, **?**]. In some environments, as we will see in Chapter 7, in order to learn the correct rules, LIVE must infer and construct hidden features and their relations in terms of both the concept description language and the action description language.

Faced with these challenges, it seems that learning by discrimination [7, 15, 24] is the most feasible choice. It is incremental, and it does not require any concept hiearachy given at the outset. Moreover, since discrimination learning specializes the learned concept by learning from mistakes, it seems very helpful for learning from environments because one needs to make predictions even if one's knowledge is not perfect yet. Recently, this method has attracted more attention in the Machine Learning literature [14, 35].

However, as Bundy has pointed out [8], rule learning is a "two way" search in some concept hierarchy (or some lattice with generalization as the partial order). One way is generalization which goes bottom-up, and the other is discrimination (or specialization) which goes from top-down. Any one way learning will suffer some incompleteness. For example, pure discrimination goes only from general to specific; it is difficult to recover from errors and to learn from positive examples. Moreover, to learn disjunctive rules by discrimination, all the existing methods rely totally on getting what Bundy called a *far misses*[1] [8], which depends very much on the order of the instances and is not always available. For these problems, previous methods for learning by discrimination have not provided any flawless solutions. Maybe that is why this powerful learning method which originated many years ago by John Stuart Mill [34] has not gained much popularity in the Machine Learning literature.

As we will see now, LIVE's learning method is an advanced discrimination method. It provides a solution to the above questions by learning not only the concept itself, but also

[1] More than one differences must be found between two states.

- A set of objects:

  Shape: circle, sqaure, triangle.
  Size: samll, large.

- The target concept:

  (*shape*=circle)∨(*shape*=square)

- Examples:

  (+ (small circle)), (- (large triangle)), (+ (large circle)), (+ (large square)).

Figure 6.1: A simple learning task.

its complement. This method performs both generalization and discrimination because generalizing a concept is equvilant to discriminating its complement, and vice versa. Moreover, the result that a concept and its complement go hand by hand during the learning process provides a convenient way for both designing experiments to recover from discrimination errors and discovering hidden features when necessary.

6.2 Complementary Discrimination

6.2.1 A Simple Learning Task

Before we give the algorithm for Complementary Discrimination, let us start with a very simple but informative example. Figure 6.1 is a learning task modified from the Handbook of Artificial Intelligence [6, pages 388–391].[2]

Here we see that the represenation language specifies an instance space that contains a set of objects with two features: *size* and *shape*. The size of an object can be *small* or *large*, and the shape of an object can be *circle, square*, or *triangle*. The goal of this task is to learn the target concept (cr ∨ sq) based on the observation of given examples. Those marked with "+" are positive examples, and those marked with "-" are negtive examples.

Although this is a very simple learning task, many existing learning methods fail to learn the target concept. As an example, let us consider Version Spaces [31]. Figure 6.2 shows an initial version space for this representation language. We use a variable x to represent

[2]The original target concept in the handbook is (*shape*=circle).

the size, and a variable y to represent the shape.

Figure 6.2: The initial version space.

Figure 6.3: Updating the version space.

Figure 6.3 shows how the space is updated when examples are presented. After three training instances are processed, instead of drawing a natural conclusion (x ¬triangle), the version space jumps to the conclusion (x cr) even though it has never seen an example

Figure 6.4: The intuition of complementary discrimination.

of squares. When the fourth positive instance (lg sq) is presented, the version space fails to learn anything further. This failure is partially caused by the initial concept hierarchy which is a stronge bias that leds the concepts of square and triangle to be knocked out in the first step. Another partial reason is that the target is a disjunctive concept.

6.2.2 The Algorithm

Complementary Discrimination is a new method for learning from examples. As shown intuitively in Figure 6.4, the method learns both the target concept **C** and the complement of the target $\overline{\mathbf{C}}$. It starts from the most general case, i.e., taking the whole instance space as one concept and the empty set as its complement. Based on the observations on the training instances, it incrementally learns where to draw the boundary between **C** and $\overline{\mathbf{C}}$. When a training instance is presented, the method will predict whether the instance belongs to **C** or $\overline{\mathbf{C}}$. If the prediction is correct, then no change is necessary on the current concept. If the prediction is wrong, the the method will discriminate the failure from the previous successes by finding the facts that were true in the successful instances but are false in the failed instance. (see the examples below.) The difference is then used to specialize, or "shrink", the concept that made the mistake and to generalize, or "expand", its complement. That is equivalent to adjusting the boundary between **C** and $\overline{\mathbf{C}}$ in the instance space. Figure 6.5 shows the algorithm in detail.

Let U be the instance space, $C \leftarrow U$ and $\neg C \leftarrow \emptyset$; While instance i Predict X where $i \in X$; ;;; X can be C or $\neg C$
If the prediction is correct then put i as X's example else find differences D between i and X's examples; $X \leftarrow X \wedge D$; $\neg X \leftarrow U \setminus X$; put i as $\neg X$'s example.

Figure 6.5: Complementary Discrimination for concept learning.

To show how the algorithm works, let us go back to the current learning task. Unlike Version Spaces, Complementary Discrimination (CD) does not need any initial concept hiearachy to start with. Instead, all that it needs is the information about complements: the feature *size* has two values *small* and *large*, and the feature *shape* has three values *circle, square*, and *triangle*. When the first training instance (sm cr) is given, the algorithm forms its initial hypothesis: The symbol # means "don't care" and the square brackets contain

C: $(x = \#y = \#)$	[(sm cr)] $\overline{\mathbf{C}}$: NIL	[ ]

the successful instances for the concept. When the second instance (lg tr) is presented, the algorithm predicts that the instance belongs to **C** but the prediction fails. It then compares (sm cr), the success, with (lg tr), the failure, and finds that the difference is $(x = sm) \wedge (y = cr)$ which is true in the success but false in the failure. Putting the difference in conjunction with the existing **C**, the algorithm shrinks or specilizes **C** as follows: By negating the new **C**, the algorithm constructs a new $\overline{\mathbf{C}}$ as follows: Note that it

C: $(x = sm) \wedge (y = cr)$;	[(sm cr)]

$\overline{\mathbf{C}} : \neg((x = sm) \wedge (y = cr)) \Rightarrow \neg(x = sm) \vee \neg(y = cr) \Rightarrow \neg(x = sm) \vee (y = sq) \vee (y = tr)$ [(lg tr)]

is permissible to write $\neg(y = cr)$ as $(y = sq) \vee (y = tr)$ because it is known that the feature y can only have these three values.

When the third instance (lg cr) is given, the algorithm predicts that the instance is negative because it matches $\overline{\mathbf{C}}$. After being surprised, the algorithm compares (lg tr), the success of $\overline{\mathbf{C}}$, with (lg cr), the failure of $\overline{\mathbf{C}}$, and finds that the difference is $(y = tr)$. Therefore, the concept $\overline{\mathbf{C}}$ is specalized by a conjuction with the difference, and the concept **C** is modified as the negation of the new $\overline{\mathbf{C}}$:

$\overline{\mathbf{C}}$: $(\neg(x = sm) \vee (y = sq) \vee (y = tr)) \wedge (y = tr) \Rightarrow (y = tr)$	[(lg tr)] **C**:
$\neg(y = tr) \Rightarrow (y = cr) \vee (y = sq)$;	[(sm cr) (lg cr)]

At this point, the algorithm has learned the correct concept (*shape*=circle)$\vee$(*shape*=square)

as its concept **C**. When the fourth instance (lg sq) is presented, the algorithm will correctly predict it as a positive instance of **C**.

It is interesting to see how the algorithm will learn further if the concept were $(y = cr)$ instead of $(y = cr) \vee (y = sq)$. If this were the case, then the fourth instance (lg sq) would be negative and the algorithm's last prediction would fail. That would force the algorithm to discriminate the concept **C** further. In that case, the algorithm would compare the previous success (lg cr)[3] with the failure (lg sq) and find that the difference is $(y = cr)$. Thus, the concept **C** is specialized and $\overline{\mathbf{C}}$ follows as the negation of the new **C**:

C: $((y = cr) \vee (y = sq)) \wedge (y = cr) \Rightarrow (y = cr)$; [(sm cr) (lg cr)] $\overline{\mathbf{C}}$: $\neg(y = cr)$ [(lg tr) (lg sq)]

As we can see from this simple example, Complementary Discrimination is farily straightforward learning algorithm. The key idea is to learn not only the concept itself but also its complement. Thus the bias caused by the initial concept hierarchy can be eliminated. Since CD is to find the correct boundary in the instance space and it is capable of learning both conjuctives and disjunctives, the algorithm can learn any concept that is in the power set of the instance space. It performs both discrimination and generalization because generalizing a concept is equivalent to discriminating its complement. Its learning is also incremental since whatever have been learned so far can always be used to make predictions to solve problems. Although we have not given any theoretical proofs, it seems emperically that the more examples the algorithm sees the closer the learned boundary to the target concept. Moreover, this new learning method suggests a view to unify learning from differences and learning from similarity because in this algorithm similiarities to a concept are the same as differences to the concept's complement.

6.3 How LIVE Revises its Rules

Complementary Discrimination can be easily applied to revising incorrect rules. Let us first go through an example. As we saw in the last chapter, the rules created by LIVE during exploration are often over-general and incomplete. The following is one of them that created

[3]There are two successful instances of **C** at this point, the algorithm will choose (lg cr) because it is closer to the failure. At worst, the algorithm will compare the failure to all the successes and find the most general difference, which is $(y = cr)$.

in the RPMA environment shown in Figure 6.6 where LIVE successfully picked up DISK1 from PEG1:

Rule Index: 6-1 Condition: ((ON *diskx peg*) **(NOT ((IN-HAND** *diskx*)))) **Action: Move-Pick(***diskx peg***) Prediction: ((IN-HAND** *diskx*) **(NOT ((ON** *diskx peg*)))) **Sibling: ()**

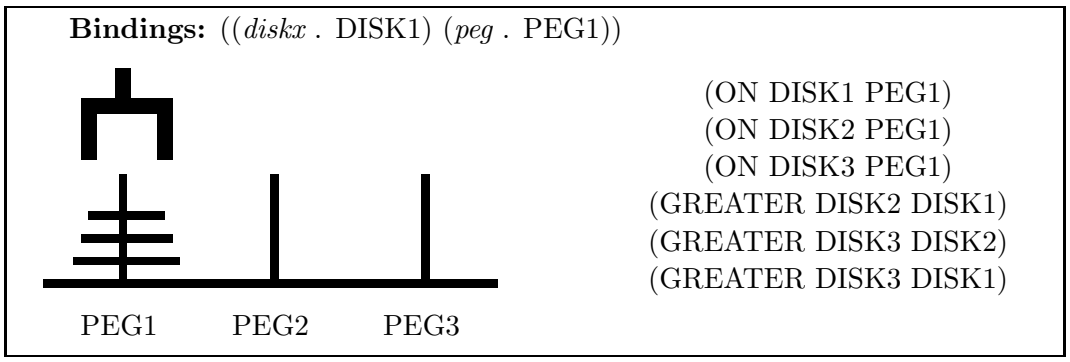


Figure 6.6: A successful application of Rule6-1.

This Rule6-1 is incomplete. One problem is that if the hand is not empty then the rule's prediction will fail (because the *diskx* will not be in the hand but still on the *peg*). Clearly, in using these kind of rules to solve problems, LIVE will inevitably make mistakes.

However, no matter how incomplete the new rules may be, they provide a springboard for further learning from the environment. It is because of their generality and incompleteness that LIVE will have chances to be surprised, to learn from mistakes, and hence to increase its knowledge about the environment. The question is how LIVE detects these incomplete rules and corrects them after mistakes are made.

Detecting incomplete rules is straightforward because all the rules make predictions and LIVE compares their predictions with the actual outcomes in the environment whenever an action is taken. If a rule's prediction does not match the actual outcome of the rule's action, then the rule is wrong. In that case, we say LIVE has encountered a *surprise*.

As an example of a surprise, suppose that Rule6-1 is applied in the state shown in Figure 6.7, where LIVE is trying to pick up DISK2 even if its hand is not empty. To get the desired result (IN-HAND DISK2), *diskx* is bound to DISK2 and *peg* bound to PEG1 (because DISK2 is on PEG1). Unfortunately, after the action Move-Pick(DISK2, PEG1),

LIVE is surprised to see that DISK2 is not in the hand but still on PEG1 (because the environment allows only one disk to be picked at a time). At this point, LIVE realizes that Rule6-1 is wrong.

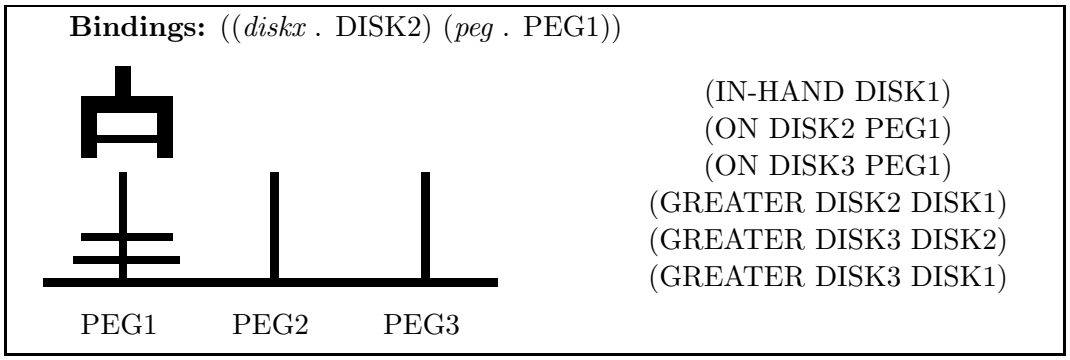


Figure 6.7: A failure application of Rule6-1.

In general, to revise an incomplete rule, LIVE will discriminate the failed application from a successful application and find out *why* it is surprised. It will bring up a remembered success, such as Figure 6.6, and compare it to the failure in Figure 6.7. (LIVE remembers the latest successful application for each existing rule, and the memorized information contains the rule index, the state, and the set of variable bindings.) The differences found in the comparison are the explanations for the surprise and will be used to split the faulty rule into two more specific and more complete rules. The two new rules are known as siblings.

In this example, comparing Figure 6.6 with Figure 6.7 will result in the difference (NOT (INHAND *disky*)). In other words, the reason Rule6-1 succeeded before is because there was nothing in its hand. (Why this reason is found but not others will be explained in Section 6.2.2.) With this explanation, Rule6-1 is then split into the following two new rules:

Rule Index: 6-3 Condition: ((ON *diskx peg*) **(NOT ((IN-HAND** *diskx*))) **(NOT ((IN-HAND** *disky*)))) **Action: Move-Pick(***diskx peg***) Prediction: ((IN-HAND** *diskx*) **(NOT ((ON** *diskx peg*)))) **Sibling: 6-5**

Rule Index: 6-5 Condition: ((ON *diskx peg*) **(NOT ((IN-HAND** *diskx*))) **(IN-HAND** *disky*)) **Action: Move-Pick(***diskx peg***) Prediction: ((ON** *diskx peg*)) **Sibling: 6-3**

Note that Rule6-3 is a variant of Rule6-1 with the condition augmented by the the

difference; Rule6-5 is a new rule whose condition is the conjunction of the old condition and the negation of the difference[4], and whose prediction (constructed by the same method used in the rule creation) is the observed consequences of the current application.

6.3.1 Applying Complementary Discrimination to Rule Revision

From this simple example, we can now state LIVE's rule revision method as the following three step procedure:

1. Detect an erroneous rule by noticing a *surprise*. As we mentioned in the above example, a surprise is a situation where a rule's prediction does not match the outcome of the rule's action. Technically, since a prediction is an expression in LIVE's description language and an outcome is an internal state perceived from the external environment, a surprise is formally defined as follows:

   Surprise $\Longleftrightarrow \neg$(*MATCH* **prediction internal-state)**,

   where "*MATCH*" is LIVE's matcher described in Chapter 4.

2. Explain the surprise by finding the differences between the rule's previous success and its current failure. Given the state from a success and the state from a failure, along with the rule's variable bindings in both cases, LIVE conducts a comparison from the *inner* circles of the states to the *outer* circles of the states (to be defined shortly). The final result of this step is a set of differences, $(D_1\ D_2\ \ldots\ D_j)$, where D_j is either a conjunction of relations or a negation of conjunctive relations that is true in the successful state but false in the surprising state.

3. If the erroneous rule m does not have any siblings, then it is split into two new, sibling rules as follows (the I-Condition is constructed at the rule creation time):

[4](IN-HAND *disky*) is a replacement of (NOT (NOT ((IN-HAND *disky*))))

		Index m
		I-Condition $\wedge D_1 \wedge D_2 \ldots \wedge D_j$
	$\Longrightarrow$	Action
Index m		Prediction
I-Condition		Sibling n
Action		
Prediction		Index n
Sibling ()		I-Condition $\wedge\neg(D_1 \wedge D_2 \ldots \wedge D_j)$
	$\Longrightarrow$	Action
		Observed Effects
		Sibling m

The first rule is a variant of the old rule, whose condition is conjuncted with the differences found in the step 2; the second rule is a new rule, whose condition is a conjunction of the old condition with the negation of the differences, and whose prediction (constructed by the same method used in the rule creation) is the observed consequences in the current application.

If the erroneous rule m already has a sibling, then modify both of them as follows:

Index m		Index m
I-Condition $\wedge C_1 \ldots \wedge C_i$		I-Condition $\wedge C_1 \ldots \wedge C_i \wedge D_1 \ldots \wedge D_j$
Action	$\Longrightarrow$	Action
Prediction		Prediction
Sibling n		Sibling n
Index n		Index n
I-Condition $\wedge O_1 \ldots \wedge O_i$		I-Condition $\wedge\neg(C_1 \ldots \wedge C_i \wedge D_1 \ldots \wedge D_j)$
Action	$\Longrightarrow$	Action
Prediction		Prediction
Sibling m		Sibling m

Conditions $C_1, \ldots, C_i$ and $O_1, \ldots, O_i$ are the conditions that are learned after the rule is created. For example, (NOT ((IN-HAND *disky*))) in Rule 6-3 and (IN-HAND *disky*) in Rule 6-5 are such conditions. Notice that at any time the learned conditions of the sibling rules are complement to each other, and that is why the method is called *complementary discrimination*.

In this rule revision method, the second step is the most crucial step because it finds the reasons why the rule's prediction fails. In order to find the most relevant reasons

for the surprise, LIVE divides each state into the inner and outer circles according to the rule's variable bindings in that state. The *inner circle* of a state includes the objects that are bound to variables, the learner itself, and all the relations among these objects. For example, in Figure 6.6, the inner circle includes one relation (ON DISK1 PEG1) because this is the only relation among the bound objects DISK1 and PEG1. In Figure 6.7, the inner circle includes two relations (ON DISK2 PEG1) and (IN-HAND DISK1) because DISK2 and PEG1 are bound objects and IN-HAND is a part of the learner itself. The *outer circle* of a state includes the inner circle as well as those objects that are related to the inner circle objects through some relations. For example, in Figure 6.6, the outer circle includes (ON DISK1 PEG1), (ON DISK2 PEG1), (ON DISK3 PEG1), (GREATER DISK3 DISK1) and (GREATER DISK2 DISK1) because they all contain either DISK1 or PEG1. In Figure 6.7, the outer circle includes (IN-HAND DISK1), (ON DISK2 PEG1), (ON DISK3 PEG1), (GREATER DISK3 DISK2) and (GREATER DISK2 DISK1) because they all contain either DISK2 or PEG1 or IN-HAND which is in the inner circle. When two states are to be compared, LIVE will find the differences in the inner circles before considering the outer circles, as we will see in the next two subsections.

6.3.2 Explaining Surprises in the Inner Circles

In this section, we will explain how LIVE finds differences in the inner circles.[5] Continuing our example of revising Rule6-1 to Rule6-3 and Rule6-5, let us see how (NOT ((IN-HAND *disky*))) is found by comparing Figure 6.6 with Figure 6.7.

For convenience, we reprint the states and the bindings from both figures and their inner circles in the following table:

[5]It has been pointed out that LIVE's method for searching differences between two states is very similar to Langley's method used in his learning program SAGE. Interested readers may find more descriptions in [24, 22, 23].

	The Success	The Failure
States	(ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG1) (GREATER DISK2 DISK1) (GREATER DISK3 DISK2) (GREATER DISK3 DISK1)	(IN-HAND DISK1) (ON DISK2 PEG1) (ON DISK3 PEG1) (GREATER DISK2 DISK1) (GREATER DISK3 DISK2) (GREATER DISK3 DISK1)
Bindings	((*diskx* . DISK1) (*peg* . PEG1))	((*diskx* . DISK2) (*peg* . PEG1))
Inner Circle	(ON DISK1 PEG1)	(ON DISK2 PEG1) (INHAND DISK1)

The inner circle of the failed state includes the relation (ON DISK2 PEG1) because both DISK2 and PEG1 are bound objects and this is the only relation between them. It also includes (INHAND DISK1) because INHAND is a part of the learner itself. In the successful state, the inner circle includes only (ON DISK1 PEG1).

To find the differences between these two inner circles, LIVE first replace the objects by the original variables specified in the bindings. The resulting inner circles are the following:

	The Success	The Failure
Inner Circle	(ON *diskx peg*)	(ON *diskx peg*) (INHAND DISK1)

Comparing these two sets, it is clear that (IN-HAND DISK1) is in the failed state but not in the successful state. Since DISK1 is not bound in the bindings of the failed application, it is generalized as *disky*. Thus the comparison returns (NOT ((IN-HAND *disky*))), which is true in the successful application but false in the failed application.

6.3.3 Explaining Surprises in the Outer Circles

It is not always the case that differences between two states can be found in the inner circles. When outer circle objects must be considered, LIVE will use a "chaining" method, similar to the one used in [24], to find a set of relations that are true in one state but false in the other.

Let us consider the rule Rule6-3, reprinted below, that was just learned and suppose that LIVE is trying to pick up DISK3 in the state shown in Figure 6.8. The bindings are ((*diskx* . DISK3) (*peg* . PEG1)), and LIVE executes Move-Pick(DISK3 PEG1) to get (INHAND DISK3). The outcome surprises LIVE because DISK3 is not in the hand but still on PEG1.

Rule Index: 6-3 Condition: ((ON *diskx peg*) **(NOT ((IN-HAND** *diskx*))) **(NOT ((IN-HAND** *disky*))) **Action: Move-Pick(***diskx peg***) Prediction: ((IN-HAND** *diskx*) **(NOT ((ON** *diskx peg*))))

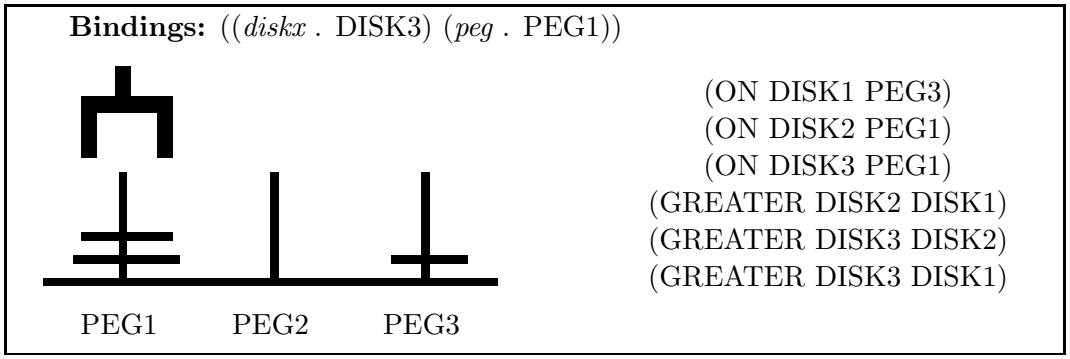


Figure 6.8: Finding differences in the outer circles.

Comparing this to the rule's last successful application, which is in Figure 6.6, LIVE constructs the inner and outer circles of the two states in the following table:

	The Success	The Failure
States	(ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG1) (GREATER DISK2 DISK1) (GREATER DISK3 DISK2) (GREATER DISK3 DISK1)	(ON DISK1 PEG3) (ON DISK2 PEG1) (ON DISK3 PEG1) (GREATER DISK2 DISK1) (GREATER DISK3 DISK2) (GREATER DISK3 DISK1)
Bindings	((*diskx* . DISK1) (*peg* . PEG1))	((*diskx* . DISK3) (*peg* . PEG1))
Inner Circle	(ON DISK1 PEG1)	(ON DISK3 PEG1)
Outer Circle	(ON DISK2 PEG1) (ON DISK3 PEG1) (GR DISK3 DISK1) (GR DISK2 DISK1)	(ON DISK2 PEG1) (GR DISK3 DISK2) (GR DISK3 DISK1)

As before, LIVE first substitutes the objects by the variables, and attempts to find differences in the relations of the inner circle. Unfortunately, no differences can be found there because (ON DISK3 PEG1) and (ON DISK1 PEG1) are the same after the variable substitutions (see the following table).

	The Success	The Failure
Inner Circle	(ON *diskx peg*)	(ON *diskx peg*)
Outer Circle	(ON DISK2 *peg*) (ON DISK3 *peg*) (GR DISK3 *diskx*) (GR DISK2 *diskx*)	(ON DISK2 *peg*) (GR *diskx* DISK2) (GR *diskx* DISK1)

LIVE is now forced to consider the objects in the outer circles. In order to find the differences in the outer circles, LIVE will search for a "chain" of relations that is true in the failed application but false in successful application. The search starts from the variables. From *peg*, LIVE finds a common element, (ON DISK2 *peg*), that is true in both states, so the first relation in the chain is found. Next, LIVE uses DISK2 as the new leading object and continues the search. (GR *diskx* DISK2) is then found in the failed state, but nothing similar can be found in the successful state. At this point, LIVE concludes that a differential chain ((ON DISK2 *peg*) (GR *diskx* DISK2)) has been found. Since the learning algorithm requires the final result to be true in the successful application and false in the failed application, LIVE returns (NOT ((ON *disky peg*) (GR *diskx disky*))), where *disky* is a generalization of DISK2.

With this differential chain, Rule6-3 and its sibling rule, Rule6-5, are modified into the following two new rules (we will call them Rule6-7 and Rule6-9 for convenience):

Rule Index: 6-7 Condition: ((ON *diskx peg*) **(NOT ((IN-HAND** *diskx*))) **(NOT ((IN-HAND** *disky*))) **(NOT ((ON** *disky peg*) **(GR** *diskx disky*)))) **Action: Move-Pick(***diskx peg*) **Prediction: ((IN-HAND** *diskx*) **(NOT ((ON** *diskx peg*)))) **Sibling: 6-9**

Rule Index: 6-9 Condition: ((ON *diskx peg*) **(NOT ((IN-HAND** *diskx*))) **(NOT ((NOT ((IN-HAND** *disky*))) **(NOT ((ON** *disky peg*) **(GR** *diskx disky*)))))) **Action: Move-Pick(***diskx peg*) **Prediction: ((ON** *diskx peg*)) **Sibling: 6-7**

Rule6-7 says that in order to pick up a disk from a peg, the HAND must be empty, as the condition (NOT ((IN-HAND *disky*))) indicates, and there must be no smaller disks on the peg, as the condition (NOT ((ON *disky peg*) (GR *diskx disky*))) represents. Otherwise, as Rule6-9 predicts, the disk will remain on the peg.

In general, search for a differential chain of relations is pairing "common" relations from both states starting with the variables. Such pairing stops when a relation in the failed

state cannot be paired with any relation in the successful state, and the relations found in the failed state will be returned. The word "common" here does not mean exactly equal. Two relations are common as long as they have the same relational predicate and the same position for the leading object. For example, if *disk* is the lead, (GR DISK3 *disk*) and (GR DISK2 *disk*) are common, but (GR *disk* DISK1) and (GR DISK3 *disk*) are not. To search for a differential chain in the following table, LIVE will start with *peg*:[6]

	The Success	The Failure
Outer Circle	(ON DISK3 *peg*) (GR *diskx* DISK1) (GR DISK3 *diskx*)	(ON DISK2 *peg*) (GR *diskx* DISK2) (GR *diskx* DISK1)

It will find that (ON DISK2 *peg*) and (ON DISK3 *peg*) are common (*peg* is the leading object). For searching for the next relation, DISK2 becomes the leading object in the failed state, and DISK3 becomes the leading object in the successful state. However, no more pairs can be found because (GR *diskx* DISK2) and (GR DISK3 *diskx*) are not considered to be common. Thus, ((ON DISK2 *peg*) (GR *diskx* DISK2)) is the differential chain, and as before it will be returned as (NOT ((ON *disky peg*) (GR *diskx disky*))).

6.3.4 Defining New Relations for Explanations

Sometimes, when the relations perceived from two states are the same but the objects' features are different, new relational predicates may be defined in order to discriminate the two states. For example, suppose objects are perceived as vectors of four features: ($objID\ f_1\ f_2\ f_3\ f_4$), and LIVE is given ">" as a constructor and has defined the following predicates in its previous learning: Suppose LIVE is now to compare the two states in the

(F1> x y) $\Longleftrightarrow$ (elt x 1) > (elt y 1); (F2> x y) $\Longleftrightarrow$ (elt x 2) > (elt y 2); (F3> x y) $\Longleftrightarrow$ (elt x 3) > (elt y 3).

following table in order to revise some faulty rule:

	The Success	The Failure
States	(A 2 3 1 6) (B 4 1 1 4) (F1> B A) (F2> A B)	(A 2 2 1 4) (B 4 1 1 4) (F1> B A) (F2> A B)

[6]In this example, LIVE starts with the variable *peg* because starting with *diskx* has failed to find any differential chain.

Based on the relations that are perceived, no relational differences can be found from the success and the failure. However, Object A's features are different in these two states. In the state of the success, Object A's 4th feature is greater than object B's 4th feature, while in the state of the failure, no such relation exists. The fact that there is no predefined relation on the 4th feature gives us a hint that such a relation should be defined in order to discriminate these two states. Just as it defines new predicates during the rule creation time (see Chapter 5), LIVE will define a new relation as follows: and return (F4> A B) as

---

(F4> x y) $\Longleftrightarrow$ (elt x 4) > (elt y 4).

---

the difference between these two states.

In general, when comparing two states that have no relational differences, LIVE will check if there are any differences between those features that have no predefined relations. If differences can be found according to the given constructors on these features, LIVE will define the new and necessary relational predicates as we outlined in the above example. Since creating new relations here is the same as defining new relations at rule creation time, we will have no further discussion here. Interested readers can see Section 5.2 for more examples and the detailed description.

6.3.5 When Over-Specific Rules are Learned

Although LIVE's discrimination method has returned the correct reasons for each of the surprises we have talked about so far, sometimes wrong reasons will be found and over-specific rules will be learned. Fortunately, such wrong rules will be detected as LIVE begins to use them in the problem solving, and will be corrected by LIVE's method of experimention. In this section, we will give an example of how a wrong rule is learned but will leave the correction of it to section 6.3 where LIVE's learning from experiments is presented.

Suppose Rule6-0 is a rule learned by LIVE while putting DISK1 on PEG2 in the state shown in Figure 6.9.

Rule Index: 6-0 Condition: ((**IN-HAND** *diskx*) (**NOT** ((**ON** *diskx peg*)))) **Action:** **Move-Put**(*diskx peg*) **Prediction:** ((**ON** *diskx peg*) (**NOT** ((**IN-HAND** *diskx*)))) **Sibling:** ()

In order to put DISK2 on PEG2 in the state shown in Figure 6.10, Rule6-0 is applied with

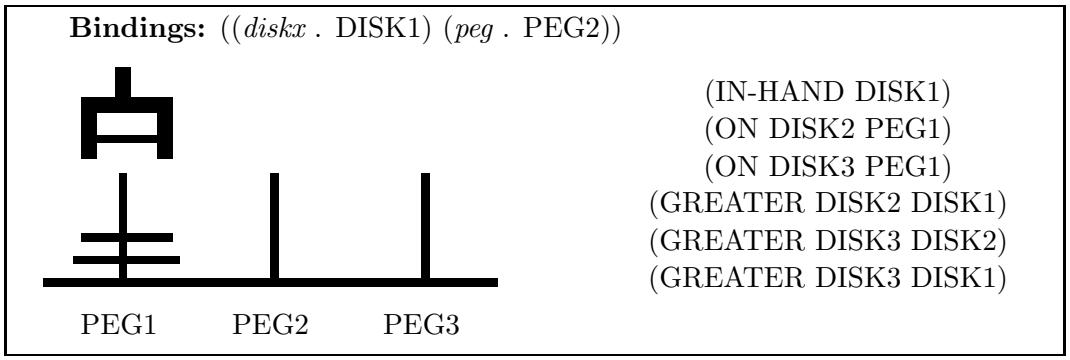


Figure 6.9: A successful application of Rule6-0.

disk bound to DISK2 and *peg* bound to PEG2. LIVE is surprised after Move-Put(DISK2 PEG2) is applied because DISK2 is not on PEG2 but is still in the hand.

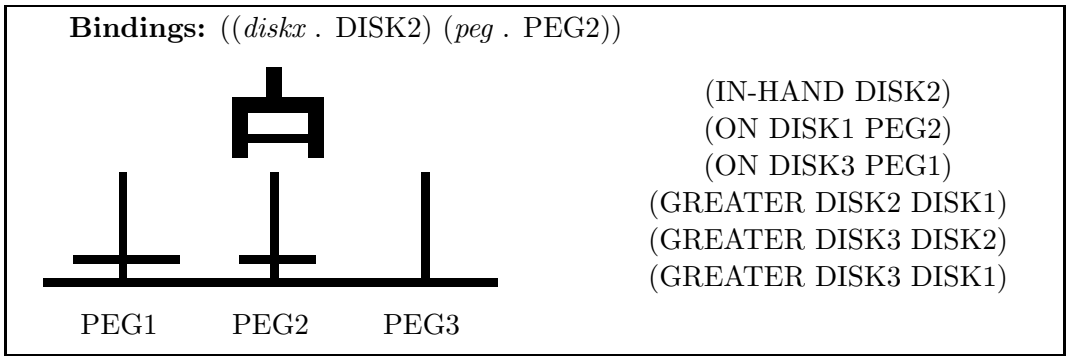


Figure 6.10: A failed application of Rule6-0.

As before, the two states are compared to find relational differences. The inner circles do not reveal any differences because in both states (IN-HAND *diskx*) is the only relation in the inner circle. However, since in the outer circle of the successful state, ((GR DISK2 *diskx*)(GR DISK3 *diskx*)), is different from the outer circle of the failed state, ((ON DISK1 *peg*) (GR DISK2 *diskx*)(GR DISK3 *diskx*)), LIVE finds the difference to be (NOT ((ON *disky peg*))) and splits Rule6-0 into Rule6-2 and Rule6-4.

Rule Index: 6-2 Condition: ((IN-HAND *diskx*) **(NOT ((ON** *diskx peg*))) **(NOT ((ON** *disky peg*)))) **Action: Move-Put(***diskx peg***) Prediction: ((ON** *diskx peg*) **(NOT ((IN-HAND** *diskx*)))) **Sibling: 6-4**

Rule Index: 6-4 Condition: ((IN-HAND *diskx*) **(NOT ((ON** *diskx peg*))) **(ON** *disky peg*)) **Action: Move-Put(***diskx peg***) Prediction: ((IN-HAND** *diskx*) **(NOT ((ON** *diskx peg*)))) **Sibling: 6-2**

However, Rule6-2 is a wrong rule. It says that in order to put a disk on a peg, that peg must be empty. According to the real rules of Tower of Hanoi, this rule is over-specific because it prevents LIVE from putting the smaller disks on the bigger ones.

In previous methods of learning by discrimination, as Bundy *et. al.* pointed out in [8], when situations like this happen, it is very hard to recover. Fortunately, LIVE's learning will not suffer fatally from this kind of error because the over-specific rule has a over-general sibling. As we will see in the next section, LIVE is able to detect such wrong rules and correct them by experimentation.

6.4 Experiments: Seeking Surprises

Up to now, we have only said that faulty rules are detected during execution, and that they are corrected based on the information derived from surprises. In fact, faulty rules, especially those that are overly specific, can also be detected during planning. In that case we need to seek surprises in order to correct them.

6.4.1 Detecting Faulty Rules during Planning

LIVE can detect two kinds of faulty rules at planning time: the rules that cause given goals to become impossible to reach (called *goal-contradiction*), and the rules that repeatly propose subgoals that are already in the goal stack (called *subgoal-loop*). We will give two examples, one demonstrating each kind of faulty rule to explain how the errors are detected.

Goal contradiction is an error wherein no plan can be proposed because subgoals destroy each other no matter how they are arranged. As an example, suppose LIVE is given the following goals to reach:

((ON DISK1 PEG3) (ON DISK2 PEG3) (ON DISK3 PEG3))

and it has Rule6-2 (reprinted below) learned previously, which says that in order to put *diskx* on *peg* successfully, *peg* must be empty (i.e. no *disky* can be on *peg*):

As we described in Section 4.4, to construct a solution for the given goals, LIVE first selects a rule for each of the given goals. It then orders the goals according to their corre-

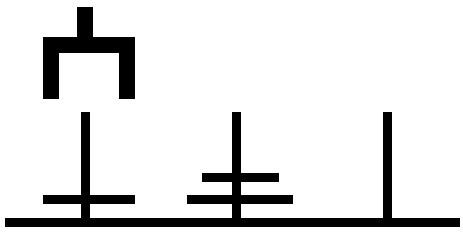
Rule Index: 6-2 Condition: ((IN-HAND *diskx*) **(NOT ((ON** *diskx peg*))) **(NOT ((ON** *disky peg*)))) **Action: Move-Put(***diskx peg***) Prediction: ((ON** *diskx peg*) **(NOT ((IN-HAND** *diskx*)))) **Sibling: 6-4**

sponding rules so that goals achieved earlier in the sequence will not violate the conditions for the goals to be achieved later. Suppose Rule6-2 is chosen for all the goals in this example (of course with different bindings). When LIVE tries to order these goals, it discovers that no matter how these goals are arranged, they destroy each other. For example, assume (ON DISK3 PEG3) is achieved first; in order to achieve (ON DISK2 PEG3), the condition of its rule requires (NOT ((ON *disky* PEG3))) to be satisfied. This condition is violated by the achieved goal (ON DISK3 PEG3) because the free variable *disky* can be bound to DISK3. At this point, LIVE reports a goal-contradiction error and returns the rule as an erroneous rule to be fixed.

Unlike goal-contradiction, subgoal-looping is an error whereby planning goes into an infinite loop. Suppose LIVE has learned a rule Rule-M as follows: and wants to pick up

Rule Index: M **Condition: ((ON** *diskx peg*) **(NOT ((IN-HAND** *diskx*))) **(NOT ((ON** *disky peg*)))) **Action: Move-Pick(***diskx peg***) Prediction: ((IN-HAND** *diskx*) **(NOT ((ON** *diskx peg*))) **Sibling:** N

DISK3 in the following state:



(ON DISK1 PEG2)
(ON DISK2 PEG1)
(ON DISK3 PEG2)
(GREATER DISK2 DISK1)
(GREATER DISK3 DISK2)
(GREATER DISK3 DISK1)

Since Rule-M predicts (INHAND *diskx*) , it is chosen to accomplish (INHAND DISK3) with *diskx* bound to DISK3 and *peg* bound to PEG2. However, since one of its conditions, (NOT ((*disky peg*))), is not true in the current state, the rule is not immediately applicable. (The reason that (NOT ((*disky peg*))) is false in the state is that *peg* is bound to PEG2 where DISK3 is and *disky* is bound to DISK1 by the matcher because it is on PEG2.)

So LIVE proposes (NOT ((DISK1 PEG2))) as a new subgoal to achieve and continues its planning. To achieve this new goal, Rule-M is again chosen because it predicts (NOT ((DISK1 PEG2))) with bindings (($diskx$. DISK1) (peg . PEG2)). Unfortunately, in order to apply this new rule instance in the current state, subgoal (NOT ((ON DISK3 PEG2))) is proposed again for the same reason explained above. To achieve this subgoal, Rule-M is chosen again and the subgoal (NOT ((DISK1 PEG2)) is proposed again. At this point, we loop back to where we started (see the goal stack below). Clearly, such a subgoaling loop will iterate forever if not detected and stopped.

The Goal Stack	The Bindings
...	
(NOT ((DISK3 PEG2)))	(($diskx$.DISK3)(peg.PEG2)($disky$.DISK1))
(NOT ((DISK1 PEG2)))	(($diskx$.DISK1)(peg.PEG2)($disky$.DISK3))
(NOT ((DISK3 PEG2)))	(($diskx$.DISK3)(peg.PEG2)($disky$.DISK1))
(NOT ((DISK1 PEG2)))	(($diskx$.DISK1)(peg.PEG2)($disky$.DISK3))
(INHAND DISK3)	(($diskx$.DISK3)(peg.PEG2)($disky$.DISK1))
...	

In general, LIVE detects subgoal loop errors by comparing a newly proposed subgoal to the goals that are already on the goal stack. Suppose a new subgoal G_0 is proposed and the existing goals are $(G_1, \ldots G_i, G_{i+1}, \ldots)$. LIVE will report a subgoal-loop error as soon as it finds that $G_0 = G_i$ and $G_1 = G_{i+1}$. The rule that is responsible for proposing these subgoals will be returned as the faulty rule.

6.4.2 What Is an Experiment

Unlike fault rules detected at the time of a surprise, faulty rules detected during planning do not come with information about how to fix them. In order to find out why they are wrong, LIVE must design and perform experiments with the hope that surprises will occur in the experiments.

Before we present LIVE's experiments, let us first examine one such faulty rule carefully to get some hints on how we should pursue the matter. Recall that Rule6-2 causes a goal contradiction. The rule says that in order to put a disk on a peg, the target peg must be empty. We assume that LIVE's goals are not impossible to reach and that the rule has been proved wrong by the goal contradiction error. We can now draw the conclusion that there must be situations in which a disk *can* be put on a non-empty peg. The reason that we

need experiments is that we must find out more about such situations, although we already know that in this case the target peg should be non-empty. Such a conclusion tells us that not only is Rule6-2 wrong, but that its sibling, Rule6-4, is also wrong because the sibling says that if a peg is not empty, no disk can be put on it.

Since our primary learning method is learning from surprises, experiments must have predictions based on our knowledge; without predictions LIVE will never be surprised. By putting all these together, three constraints for experiment design have become clear. First, we want to be in a situation where the target peg is not empty; second, we want to do Move-Put to put a disk on that target peg; third, we must have predictions because we want to learn from any surprises that might occur after the action.

Given these constraints, we find Rule6-4, the sibling of the faulty rule Rule6-2, is a good candidate for the experiment. This is because (1) Rule6-4 is applicable in the states where the target peg is not empty (while Rule6-2 is not); (2) Rule6-4 has the same action as Rule6-2 so the desired action Move-Put will be performed; (3) predictions can be made because Rule6-4 is applicable in those "target peg not empty" situations and its prediction reflects LIVE's existing knowledge.

Therefore, an *experiment* in LIVE is nothing but an instantiation of the faulty rule's sibling rule. Its condition specifies a situation in which LIVE must put itself, and its action specifies an action that LIVE must perform once it is in that situation. LIVE carries out an experiment as if it were a new problem to be solved. In particular, the problem solver will make and execute a plan to achieve the experiment's condition. Once the condition is satisfied, the problem solver will perform the required action and observe the outcome. An experiment succeeds if the outcome of its action produces a surprise, i.e. violates the experiment's prediction. In that case, the explanation derived from the surprise will be used to revise the faulty rule and its sibling. If the experiment's action does not produce any surprises, then LIVE will propose another experiment using the same rule instantiated differently.

Reconsider our goal-contradiction example. Once Rule6-2 is identified as the faulty rule, LIVE will instantiate Rule6-4, the sibling of Rule6-2, to construct an experiment:

Since LIVE is currently in the state of Figure 6.10 where Rule6-2 and Rule6-4 are created and the goal contradiction error is detected, the condition of this experiment is satisfied, and the action can be performed immediately. The experiment is successful because its

Experiment: For Rule6-4 Condition: ((IN-HAND DISK2) (NOT ((ON DISK2 PEG1))) (ON DISK3 PEG1)) Action: Move-Put(DISK2 PEG1) Prediction: ((IN-HAND DISK2) (NOT ((ON DISK2 PEG1)))) Sibling: Rule6-2 Bindings: ((*diskx* . DISK2) (*peg* . PEG1) (*disky* . DISK3))

outcome, (ON DISK2 PEG1), surprises the prediction (NOT ((ON DISK2 PEG1))).

Surprises in experiments are explained in the same way as described in Section 6.1. In this example, LIVE will compare the the success, Figure 6.10 with bindings ((*diskx* . DISK2) (*peg* . PEG2) (*disky* . DISK1)) with the failure, Figure 6.10 with bindings ((*diskx* . DISK2) (*peg* . PEG1) (*disky* . DISK3)). These two applications are in the same state but have different bindings. The comparison yields the explanation ((ON *disky peg*)(GR *diskx disky*)) which is true in the successful case but false in the failed case. Thus, Rule6-4 is revised into the new Rule6-6, and its sibling rule Rule6-2 is revised into the new Rule6-8:

Rule Index: 6-6 Condition: ((IN-HAND *diskx*) **(NOT ((ON** *diskx peg*))) **(ON** *disky peg*) **(GR** *diskx disky*)) **Action: Move-Put(***diskx peg*) **Prediction: ((IN-HAND** *diskx*) **(NOT ((ON** *diskx peg*)))) **Sibling: 6-8**

Rule Index: 6-8 Condition: ((IN-HAND *diskx*) **(NOT ((ON** *diskx peg*))) **(NOT ((ON** *disky peg*) **(GR** *diskx disky*))) **Action: Move-Put(***diskx peg*) **Prediction: ((ON** *diskx peg*) **(NOT ((IN-HAND** *diskx*)))) **Sibling: 6-6**

The new rule Rule6-8 says that in order to put a disk (*diskx*) on a peg (*peg*), the peg should not contain any smaller disks (*disky*). After the experiment, LIVE has learned the correct rules about when a disk can be put on a peg and when cannot. In a similar way, although details will not be given here, the rules Rule$M$ and Rule$N$ which cause subgoal-loop errors will also be corrected by LIVE's learning from experiments.

6.4.3 Experiment Design and Execution

From the above example and the analysis, we can see that LIVE's experimentation method, which is founded on the rule representation and the complementary discrimination learning method, is quite simple. When an incorrect rule is detected, LIVE first retrieves the sibling of that rule. Then it instantiates the sibling to seek surprises as easily and as quickly as possible. The instantiation is implemented as a set of heuristics that will be described shortly.

Once the instantiation is completed, an experiment is designed, and it is the problem solver's job to bring LIVE into the state specified by the conditions of the experiment. Finally, the experiment's action will be executed in that state, and the outcome will be compared to the experiment's prediction. If the experiment succeeds, learning will be triggered as if LIVE had met a normal surprise in its plan execution; otherwise, a new instantiation from the same sibling rule will be generated, and LIVE will continue its experiments.

From the description of experiments, it is not hard to see what makes a good experiment. First, since the purpose of an experiment is to seek surprises for a given rule, an experiment is better if its action is instantiated in such a way that it is likely to produce surprises. Second, since the condition of an experiment must be satisfied before the real experiment can begin and such preparation involves both planning and execution, an experiment should have a condition that is easily satisfied from the current state. Ideally, it could be already satisfied. These two criteria give us the following procedure which is used by LIVE's experiment module:

- Given a rule, experiments are instantiated from the rule by alternately assigning different objects to the parameters of the action. For instance, if the experiment rule's action is Move-Pick(*disk*, *peg*), then different disks will be assigned to the variable *disk*, while different pegs will be assigned to *peg*. Such assignments are guided by the following heuristics:

- An experiment's action must be different from all the actions that have been done successfully by its sibling rule (because such actions will not produce any surprises) and all the actions that were performed in previous experiments instantiated from the same rule that did not produce any surprises.

- An experiment is preferred if its condition is true in the current state or can be easily reached from the current state.

6.4.4 Related Work on Learning from Experiments

In our theory of learning from environments (Chapter 3), experiment design is equivalent to the learner selecting its own training instances. An early system of this kind is Mitchell *et. al.*'s LEX [32] which generates problems autonomously with two heuristics. One heuristic produces problems that will allow refinement of existing, partially-learned rules, and the

other creates problems that will lead to proposing new rules. Although their underlying rule representation is very different from ours, their heuristics are very similar to what we have used in LIVE's experiment module and exploration module.

Carbonell and Gil [9] have also studied learning from experiments. In their program, when an operator does not produce the results that are expected, the program will apply the operator to different objects to find the missing preconditions. Kulkarni and Simon [**?**] have focused on the strategy of experimentation in scientific discovery; their work has identified a set of heuristics for autonomously collecting data for discovering scientific laws.

6.5 Comparison with Some Previous Rule Learning Methods

We have already seen some examples that Complementary Discrimination differs from Version Spaces that a generalization is accomplished by discriminating a concept's complement. This provides an advantage that no initial concept hierarchy is necessary. If we view the initial concept hierarchy as a bias for learning [**?**] (for example, the Version Space in Figure 6.2 cannot learn $(y = cr) \vee (y = sq)$ because the concept is not included in the initial concept hierarchy), then complementary discrimination does not commit to any bias aside from the concept description language at the outset. This property plus the fact that the method is data-driven suggests that it can construct new terms if the current language is inadequate for describing the whole phenomena of the environment (as we will see in Chapter 7). It should also be able to deal with noisy data to some extent because the method will not throw concepts away when training instances are not consistent, although we have not tested LIVE in any noisy environment in this thesis. As for the time needed for LIVE to converge to the correct concept, it seems that complementary discrimination will do no worse than Version Spaces as demonstracted by the example we just showed. (Formal proofs of the claimes made here will be future work.)

From the examples of rule revision, we also saw that Complementary Discrimination extends Brazdil's and Langley's pure discrimination methods because it can recover from over-specific errors and to learn disjunctive rules without depending solely on far misses. Compared to Decision Trees [38] and ID3 learning algorithm, one major difference is that Complementary Discrimination is an incremental method and it splits or revises a concept according to the differences found in the comparison between failures and successes. Thus the revision is based on an expression of multiple features, instead of any particular single

feature as in a decision tree.

Compared to other one-way generalization learning methods such as Winston [60], our method utilizes both generalization and specialization and uses counterexamples is a constructive way. Vere's excellent paper [58] is the most closely related work to complementary discrimination. His "counterfactual" is very similar to our learning from mistakes. We share the same spirit, but he emphasizes the aspect of generalization while we emphasize the aspect of discrimination.

Although the final concepts are somewhat like a decision tree, LIVE's method is incremental and does not require all the training instances be given before learning. This important characteristic makes it possible for LIVE to interleave its learning with problem solving.

6.6 Discussion

Note that in all the examples we have given so far, LIVE revises only the conditions of the rules and not the predictions, as does Carbonell and Gil's work [9]. However, if we share their assumption that all the differences caused by an action can be seen and remembered even if they are not used in the rules, then LIVE can handle their telescope task easily. For example, as we pointed out in Chapter 5, the missing prediction ¬(IS-CLEAN *obj*) in their initial Operator-3 will be included by LIVE at rule creation time because there is a condition (IS-CLEAN *obj*) which has been changed by the action ALUMINIZE. Similarly, the missing predictions ¬(IS-POLISHED *obj*) and ¬(IS-REFLECTIVE *obj*) in their initial Operator-1 will also be included if the action GRIND-CONCAVE is applied to a POLISHED and REFLECTIVE object for the first time. Suppose it is not, then in the future LIVE will notice that the action GRIND-CONCAVE produces more changes than predicted and will remember these extra changes. If these changes violate the conditions of some later rules in the plan execution, LIVE will then insert the changes into the rule's prediction.

In the examples of detecting faulty rules during planning, LIVE was lucky in that only one faulty rule was found at a time. If more than one rule is found, LIVE will face a credit-assignment problem. At present, we have not implemented any mechanisms to dealt with this difficult problem, but any good method for credit-assignment, such as Shapiro's *Contradiction Backtracking* [41], can be employed here.

There is one potential difficulty in LIVE's learning from experiments method. Experi-

ments may be beyond the problem solving ability of LIVE's existing knowledge. For example, during the execution of an experiment for correcting a faulty rule, another faulty rule may be detected or the same faulty rule may be required in order to accomplish the experiment. Although LIVE can make sure that unsuccessful experiments are not repeated, no general solution for this problem has been identified except totally giving up on the current experiment and exploring the environment further to learn better rules.

Although complementary discrimination is best for learning a single concept and its complement, the method can also be used to learn multiple concepts if the concept description language has the predicates that can divide the set of concepts correctly. For example, in order to learn the rules that predict the result of releasing the balance beam [46], LIVE must have a predicate that can divide *balance* as one set and *left* and *right* as the other. We have also investigated a method that always splits the faulty rule even if the rule has siblings already [**?**]. This method learns multiple concepts, but it is not as complete as complementary discrimination in general.

In principle, complementary discrimination has the potential to deal with noisy data because it is incremental and it does not throw away concepts even if the training instances are not consistent. For example, if (lg sq) is a positive instance at time t but becomes a negtive instance at time $t + i$, complementary discrimination should be able to recover from the earlier commitment by revising the complement concept. Although we have not conducted any tests in this thesis, we would like to point out a philosophical difficulty here, which we believe applies to any learning method for noisy data, including the methods based on statistics. The problem is that when errors are made, it is very difficult to decide whether the learner should believe in itself or the environment.[7] Although there are only two possibilities, either the learning data are bad or the existing knowledge is wrong; it is extremely difficult to distinguish one from the other. A naive solution would be to remember all of the instances when the learner has chosen to believe in itself which means refusing to adapt itself to the environment's feedback. Then when it discovers it was really wrong, the history can be used to do the adaptation. This method is very expensive, and we do not know any existing learning programs that can do the task smoothly or flawlessly.

[7] As we pointed out in our assumptions in Chapter 3, this is the question of whether we believe the world is deterministic or non-deterministic.

Chapter 7

Discovering Hidden Features in the Environment

This chapter extends Complementary Discrimination to discover hidden features from environments. We will first outline the extension, then illustrate it by two examples: one for discovering hidden features that are new terms constructable in the concept language; the other for discovering hidden features that are not in the concept language but in the combination of the action language and the concept language. We will also relate the hidden features to the theoretical terms long studied in the philosophy of science and point out some interesting aspects of the theoretical terms that have not been studied before. Finally, we will compare our discovery method with some previous discovering programs, such as AM [27], BACON [25], ARE [43], and STABB [**?**].

7.1 What Are Hidden Features?

Previous research in discrimination learning has developed many methods for finding the critical difference between two states, but what if the two states have no difference at all, as when two pairs of green peas look exactly the same but produce different offspring? In this case, we say that the environment has *hidden features*, something unobservable that yet can discriminate two states that appear identical.

Hidden features are features unobservable by the learner but necessary for predicting the outcomes of actions. For example, suppose a learner has an action Push(*obj*) and can perceive only objects' shapes, volumes, densities and locations. If *weight* is the feature that determines whether an object is pushable or not, then the weight feature is an example of

a hidden feature.

From a language point of view, hidden features are commonly called *new terms* in Artificial Intelligence. They are terms that are missing in the initial concept description language but necessary for describing the target concepts. Since terms can be either variables and constants or defined by functions on other terms (see the definition of term in Section 4.3.1), hidden features can be further divided into two different kinds with respect to a particular concept language C: those that can be defined as $f_i^n(t_1, \ldots, t_n)$ (where f_i^n is a function given in C and $t_1, \ldots, t_n$ are existing terms in C); and those that are variables and constants themselves and cannot be defined in terms of existing terms and functions unless actions are considered or temporal concepts are introduced. In our current example, if the function "$*$" is known, then *weight* is a definable new term because $weight = volume * density$, otherwise, weight cannot be defined except by executing Push(*obj*) (here we assume the action is not in the language C). A more natural example of undefinable terms are the genes discussed in Section 7.4.

The problem of hidden features has been studied from different aspects with different reasons. For example, all the new concepts defined by AM [27] and ARE [43] can be regarded as hidden features in general but they are defined merely because they are "interesting". BACON [25] defines hidden features for constructing laws that explain a set of given data. Utgoff studied hidden features as a problem of bias [**?**], and proposed a framework for shifting bias in the process of learning.

7.2 How LIVE Discovers Hidden Features

Based on complementary discrimination, LIVE discovers hidden features in three steps:

1. Detect the existence of hidden features by noticing the failure of a discrimination.
2. Identify hidden features that enable LIVE to discriminate further.
3. Assimilate the hidden features into the existing language and continue the discrimination process.

The first step is to decide when discovery of hidden feature is necessary. Since LIVE learns through discrimination, it is straightforward to see the solution for this step. That is, if the discrimination process cannot find any difference between the current failure and

previous successes, then it is necessary to discover hidden features. When such situations arise, LIVE will return to the next step with the failure and the success (each with a state and a set of bindings) that are indistinguishable in the current language.

The task of the second step, when given the two indistinguishable cases, is to search for a set of new terms that can be used to distinguish the two cases. As we said earlier, such terms can either be defined by existing terms and functions, or involve actions. Thus, discovering hidden features can be done in two different ways, depending on whether an environment is time dependent or not. In a time-independent environment, where states do not depend on the previous actions, LIVE discovers hidden features by applying its constructor functions to the existing features and testing whether the result discriminates the ambiguous states. For example, when predicting whether a balance-beam [46], see Section 7.3, will tip or balance, LIVE discovers the invisible "torque" concept by multiplying distance and weight. In time-dependent environments, where states do depend on the previous actions, LIVE discovers hidden features by searching back in the history to find differences in the states preceding the two indistinguishable states. Details of such search will be given in Section 7.4 when LIVE attempts to discover genes.

Identifying the hidden features does not complete the whole discovery process; these features must be assimilated into the system to be useful in the future. In a time-independent environment, since hidden features are defined in terms of observables, the system can simply use the newly defined features as if they were visible because they are computable from the observables. For example, when the concept of torque is discovered by a system that perceives only objects' distances and weights, the concept will simply be added as another object feature. Any rule that needs torque to discriminate its condition can use it by computing the value $weight{*}distance$. In a time-dependent environment, since hidden features determine the observables, two additional things must be done before these features can be used: determine how the hidden features define the observables; and determine how the hidden features are inherited through actions. One strategy is used for both tasks: testing all the constructor functions to find one that is consistent with all the examples collected (this is a BACON like task).

7.3 Constructing Hidden Features with Existing Functions and Terms

This section gives a detailed example of how LIVE discovers hidden features in a time-independent environment. The Balance Beam, see Figure 7.1, is a task for research in child development. It was first studied by Inhelder and Piaget [18]. A set of production rules used by children from age 2 to 15 was hypothesized by Siegler [46] to demonstrate how knowledge influences learning. Flinn and Newell [35] used the Balance Beam as a test-bed for Soar's cognitive theory. In the experiment, children are shown a see-saw like beam with pegs on both sides on which weights can be placed. The task is to predict, after the beam is released, whether the beam will tip to the left, to the right, or balance. From our point of view, this experiment can be defined as a problem of learning from the environment as shown in Figure 7.2. One simplification is made in the definition: weights on one side are always put on a single peg, so that each side of the beam can be viewed as a single object with three features: *weight, distance,* and *altitude*. Note that in addition to the relations concern about weight, distance and altitude, we assume the learner has been given two primitive relations, $=$ and $>$, and two simple functions, $+$ and $*$.

Figure 7.1: The Balance Beam experiment.

In our experiment, LIVE is given a sequence of tasks (see Figure 7.3) that are taken directly from Siegler's paper [46]. When the first task is given, LIVE observes the state ((L 3 1 0) (R 3 1 0) (W= L R) (D= L R) (A= L R)), but has nothing to predict because its internal rule base is empty (recall from Chapter 5, at this point LIVE's problem solver will report a no-rule error). So it decides to explore the action Release(L, R) (this can be thought as a single-action explorative plan). After observing that nothing is changed by the action, LIVE creates the first and the most general rule from this environment. (Note that

- **Percepts** Two objects L and R, each of which is a feature vector *(objID weight distance altitude)*; six relational predicates: W= (weight=), W> (weight>), D= (distance=), D> (distance>), A= (altitude=), and A> (altitude>);
- **Actions** Release($obj_x\ obj_y$);
- **Constructors** Two primitive relations, = and >, and two functions, + and $*$.
- **Goal** To predict A= or A> after the action.

Figure 7.2: Learning from the Balance Beam environment.

Figure 7.3: A sequence of Beam tasks, copied from (Siegler 83).

because the goal is to predict A= or A>, the rule's prediction has included the relation A= even though nothing has changed in this task.)

Rule Index: 1 Condition: (($obj_x\ w_x\ d_x\ a_x$) ($obj_y\ w_y\ d_y\ a_y$) (A= $obj_x\ obj_y$)) Action: Release($obj_x\ obj_y$) Prediction: ((A= $obj_x\ obj_y$)) Sibling: ()

In the second task, because Rule1's condition is satisfied in the state ((L 3 1 0) (R 2 1 0) (W> L R) (D= L R) (A= L R)), LIVE predicts (A= L R). Unfortunately, the prediction is falsified by the result: (A> R L). By comparing to the first task, a reason is found for the surprise: ((W= L R) ¬(W> L R)) which was true in the first task but false in the current

situation. Thus the rule is split into the following two (see Section 6.2.1 for the algorithm):

Rule Index: 1 Condition: ((*obj*$_x$ w_x d_x a_x) (*obj*$_y$ w_y d_y a_y) (**A=** *obj*$_x$ *obj*$_y$) (**W=** *obj*$_x$ *obj*$_y$) ¬(**W>** *obj*$_x$ *obj*$_y$)) **Action: Release**(*obj*$_x$ *obj*$_y$) **Prediction:** ((**A=** *obj*$_x$ *obj*$_y$)) **Sibling: 2**

Rule Index: 2 Condition: ((*obj*$_x$ w_x d_x a_x) (*obj*$_y$ w_y d_y a_y) (**A=** *obj*$_x$ *obj*$_y$) ¬((**W=** *obj*$_x$ *obj*$_y$) ¬(**W>** *obj*$_x$ *obj*$_y$))) **Action: Release**(*obj*$_x$ *obj*$_y$) **Prediction:** ((**A>** *obj*$_y$ *obj*$_x$)) **Sibling: 1**

With these two rules, LIVE has learned to pay attention to the objects' weight feature. When working on the third task, ((L 3 3 0) (R 3 2 0) (W= L R) (D> L R) (A= L R)), Rule1 is fired because L and R has the same weight. Again, LIVE's prediction (A= L R) is wrong because the beam tips to the left: (A> R L). After comparing the current state to the first task and finding the explanation: ((D= L R) ¬(D> L R)), Rule1 is modified into the following new Rule1, and Rule2 is modified to be the complement of the new Rule1 (see Section 6.2.1 for the algorithm):

Rule Index: 1 Condition: ((*obj*$_x$ w_x d_x a_x) (*obj*$_y$ w_y d_y a_y) (**A=** *obj*$_x$ *obj*$_y$) (**W=** *obj*$_x$ *obj*$_y$) ¬(**W>** *obj*$_x$ *obj*$_y$) (**D=** *obj*$_x$ *obj*$_y$) ¬(**D>** *obj*$_x$ *obj*$_y$)) **Action: Release**(*obj*$_x$ *obj*$_y$) **Prediction:** ((**A=** *obj*$_x$ *obj*$_y$)) **Sibling: 2**

Rule Index: 2 Condition: ((*obj*$_x$ w_x d_x a_x) (*obj*$_y$ w_y d_y a_y) (**A=** *obj*$_x$ *obj*$_y$) ¬((**W=** *obj*$_x$ *obj*$_y$) ¬(**W>** *obj*$_x$ *obj*$_y$) (**D=** *obj*$_x$ *obj*$_y$) ¬(**D>** *obj*$_x$ *obj*$_y$))) **Action: Release**(*obj*$_x$ *obj*$_y$) **Prediction:** ((**A>** *obj*$_y$ *obj*$_x$)) **Sibling: 1**

With these two new rules, LIVE will predict the beam to be balance when both sides have the same weight and distance. However, LIVE believes that weight and distance are independent for determining which side will tip. Rule2 says that the side that has either greater weight or greater distance will tip down.

While the fourth task comes along, Rule2 is fired because in the state ((L 3 3 0) (R 2 4 0) (W> L R) (D> R L) (A= L R)) both weight and distance are different. Since the condition W¿ precedes D¿, L will be bound to *obj*$_x$ and R will be bound to *obj*$_y$ so LIVE predicts (A> R L). This time, the prediction is correct and no rule is changed.

So far, learning by discrimination has been working smoothly. However, trouble begins now. In the fifth task, Rule2 is fired again but with obj_x bound to R and obj_y bound to L. Because (W> R L), LIVE predicts (A> L R). As usual, after the prediction is falsified by the result (A> R L), LIVE begins to compare the current application with the Rule2's last successful application, task 4. Comparing these two applications, LIVE cannot find any

---

Task 4 State: ((L 3 3 0) (R 2 4 0) (W> L R) (D> R L) (A= L R)). Task 4 Bindings: ((obj_x . L) (obj_y . R))
Task 5 State: ((L 2 3 0) (R 4 1 0) (W> R L) (D> L R) (A= L R)). Task 5 Bindings: ((obj_x . R) (obj_y . L))

---

relational difference because (W> obj_y obj_x), (D> obj_x obj_y) and (A= obj_x obj_y) are all true in both cases.

At this point, LIVE is convinced that hidden features must exist in this environment, and decides to use its constructor functions, + and $*$, to search for a new feature that can discriminate these two look-alike states. Since each object has three features w, d and a, there are six possible hidden features: $w+d$, $w*d$, $d+a$, $d*a$, $d+a$, and $d*a$. LIVE finds that $w*d$ is the only one that can distinguish the two states because in task 4, obj_x's $w*d$ (3*3) is greater then obj_y's $w*d$ (2*4), while in task 5, obj_x's (4*1) is less than obj_y's (2*3). LIVE concludes that ($w*d$) is the hidden feature it is looking for. Since this new feature is defined in terms of existing functions and features, it is added as the 4th feature for the objects in this environment, and a new relation REL787 is defined with respect to the greater function > and the index 4 of the new feature (see Section 5.2.1 for details): Thus, Rule2 is then

---

REL787($objxobjy$): (the meaningful name is TORQUE¿.) **TRUE $\Longleftrightarrow$ (elt objx 4) ¿ (elt objy 4).**

---

split according to the relational difference made possible by the new feature and its relation (Rule1's condition is modified to become the complement of Rule2's condition):

Rule Index: 1 Condition: ((obj_x w_x d_x a_x w_x*d_x) (obj_y w_y d_y a_y w_y*d_y) **(A=** obj_x obj_y**)** ¬(¬**((W=** obj_x obj_y**)** ¬**(W>** obj_x obj_y**) (D=** obj_x obj_y**)** ¬**(D>** obj_x obj_y**)) (REL787** obj_x obj_y**))) Action: Release(**obj_x obj_y**) Prediction: ((A=** obj_x obj_y**)) Sibling: 2**

Note that the last condition for Rule1 is a disjunctive expression, that is, it is equivalent to ((W= obj_x obj_y) ¬(W> obj_x obj_y) (D= obj_x obj_y) ¬(D> obj_x obj_y)) OR ¬(REL787

Rule Index: 2 Condition: ((*obj*$_x$ w_x d_x a_x w_x*d_x) (*obj*$_y$ w_y d_y a_y w_y*d_y) (**A=** *obj*$_x$ *obj*$_y$) ¬((**W=** *obj*$_x$ *obj*$_y$) ¬(**W>** *obj*$_x$ *obj*$_y$) (**D=** *obj*$_x$ *obj*$_y$) ¬(**D>** *obj*$_x$ *obj*$_y$)) (**REL787** *obj*$_x$ *obj*$_y$)) **Action: Release**(*obj*$_x$ *obj*$_y$) **Prediction:** ((**A>** *obj*$_y$ *obj*$_x$)) **Sibling: 1**

obj$_x$ *obj*$_y$). Therefore, when the the sixth task comes, Rule1's condition is true in the state ((L 2 2 0 4) (R 4 1 0 4) (W> R L) (D> L R) (A= L R)) because ¬(REL787 *obj*$_x$ *obj*$_y$) is true. So LIVE predicts (A= L R) and the prediction succeeds. Up to this point, LIVE has learned the correct rules to predict the behaviors of the beam in this environment.

In this example, the hidden feature is the concept of *torque* and it is defined by (*weight*$*$ *distance*). We have seen that the discovery of this feature is triggered by the failure of the normal discrimination process, i.e. when no difference can be found between a failure and a success. Since the balance beam environment is time-independent (the tasks are independent to each other), LIVE searches for the hidden feature by applying all its constructors to the existing features and testing whether the result can discriminate the indistinguishable states.

7.4 Constructing Hidden Features with Both Percepts and Actions

In the Balance Beam environment, the hidden feature w*d, although defined in the process of learning, is already inside the concept language because the function $*$ and the features weight and distance exist in the language. In fact, this is the kind of hidden feature that has been studied the most by the previous discovery systems and constructive induction programs. But sometimes, hidden features can be more "hidden" than that, and to discover them, one must use not only the system's percepts (the concept description language) but also its actions.

In this section, we will describe how the hidden features that involve both percepts and actions are discovered. In particular, we will look at the task of discovering genes by breeding garden peas as Mendel reported in his classic paper [28].

Mendel's experiments starts with a set of purebred peas. All their ancestors are known to have the same characteristics. The peas are divided into different classes according to their observable features. For example, one class of peas may be green and have wrinkles, and another class may be yellow and have long stems. The experiments are very well

controlled so that the characteristics of hybridized peas are merely determined by their parents, not by any other factors such as weather, temperature, etc. (we shall refer to this as the *relevancy assumption* later). We have summarized his experiments in Figure 7.4, with the simplifications that a pea's color is the only observable feature and if two peas are fertilized then they will produce exactly four children (i.e. excluding the statistics).

Let $AF(pea_x\ pea_y)$ stands for the action Artificial Fertilization.
Let a pea be denoted as $(P_i\ color)$, where P_i is the pea's identifier, and *color* can be either 1, for green, or 0, for yellow).

1st Generation	$(P_1\ 0), (P_2\ 0), (P_3\ 1), (P_4\ 1), (P_5\ 0), (P_6\ 1), (P_7\ 1), (P_8\ 0).$
2nd Generation	$AF((P_1\ 0), (P_2\ 0)) \Longrightarrow \{(P_9\ 0), (P_{10}\ 0), (P_{11}\ 0), (P_{12}\ 0)\}$ $AF((P_3\ 1), (P_4\ 1)) \Longrightarrow \{(P_{13}\ 1), (P_{14}\ 1), (P_{15}\ 1), (P_{16}\ 1)\}$ $AF((P_5\ 0), (P_6\ 1)) \Longrightarrow \{(P_{17}\ 1), (P_{18}\ 1), (P_{19}\ 1), (P_{20}\ 1)\}$ $AF((P_7\ 1), (P_8\ 0)) \Longrightarrow \{(P_{21}\ 1), (P_{22}\ 1), (P_{23}\ 1), (P_{24}\ 1)\}$
3rd Generation	$AF((P_9\ 0), (P_{10}\ 0)) \Longrightarrow \{(P_{25}\ 0), (P_{26}\ 0), (P_{27}\ 0), (P_{28}\ 0)\}$ $AF((P_{13}\ 1), (P_{14}\ 1)) \Longrightarrow \{(P_{29}\ 1), (P_{30}\ 1), (P_{31}\ 1), (P_{32}\ 1)\}$ $AF((P_{17}\ 1), (P_{18}\ 1)) \Longrightarrow \{(P_{33}\ 0), (P_{34}\ 1), (P_{35}\ 1), (P_{36}\ 1)\}$ $AF((P_{21}\ 1), (P_{22}\ 1)) \Longrightarrow \{(P_{37}\ 1), (P_{38}\ 1), (P_{39}\ 1), (P_{40}\ 0)\}$
4th Generation	$AF((P_{33}\ 0), (P_{40}\ 0)) \Longrightarrow \{(P_{41}\ 0), (P_{42}\ 0), (P_{43}\ 0), (P_{44}\ 0)\}$ $AF((P_{34}\ 1), (P_{38}\ 1)) \Longrightarrow \{(P_{45}\ 1), (P_{46}\ 0), (P_{47}\ 1), (P_{48}\ 1)\}$ $AF((P_{35}\ 1), (P_{39}\ 1)) \Longrightarrow \{(P_{49}\ 1), (P_{50}\ 1), (P_{51}\ 0), (P_{52}\ 1)\}$ $AF((P_{36}\ 1), (P_{37}\ 1)) \Longrightarrow \{(P_{53}\ 1), (P_{54}\ 1), (P_{55}\ 1), (P_{56}\ 1)\}$

Figure 7.4: A simplification of Mendel's pea experiments.

In these experiments, the first generation is a set of purebreds P_1 through P_8. Four of them are green and four are yellow. From these purebreds, three more generations have been produced by the action *Artificial Fertilization*, or AF, which takes two peas and produce four children. As reported in his paper, Mendel noticed that purebreds produce only purebreds, (for example, all the children of $AF((P_9\ 0), (P_{10}\ 0))$ are yellow and all the children of $AF((P_{13}\ 1), (P_{14}\ 1))$ are green), but hybrids will produce both green and yellow offspring (for example, $AF((P_{17}\ 1), (P_{18}\ 1))$ produces one yellow child and three green children, and so does $AF((P_{21}\ 1), (P_{22}\ 1))$).

Furthermore, Mendel also noticed some regularity in the colors of hybrids' children. The yellow children of hybrids (e.g. $AF((P_{33}\ 0)(P_{40}\ 0))$) will produce only yellow grandchildren, but green children of hybrids will produce grandchildren of different color with the following proportion: one third of the green children (see $AF((P_{36}\ 1), P_{37}\ 1))$) will produce only green grandchildren, but the other two thirds ($AF((P_{34}\ 1), (P_{38}\ 1))$ and $AF((P_{35}\ 1), (P_{39}\ 1))$) will produce three green grandchildren and one yellow grandchild.

From these regularities, Mendel hypothesized that a pea's color is determined by some invisible identities (now are known as genes) and the regularity of the color display of hybrids' children is determined by the combinations of these identities. When hypothesizing how these identities determine the color, Mendel also believed that some of identities are dominant (such as green here) and others are recessive (yellow). When both are present at the same time, only the dominant characteristic will show up.

In order to simulate Mendel's discovery with LIVE's learning and discovery methods, we have formalized his experiments as a problem of learning from environments in Figure 7.5.

- **Percepts** A set of peas, each of which is a feature vector (P_i *color*) where P_i is the identifier of the pea, and *color* can be 1 (green) or 0 (yellow).
- **Actions** AF(pea_x pea_y);
- **Constructors** Two primitive relations: = and >, and three functions: *max* (maximum), *min* (minimum), and an even distributed pairing function *EvenDistr*((a b) (c d)) = ((a c) (a d) (b c) (b d));
- **Goal** To predict the colors of hybridized children.

Figure 7.5: Learning from the Pea-Hybridization environment.

In addition to the assumptions that a pea's color is the only observable feature and two peas will produce exactly four children when fertilized, this definition also makes two other important assumptions. First, it assumes that the genes of the parents are evenly distributed into their four children according to the function *EvenDistr* (in other words, we are not concerned about statistics). This assumption has put a very strong bias on the discovery task because LIVE is given a small set of constructor functions and *EvenDistr* is one of them. If there were a large set of constructors, LIVE might of course have to make a

lengthier search, but the present set will illustrate the process. Second, we will not deal with the problem of how to choose peas for fertilization, although it might be another important part of Mendel's discovery.

In the following description, we assume that LIVE will remember the whole history of its actions in this environment, and each memorized historic element has four parts: the rule index, the state, the bindings, and the results of the action. In this way LIVE can recall what were the parents of a given pea.

Following the experiments in Figure 7.4, LIVE starts with a set of purebred peas as the first generation. To make the second generation, the system fertilizes yellow with yellow, green with green, yellow with green, and green with yellow. When applying $AF((P_1\ 0),(P_2\ 0))$ (with no prediction), the system observes that all the offspring, $(P_9\ 0),(P_{10}\ 0),(P_{11}\ 0)$ and $(P_{12}\ 0)$, are yellow, thus a new rule is constructed as follows:

Rule Index: 1 Condition: $((P_i\ c_i)\ (P_j\ c_j))$ **Action: AF**$((P_i\ c_i)\ (P_j\ c_j))$ **Prediction:** $((P_k\ c_i)\ (P_l\ c_i)\ (P_m\ c_i)\ (P_n\ c_i))$ **Sibling: ()**

This rule is created using the technique described in Chapter 5. In accordance with the relevancy assumption mentioned earlier, the rule's condition contains only the peas that are fertilized but not other facts such as the weather and the soil, etc.. The rule predicts that all the children will have the same color as P_i because when correlating the predictions with the action in this rule, LIVE examines c_i before c_j.

In the next two actions, this rule successfully predicts that the hybrids of green $(P_3\ 1)$ with green $(P_4\ 1)$ will be all green: $(P_{13}\ 1),(P_{14}\ 1),(P_{15}\ 1),(P_{16}\ 1)$, but fails to predict that the hybrids of yellow $(P_5\ 0)$ with green $(P_6\ 1)$ will be all green too: $(P_{17}\ 1),(P_{18}\ 1),(P_{19}\ 1),(P_{20}\ 1)$. The incorrect prediction is made because c_i is bound to yellow. This surprise causes LIVE to compare this application with the last one (green with green), finding a difference that $c_i = c_j$ was true previously but is false now. (Following the method described in Section 5.2.1, $c_i = c_j$ is in fact defined as a new relation RELxxx, such that $RELxxx(P_i, P_j) \Leftrightarrow (elt\ P_i\ 1) = (elt\ P_j\ 1)$, where 1 is the index for the color feature and $=$ is a given constructor, but we will use the shorthand $c_i = c_j$ in the description of the rules.) With this relational difference, Rule1 is then split into the two new rules as follows:

Rule Index: 1 Condition: $((P_i\ c_i)\ (P_j\ c_j)\ (c_i = c_j))$ **Action: AF**$((P_i\ c_i)\ (P_j\ c_j))$ **Prediction:** $((P_k\ c_i)\ (P_l\ c_i)\ (P_m\ c_i)\ (P_n\ c_i))$ **Sibling: 2**

Rule Index: 2 Condition: $((P_i\ c_i)\ (P_j\ c_j)\ \neg(c_i = c_j))$ **Action: AF**$((P_i\ c_i)\ (P_j\ c_j))$ **Prediction:** $((P_k\ c_j)\ (P_l\ c_j)\ (P_m\ c_j)\ (P_n\ c_j))$ **Sibling: 1**

Rule2 is then applied to predict that all hybrids of green $(P_7\ 1)$ with yellow $(P_8\ 0)$ will be yellow (because c_j is bound to yellow), but the result is again a surprise: $(P_{21}\ 1)$, $(P_{22}\ 1)$, $(P_{23}\ 1)$ and $(P_{24}\ 1)$ are all green. After comparing the two cases, LIVE finds that the reason for the surprise is $(c_j > c_i)$ (using the same technique for creating the relation $(c_i = c_j)$), and revises Rule2's condition as below (Rule1's condition is modified to become the complement of Rule2's condition). These two new rules reflect the fact that the green color dominates the yellow color, and that ends the second generation:

Rule Index: 1 Condition: $((P_i\ c_i)\ (P_j\ c_j)\ \neg(\neg(c_i = c_j)\ (c_j > c_i)))$ **Action: AF**$((P_i\ c_i)\ (P_j\ c_j))$ **Prediction:** $((P_k\ c_i)\ (P_l\ c_i)\ (P_m\ c_i)\ (P_n\ c_i))$ **Sibling: 2**

Rule Index: 2 Condition: $((P_i\ c_i)\ (P_j\ c_j)\ \neg(c_i = c_j)\ (c_j > c_i))$ **Action: AF**$((P_i\ c_i)\ (P_j\ c_j))$ **Prediction:** $((P_k\ c_j)\ (P_l\ c_j)\ (P_m\ c_j)\ (P_n\ c_j))$ **Sibling: 1**

The third generation of the experiment is made by self-fertilizing the second generation. We assume the pairs to be hybridized are the same as the third generation in Figure 7.4. When fertilizing $(P_9\ 0)$ with $(P_{10}\ 0)$, and $(P_{13}\ 1)$ with $(P_{14}\ 1)$, Rule1 successfully predicts the children will be $\{(P_{25}\,0)(P_{26}\,0)(P_{27}\,0)(P_{28}\,0)\}$ and $\{(P_{29}\,1)(P_{30}\,1)(P_{31}\,1)(P_{32}\,1))\}$ respectively. However, the hybridization of $(P_{17}\ 1)$ with $(P_{18}\ 1)$ surprises LIVE because the children have different colors: $(P_{33}\,1)(P_{34}\,1)(P_{35}\,1)P_{36}\,0))$. In explaining the surprise, the last application $((P_{13}\ 0)\ (P_{14}\ 0))$ is brought in to compare with the current application, but the system fails to find any relational difference because both pairs are green. This is the point where hidden features must be discovered. Since the environment is time-dependent, LIVE traces back in the history and searches for differences in previous states. Fortunately, a difference is found when comparing the parents of these two indistinguishable states: $c_i = c_j$ was true for $((P_{13}\ 1)\ (P_{14}\ 1))$'s parents $[(P_3\ 1)\ (P_4\ 1)]$ but not for $((P_{17}\ 1)\ (P_{18}\ 1))$'s parents $[(P_5\ 0)\ (P_6\ 1)]$. Since each pea has one "mother" and one "father", this difference indicates the existence of two hidden features from the grandparents, denoted as m and f in the following, that are invisible in the parental generation but are necessary to determine the grandchildren's color. The representation of the pea is then extended to include three features: (*color, m, f*).

As we noted before, two things must be done at this point in order to use the hidden features. One is to figure out how m and f are related to the *color* feature, and the other is to figure out how m and f are inherited through the action AF.

For the first task, LIVE begins to collect examples in order to figure out the determination. Since it is known that the first generation peas are purebred, that is, all their ancestors are having the same color, their m and f must be the same as their color. Thus all the peas in the first generation are the relevant examples: The second generation peas are also ex-

(P_1 0 0 0) (P_2 0 0 0) (P_3 1 1 1) (P_4 1 1 1) (P_5 0 0 0) (P_6 1 1 1) (P_7 1 1 1) (P_8 0 0 0)

amples of how m and f determine color. This is because their parents, the first generation, all have a single value for m, f and color, and no matter how m and f are inherited, each parent has only one possible value to inherit. For example, (P_9 0)'s m and f can only be 0 because its parents (P_1 0 0 0) and (P_2 0 0 0) have only 0s to inherit. (P_{17} 1)'s m and f can only be (0 1) because its "mother" (P_5 0 0 0) has only 0s and its "father" (P_6 1 1 1) has only 1s. Therefore, the examples from the second generation are as follows: For the peas in

(P_9 0 0 0) (P_{10} 0 0 0) (P_{11} 0 0 0) (P_{12} 0 0 0) (P_{13} 1 1 1) (P_{14} 1 1 1) (P_{15} 1 1 1) (P_{16} 1 1 1) (P_{17} 1 0 1) ..... (P_{21} 1 1 0) .....

third generation, some of them, such as (P_{25}000) (P_{29}000), are also examples because their parents are still purebreds. But others may not, such as (P_{33}000) or (P_{37}000), because their parents are hybrids and it is uncertain at this point how the ms and fs are inherited from hybrids.

With the examples collected, it is straightforward for LIVE to search through the constructor functions and see that the following formula is consistent with all the examples (i.e. 0=*max*(0 0), 1=*max*(1 1), 1=*max*(1 0), and 1=*max*(0 1)): This formula, in Mendel's

color = *max*(**m f**),	**[Formula-1]**

words, says that a pea's color is determined by the dominant color: green.

For the second task, we have known that parents' m and f are inherited into all four children's m and f, and the actions LIVE has done can be used as examples. Here, we list some of the examples that show how the third generation is produced from the second

generation (where surprises arise):

$$(P_9\ 0\ 0\ 0)\ (P_{10}\ 0\ 0\ 0) \implies (P_{25}\ 0\ 0\ 0)\ (P_{26}\ 0\ 0\ 0)\ (P_{27}\ 0\ 0\ 0)\ (P_{28}\ 0\ 0\ 0)$$

$$(P_{13}\ 1\ 1\ 1)\ (P_{14}\ 1\ 1\ 1) \implies (P_{29}\ 1\ 1\ 1)\ (P_{30}\ 1\ 1\ 1)\ (P_{31}\ 1\ 1\ 1)\ (P_{32}\ 1\ 1\ 1)$$

$$(P_{17}\ 1\ 0\ 1)\ (P_{18}\ 1\ 0\ 1) \implies (P_{33}\ 1\ x_1\ x_2)\ (P_{34}\ 1\ x_3\ x_4)\ (P_{35}\ 1\ x_5\ x_6)\ (P_{36}\ 0\ x_7\ x_8).$$

These examples, together with the formula *color = max(m,f)* we just identified, have specified a unknown function, $Inherit$, that must satisfy the following equations:

$$\begin{aligned}
Inherit((00)(00)) &= ((00)(00)(00)(00)) \\
Inherit((11)(11)) &= ((11)(11)(11)(11)) \\
Inherit((01)(01)) &= ((x_1\ x_2)(x_3\ x_4)(x_5\ x_6)(x_7\ x_8)) \\
1 &= max(x_1, x_2) \\
1 &= max(x_3, x_4) \\
1 &= max(x_5, x_6) \\
0 &= max(x_7, x_8)
\end{aligned}$$

Searching through the constructor functions, LIVE finds that *EvenDistr* fits the data (i.e. *EvenDistr* can replace $Inherit$), thus concludes that if the mother has $(m_i\ f_i)$ and the father has $(m_j\ f_j)$, then four children's m and f will be determined as follows: Note

$EvenDistr((m_i\ f_i)(m_j\ f_j)) = ((m_i\ m_j)(m_i\ f_j)(f_i\ m_j)(f_i\ f_j))$ **[Formula-2]**

that m and f are not defined in terms of any observables but of m and f *recursively*. With Formula-1 and Formula-2, LIVE now modifies Rule1 (the one that is surprised) and Rule2 as follows (Rule2's condition is the complement of Rule1's):

Rule Index: 1 Condition: $((P_i\ c_i\ m_i\ f_i)\ (P_j\ c_j\ m_j\ f_j)\ \neg(\neg(c_i = c_j)\ (c_j > c_i))\ (m_i = f_i)\ (m_j = f_j))$ **Action:** $\mathbf{AF}((P_i\ c_i\ m_i\ f_i), (P_j\ c_j\ m_j\ f_j))$ **Prediction:** $((P_k\ max(m_i\, m_j)\ m_i\ m_j)\ (P_l\ max(m_i\, f_j)\ m_i\ f_j)\ (P_m\ max(f_i\, m_j)\ f_i\ m_j)\ (P_n\ max(f_i\, f_j)\ f_i\ f_j))$ **Sibling: 2**

Note that Rule1 and Rule2 now make the same prediction. This is because the newly defined m and f determine a pea's color, and they are distributed into the prediction in the same way (by the function *EvenDistr*) no matter what the condition is.

Rule Index: 2 Condition: $((P_i\ c_i\ m_i\ f_i)\ (P_j\ c_j\ m_j\ f_j)\ \neg(\neg(\neg(c_i = c_j)\ (c_j > c_i))\ (m_i = f_i)\ (m_j = f_j)))$ **Action:** $\mathbf{AF}((P_i\ c_i\ m_i\ f_i), (P_j\ c_j\ m_j\ f_j))$ **Prediction:** $((P_k\ max(m_i\, m_j)\ m_i\ m_j)\ (P_l\ max(m_i\, f_j)\ m_i\ f_j)\ (P_m\ max(f_i\, m_j)\ f_i\ m_j)\ (P_n\ max(f_i, f_j)\ f_i\ f_j))$ **Sibling: 1**

With these two new rules, LIVE is now ready to predict other hybrids' color. When (P_{21} 1) and (P_{22} 1) are hybridized, LIVE will determine that they are really (P_{21} 1 1 0) and (P_{22} 1 1 0) by backtracking how they were produced. Rule2 predicts correctly that among their four children, three will be green and one will be yellow.

At this point, after the hidden features are made salient, it is interesting to see Mendel's experiments again, shown in Figure 7.6. This time, the addition of m and f provides a much clearer picture.

	Notation: A pea is now represented as (P_i *color* m f).
1st Generation (purebreds)	$(P_1\,000), (P_2\,000), (P_3\,111), (P_4\,111),$ $(P_5\,000), (P_6\,111), (P_7\,111), (P_8\,000).$
2nd Generation	$AF((P_1\,000), (P_2\,000)) \Longrightarrow \{(P_9\,000), (P_{10}\,000), (P_{11}\,000), (P_{12}\,000)\}$ $AF((P_3\,111), (P_4\,111)) \Longrightarrow \{(P_{13}\,111), (P_{14}\,111), (P_{15}\,111), (P_{16}\,111)\}$ $AF((P_5\,000), (P_6\,111)) \Longrightarrow \{(P_{17}\,101), (P_{18}\,101), (P_{19}\,101), (P_{20}\,101)\}$ $AF((P_7\,111), (P_8\,000)) \Longrightarrow \{(P_{21}\,110), (P_{22}\,110), (P_{23}\,110), (P_{24}\,110)\}$
3rd Generation	$AF((P_9\,000), (P_{10}\,000)) \Longrightarrow \{(P_{25}\,000), (P_{26}\,000), (P_{27}\,000), (P_{28}\,000)\}$ $AF((P_{13}\,111), (P_{14}\,111)) \Longrightarrow \{(P_{29}\,111), (P_{30}\,111), (P_{31}\,111), (P_{32}\,111)\}$ $AF((P_{17}\,101), (P_{18}\,101)) \Longrightarrow \{(P_{33}\,000), (P_{34}\,101), (P_{35}\,110), (P_{36}\,111)\}$ $AF((P_{21}\,110), (P_{22}\,110)) \Longrightarrow \{(P_{37}\,111), (P_{38}\,110), (P_{39}\,101), (P_{40}\,000)\}$
4th Generation	$AF((P_{33}\,000), (P_{40}\,000)) \Longrightarrow \{(P_{41}\,000), (P_{42}\,000), (P_{43}\,000), (P_{44}\,000)\}$ $AF((P_{34}\,101), (P_{38}\,110)) \Longrightarrow \{(P_{45}\,101), (P_{46}\,000), (P_{47}\,111), (P_{48}\,110)\}$ $AF((P_{35}\,110), (P_{39}\,101)) \Longrightarrow \{(P_{49}\,110), (P_{50}\,111), (P_{51}\,000), (P_{52}\,101)\}$ $AF((P_{36}\,111), (P_{37}\,111)) \Longrightarrow \{(P_{53}\,111), (P_{54}\,111), (P_{55}\,111), (P_{56}\,111)\}$

Figure 7.6: Mendel's pea experiments with hidden features m and f.

There are a few differences between LIVE's behavior and Mendel's experiments. First,

LIVE has defined the hidden features when fertilizing the third generation, while Mendel's experiments lasted 8 years. Of course, he might have had the idea very early in the experiments and all the later experiments were verifications of his hypothesis. Second, when defining hidden features, LIVE assumes that children's color is determined by their ancestors' color only. Mendel may have considered many more potential combinations of features. If children's color is determined by parents' height, LIVE's discovery method will not always work.

Nevertheless, gene discovery has provided a natural example of hidden features that can lie outside a given concept language (or an axiom system). In contrast to previous discovery systems, such as BACON [25] and STABB [**?**], these hidden features are not functional combinations of any observable terms inside the language but stand by themselves. In this example, even though the first generation's m and f are equivalent to their observable color, they are by no means *determined* by the colors. Some interesting characteristics will become clear in the next section, as we compare the hidden features with the theoretical terms.

7.5 The Recursive Nature of Theoretical Terms

This section examines the relation between hidden features (especially in time-dependent environments) and the *theoretical terms* (for example, see [48]) long studied in the philosophy of science. Detailed analysis has revealed an interesting aspect of theoretical terms that has not been studied before and it may cast some light on the discussions of the identifiability of theoretical terms.

By definition, theoretical terms are those terms that are not directly observable, but derivable from observables. The concept of torque in the Balance Beam environment and genes in the Mendel's experiments are two examples. However, whether it is possible to define such terms based on given experiments and observables depends on what is the *definition* of such terms.

Lesniewski has proposed that definitions should satisfy two conditions, which P. Suppes [53, p.153] has described intuitively:

> Two criteria which make more specific ... intuitive ideas about the character of definitions are that (i) a defined symbol should always be eliminable from any formula of the theory, and (ii) a new definition does not permit the proof of

relationships among the old symbols which were previously unprovable; that is, it does not function as a creative axiom.

Following Suppes, we will refer these two criteria as *eliminability* and *noncreativity*. Formally, Tarski [55, pp. 301-302] has proved a theorem that a term can be defined by means of other terms if and only if its definition is derivable from the primitive terms and axioms. These criteria and theorem, often repeated in works of logic, stem from the notion that definitions are mere notational abbreviations, allowing theory to be stated in more compact form without changing its content in any way.

More recently, Simon [48] has proposed a somewhat weaker condition called *general definability*, which says that a term is generally definable by means of the other terms if a sufficient number of observations is taken. His argument is based on the analysis that some "should be defined" theoretical terms are excluded by Tarski's criteria, and if observations are sufficient then theoretical terms can be both eliminable and creative simultaneously. Here, we briefly repeat his axiomatization of Ohm's law, for it is the best example to illustrate his argument.

Γ is a system of Ohmic observations iff there exist D, r, c, such that:

(1) $\Gamma = \langle D, r, c \rangle$;

(2) D is a non-empty set;

(3) r and c are functions from D into the real numbers;

(4) for all $x \in D, r(x) > 0$ and $c(x) > 0$.

Γ' is an Ohmic circuit iff there exist D, r, c, b, and v such that:

(5) $\Gamma' = \langle D, r, c, b, v \rangle$;

(6) $\Gamma = \langle D, r, c \rangle$ is a system of Ohmic observations;

(7) v and b are real numbers;

(8) for all $x \in D$,

$$[\alpha]. \qquad c(x) = v/(b + r(x)).$$

In this system, r and c are the observables (they stand for the external resistance and the current respectively), and b and v are theoretical terms (they stand for the internal resistance and the voltage of the battery respectively). However, according to the method of Padoa (Tarski [55, pp. 304-305]), v and b are not definable in the system above as they should be unless the following additional condition is given:

. $b = (c_2r_2 - c_1r_1)/(c_1 - c_2)$ $[\gamma]$. $v = (c_2r_2 - c_1r_1)/(c_1 - c_2)$.

(9) D contains at least two members with distinct r's and c's.

This is because after two such observations, (c_1, r_1) and (c_2, r_2), b and v can be uniquely defined as follows: Furthermore, if we substitute these values for v and b in $[\alpha]$, and substitute the third observation (c_3, r_3) for $c(x)$ and $r(x)$ respectively, then we obtain a relation among three pairs of observations. Thus, the system becomes creative because the relation can hold among any three pair of observations. Simon's general definability has included the condition that a sufficient number of observations must be made to uniquely determine the values of the theoretical terms like v and b.

However, Simon did not make any further distinctions among observations. This is suitable for time independent environments, such as an Ohmic circuit and a balance beam, because the values of theoretical terms depend only on the values of observables regardless of when and in which order the observations are made. In the Ohmic circuit, for example, the values of b and v depend only on the values of r's and c's, thus *any* two observations in the circuit could serve for the definitions of b and v. Examining equations $[\beta]$ and $[\gamma]$ carefully, we can see that no theoretical terms appear on the right hand side of the equations, and the equations do not have the sense of time.

In the time dependent environments, the situation is not that simple because of the order of actions. The values of theoretical terms can be different at different times even though the values of observables at those times may appear the same. For example, in Mendel's experiments, the values of m and f are different for (P_3) and (P_{17}) even though they are both green. If we examine how m and f are defined, we can see that the theoretical terms are defined *recursively* in terms of themselves; the definition has a clear sense of time because they are linked by the action AF:

Before AF (parent's m and f): $((m_i\ f_i)(m_j\ f_j))$ **After AF (children's m and f):** $((m_i\ m_j)(m_i\ f_j)(f_i\ m_j)(f_i\ f_j))$

In general, theoretical terms might have to be defined recursively in time dependent environments. These terms do not depend on the values of observables at the current time but on the history of the observables and the actions, for example purebred pea's m and f depend on their ancestors color. Furthermore, these terms also depend on the history of

themselves, for example hybrids m and f depend on their parent's m and f.

Formally, in order to both define and use a set of theoretical terms T in a time dependent environment, the following three functions might have to be identified:

E(): How T is computed from existing terms, if possible; **U():** How T is used to determine the values of observables; **I():** How T is inherited by the actions.

Since theoretical terms in a time dependent environment are defined recursively on themselves, the first function $E()$ meant to ask where the first generation of theoretical terms come from. This function may not be possible to identify because it is equivalent to asking "where the first egg comes from". However, in the environment of Mendel's experiments, we have used the fact that the purebreds' m and f are the same as their color because all their ancestors are known to have the same color. (Such fact is not $E()$ in a real sense because m and f may differ from their color if the peas are not purebred.) Nevertheless, in that environment, the time line between the purebreds and the hybrids provides us a starting point for using the recursive definitions. But whether such starting points always exist and are identifiable in any time dependent environment is an open question. Furthermore, if no such starting point exists, that is, no $E()$ exists, then whether or not these recursive theoretical terms are identifiable is another intersting question to ask.

The second function, $U()$, tells us how the theoretical terms are used at a particular time point. If $U()$ exists, then the theoretical terms are *meaningful* because they determine, thus predict, the values of observables. For example, in Mendel's experiments, a pea's m and f determine its color. This may suggest that, at any particular time point, these recursively defined theoretical terms may not be eliminable because they not only determine the values of observables at that time but also preserve the information from history. This is another deviation of identifiability because previously only time independent environments were concerned.

Finally, the third function $I()$ is the useful (vs. $E()$) definition of recursive theoretical terms. The function tells us how such terms are developed, and once they are identified how they are changed with time (actions can also be viewed as time). In some sense, once we know the function $I()$ and the current values of theoretical terms T, then we can use T without knowing the "real" definition of T. The function $I()$ also indicates that actions may play some important roles in defining recursive theoretical terms in time dependent

environments. Further study is needed to utilize such "floating" definition in our attempts to form scientific theories computationally.

Based on these analysis, we can safely say that theoretical terms can be recursive, that is, not defined in a concrete sense but defined by how they are inherited through actions. It is important to note that theoretical terms are derivable from the observables, but it is also important to note that how they are derived and defined from themselves. As a special property of the theoretical terms, the recursive nature may add some interesting discussions on the identifiability of theoretical terms.

7.6 Comparison with Some Previous Discovery Systems

7.6.1 Closed-Eye versus Open-Eye Discovery

Comparing with some previous discovery systems in mathematics, like AM and [27] ARE [43], learning from environments has an interesting character. It not only "thinks" but also interacts with the surrounding environment. This property, we think, is a very important component for a system to discover novel things that are unknown to its designer. We shall call the systems with such ability open-eye discovery, while those lack of such ability, closed-eye discovery.

AM is a system designed to discovery "interesting" mathematic concepts and functions. It starts with a small set of primitive concepts and functions, then lunches the search with a set of powerful concept manipulating operators. The search is guided by the interestingness of concepts which is determined solely by a set of fixed heuristics. Since AM does not need to interact with any external environments, it is a closed-eye discovery system.

In learning from environments, the interestingness of new concepts is determined by the evaluation in the environment, not by the learner itself. In different environment, different concepts will be learned. In some sense, the environment plays the role of teacher; the learner asks questions through predictions. Thus, any system that learns from environments is open-eye.

Compare to previous data driven discovery systems like BACON, we have seen that LIVE's discovery is more autonomous. Our system not only finds the regularity in a given set of data, like BACON did, but also detects when a discovery is needed and collects examples itself.

7.6.2 Discrimination Helps STABB to Assimilate N22S Properly

Besides for the purpose of discovery , hidden features are also studied as the problem of shifting biases in learning. Utgoff [**?**, 56] has designed a program called STABB for automating such process in inductive concept learning when the initial concept space is biased. In his framework, the learner starts with a restricted hypothesis space (a bias), and shifts to a better one according to the following three steps:

1. Recommend (via heuristics) new concept description to be added to the concept description language.

2. Translate recommendations into new concept descriptions that are expressed in the formalism of the concept description language.

3. Assimilate any new concepts into the restricted space of hypotheses so that the organization of the hypothesis space is maintained.

However, since he does not compare failures with successes for learning, his program STABB has encounted some difficulties for properly assimilating newly defined terms into the restricted hypothesis space. A well know example is the new term N22S, which is defined as the definition of even integers when STABB notices the same sequence of operators leads the state $\int(\cos^7 x)dx$ to a success but the state $\int(\cos^6 x)dx$ to a failure. When describing how his Constraints Back-Propagation procedure assimilates a new term [56, page 134], he wrote:

> The procedure assimilates a new description NS by adding a new grammar rule of the form $d \Rightarrow NS$. The description d is the unconstrained description that was used in the domain of the corresponding operator.

Thus, N22S has to be assimilated under the real number instead of the integer because the domain of the corresponding operator is the real number.

During the construction of N22S, STABB only considered the successful problem solving trace from the state $\int(\cos^7 x)dx$, but ignored the failure trace from the state $\int(\cos^6 x)dx$. From our point of view, the new term N22S is necessary only because $\int(\cos^7 x)dx$ and $\int(\cos^6 x)dx$ are indistinguishable in the current language yet they yield different results when the same sequence of operators is applied. If we compare these two states, it is easy

to notice that the reason 6 and 7 are indistinguishable in the current language is that the least general concept, integer, includes both 6 and 7, while all concepts that are more specific than integer exclude both 6 and 7. Thus, it is the concept of integer, not anything else, that needs further discrimination. Naturally, after N22S is defined, it should be assimilated for the purpose of *discriminating* integer, not real number.

7.6.3 LIVE as an Integrated Discovery System

In their book on scientific discovery [26], Langley, Simon *at el* have outlined a system that infers Mendel's model by integrating two separate systems: GLAUBER, a data-driven system that can form classes from instances, and DALTON, a theory-driven system that can infer internal structures based on the reactions between classes. Using a given high-level predicate *child-has-quality* and Mendel's data, the first system GLAUBER can divide the peas into three classes, G (pure greens), G' (hybrid greens) and Y (pure yellow), and constructs four rules to describe the reactions between these classes:

(G G → G)

(Y Y → Y)

(G Y → G')

(G' G' → G G' Y)

The first two rules state that pure green peas produce only pure green offspring and that pure yellow peas produce only pure yellow offspring. The third rule states that crossing pure green and pure yellow peas produces only mixed green offsprings. The four rules states that breeding hybrid green peas generates offspring in all three classes.

Given these rules, the second system DALTON can infer that two primitive traits (say g and y) are required, and decide that the genotype G can be modeled by the pair (g g), that Y can be modeled by the pair (y y), and that G' can be modeled by the pair (g y). These structural inferences are then past back to GLAUBER to form a law that states that g is the dominant trait of peas' color.

Although the pipeline of GLAUBER and DALTON seems capable of inferring Mendel's model, these two systems are still separated, one-space search programs. Nothing has been said if the information flow contains errors, nor about how one system's search can help the other. For example, if the classes or the laws generated by GLAUBER are incorrect, then DALTON's search for substructures may fail or return some fruitless results. Moreover,

this pipeline style of integration cannot explain convincingly why GLAUBER thinks that *child-has-quality* is the key feature to consider, and why DALTON must infer the particular substructures g and y.

Unlike GLAUBER and DALTON, LIVE integrates the search of new laws (classes) and the search of new terms (substructures) naturally. As we have seen in this chapter, a failure of a law's prediction triggers LIVE to search for hidden features, and the resulting features help LIVE to form better laws. At any stage, neither the laws nor the hidden features are required to be completely correct. The discovery process in LIVE is an interaction between these two search spaces.

Chapter 8

Performance Analysis

The methods developed in this research can be best evaluated if they are implemented on a real robot and the robot is given problems that it has never seen before. However, within the scope of this thesis, we evaluate the methods by theoretical analysis as well as experiments conducted in various domains.

This chapter presents the performance of LIVE on a set of experiments chosen from several different domains, as well as some theoretical analysis on a simplified version of LIVE's learning algorithm. We hope the formal analysis can help us to understand the nature of the problem better, and the variety of the domains can demonstrate both LIVE's strengths and weaknesses. The environments used in this thesis include: the RPMA Tower of Hanoi, the Balance Beam experiments, the Mendel's pea-hybridization experiments, and the Hand-eye version of Tower of Hanoi. In some of the environments, LIVE is tested with various conditions. In all the test environments, LIVE's performances are satisfactory and encouraging.

LIVE is currently implemented in Common Lisp and runs on a IBM-RT PC with 10 megabytes of primary memory. When running, LIVE is interacting with an "environment" process which is specified by some separated Lisp files (see Appendex A) and the communication is conducted through the symbols that exported from the environment. Thus, when changing from one domain to another or changing the conditions within a particular environment, we need only replace the environment files.

8.1 LIVE in the RPMA Tower of Hanoi Environment

This environment has been used intensively throughout the thesis, its definition can be found in Section 2.1, and its implementation is in Appendix A.1. There are three kinds of experiments in this section: experiments with different goals; experiments with different exploration plans; and experiments with different number of disks.

As we pointed out in Section 4.4.3, LIVE's problem solving in this environment is guided by a given domain-specific heuristic, namely, whenever LIVE faces the decision where to put the disk in his hand, it always chooses the "other" peg (the one that is not mentioned in the most recent goal).

8.1.1 Experiments with Different Goals

Table 8.1 shows LIVE's performance in the environment with different goals. The experiments are conducted under the conditions that the number of disks is 3; the disks are initially on PEG1; and the exploration plan is to pick DISK1 from PEG1 and put it down on PEG2. Table 8.1 (as all the following tables in this section) is formated as follows: the first column shows the goal expressions that are given to the system; the second column, which has two numbers, shows the total number of steps (including both actions and subgoal proposing) LIVE has spent on the problem and the last step (the number in parenthese) that LIVE had revised its rules; the third column shows the CPU time (LIVE is not compiled when these experiments are run.); the fourth column is the number of exploring actions made; the fifth is the number of surprises araised in the plan execution; the sixth is the number experiments conducted by LIVE (which is equal to the number of errors detected at the planning time, see Section 6.3); and the last column shows the total number of rules that have been learned.

The first three experiments (numbered according to their corresponding row in Table 8.1) test how LIVE behaves when the same goals are given but expressed in different orders. As we can see, the learning results are the same but the total time (including both learning and problem solving) is inversely proportional to the correctness of the given goal expression. For example, the first goal expression is the worst (for it implies smaller disks are always put on the target peg before larger ones), but LIVE's running time is the shortest. These experiments indicate that LIVE prefers to have surprises or errors happen at the early

Testing Condition: Number of Disks: 3 Initial State: ((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG1)) Exploration Plan: ((MPICK DISK1 PEG1) (MPUT DISK1 PEG2))						
Goal Expression	# of Steps	CPU Sec.	# of Explore	# of Surprise	# of Experim	# of Rules
(ON 1 3)(ON 2 3) (ON 3 3)	32(15)	191.59	2	4	1	4
(ON 3 3)(ON 2 3) (ON 1 3)	40(12)	231.48	2	4	1	4
(ON 2 3)(ON 1 3) (ON 3 3)	52(38)	273.23	2	4	1	4
(ON 3 1)(ON 2 3) (ON 1 3)	8(2)	37.31	2	0	0	2
(ON 3 3)(ON 2 1) (ON 1 3)	36(12)	201.87	2	4	1	4
(ON 1 1)(ON 2 1) (ON 3 3)	65(30)	348.61	2	5	2	4

Table 8.1: Experiments with different goals.

stage of problem solving for it can learn the correct rules before wasting too much time on attempting to solve the goals with a set of wrong rules. For example, LIVE spends a longer time in the third experiment because it does not meet any surprise or error until both DISK1 and DISK2 are already on the target peg. From these experiments, it seems that learning correct rules is more important than making progress in problem solving.

The remaining experiements shows LIVE's performance when the contents of the goals are different. The fourth experiment shows if the goal is too trival to achieve, LIVE might not learn the complete rules in one run (but it will when new and more difficult goals are given). In this particular experiment, the goals and the exploration plan are so well arranged that LIVE has no chance to make any mistakes and it is happy about the two incomplete rules learned in the exploration.

The last experiment is also interesting; it shows that the learning is sensitive to the different problem solving situations. In this run, the particular arrangement of the goals has caused LIVE to make more mistakes hence make more experiments to learn the correct rules. Consequently, both learning time and problem solving time in this problem are longer

than the others. Since this test run is most typical and informative for LIVE's behavior, we have included its detailed running trace in Appendix B.

Note that in all these experiments the number of learned rules is always 4, two rules for the action *pick* and the other two for the action *put*. This is because the actions employed in the exploration are so informative that LIVE does not create any unfruitful rules. In the next section, we will see how LIVE behaves if the exploration plan is not so perfect.

8.1.2 Experiments with Different Explorations

This section presents some experiments in which LIVE uses imperfect exploration plans. There are two sets of experiments in Table 8.2, the first set consists of experiments under the same testing conditons as those in the last section except for the exploration plans; the second set shows LIVE's behave with different initial states, goals and exploration plans.

In each set of exeriments, there are two exploration plans. Note that the first exploration plan, Exploration-Plan-1, is noneffective because picking up DISK3 underneath DISK1 and DISK2 will have no effect in the environment. To run these tests, we have forced LIVE to use the first plan first. In doing so, LIVE will create two "no-change" rules in the exploration and attempt to use them to achieve the given goals. Since these two new rules do not specify any changes in the environment, planning for a solution will fail and LIVE is forced to do some more exploration. At this point, we assume LIVE will propose and use the second exploration plan. Note that this is a simplified case for demonstrating both the difficulties of proposing a good exploration plan and the effectiveness of exploration on learning and problem solving. If LIVE did not propose the correct exploration plan the second time, then it will spend a longer time on exploration because there may be many fruitless explorations before it hits the right one.

Comparing these experiments with those in the last section, one can find that although different explorations do not change LIVE's behavior very much in general, two things are different: LIVE has learned 5 rules instead of 4, and problems will take less time to solve if LIVE explores the environment more thoroughly before it starts to attack the given goals.

The extra rule is created during the noneffective exploration when LIVE does the Move-Put action when nothing is in his hand, it looks like the following: This rule is useless in the

---

Condition: ((**ON** $disk_x$ peg_y) ($disk_x$) (peg_y)) **Action:** **Move-Put**($disk_x$ peg_y) **Predict:** ((**ON** $disk_x$ peg_y) ($disk_x$) (peg_y))

Testing Condition:
Number of Disks: 3
Initial State: ((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG1))
Exploration-Plan-1: ((MPICK DISK3 PEG1) (MPUT DISK3 PEG2))
Exploration-Plan-2: ((MPICK DISK1 PEG1) (MPUT DISK1 PEG2))

Goal Expression	# of Steps	CPU Sec.	# of Explore	# of Surprise	# of Experim	# of Rules
(ON 1 3)(ON 2 3) (ON 3 3)	32(15)	188.23	4	4	1	5
(ON 3 3)(ON 2 1) (ON 1 3)	36(12)	206.53	4	4	1	5
(ON 1 1)(ON 2 1) (ON 3 3)	57(22)	299.26	4	4	1	5

Testing Conditions:
Number of Disks: 3
Initial State: ((ON DISK1 PEG2) (ON DISK2 PEG1) (ON DISK3 PEG1))
Exploration Plan 1: ((MPICK DISK3 PEG1) (MPUT DISK3 PEG1))
Exploration Plan 2: ((MPICK DISK2 PEG1) (MPUT DISK2 PEG1))

Goal Expression	# of Steps	CPU Sec.	# of Explore	# of Surprise	# of Experim	# of Rules
(ON 1 3)(ON 2 3) (ON 3 3)	8/25	160.00	4	3	0	5
(ON 3 2)(ON 2 3) (ON 1 1)	10/29	178.59	4	3	0	5
(ON 1 2)(ON 2 2) (ON 3 2)	14/31	193.24	4	4	1	5
(ON 1 2)(ON 2 1) (ON 3 3)	16/45	252.29	4	3	0	5

Table 8.2: Experiments with different exploration plans.

problem solving because it does not specify any changes in the environment. But it does no harm to the learning and problem solving either. Instead, such dumy rule will prevent LIVE creating similar rules again if its future exploration includes more actions like putting nothing down.

Another difference is that if LIVE does more exploration then it may shorten the total time for solving a given problem. For example, the second problem in Table 8.2, which took LIVE 348.61 CPU seconds to solve in Table 8.1, is solved in 299.26 seconds here. This

phenomena is consistent to the analysis that LIVE prefers early errors than late errors; the noneffective exploration provides the chances for LIVE to be surprised in the effective exploration.

From the concept-learning point of view, doing noneffective exploration before effective exploration is like presenting a learning program with negative examples before any positive examples. The fact that LIVE's behavior in this case is no worse than the cases in Table 8.1 demonstrates again that the complementary discrimination learning method is less sensitive to the orders of the training instances.

8.1.3 Experiments with Different Number of Disks

This section presents some experiments with increasing numbers of disks in the Tower of Hanoi environment. Table 8.3 lists the results when the enviornment has 4 and 5 disks (with other testing conditions set to the same as those in Table 8.1). As we can see, except that a longer time is required to solve the problems here, LIVE's learning behavior remains the same. For learning the same number of rules, LIVE takes approximately the same number of steps, surprises, and experiments.

One may wonder why the size of problem does not increase the difficulty of learning. The reason is that the rules learned by LIVE are developed from the most general cases; they do not depend on how many disks are on a peg but how the disk in the action is related to the others. In fact, the rules are not only useful for a larger number of disks, but also transferable from one experiment to another. In other words, rules learned in one problem will work in other given problems as well.

8.2 LIVE on the Balance Beam

This section presents LIVE's behavior in the Balance Beam environment. The definition of the environment is in Section 7.3 and its implementation is in Appendix A.2. The main purpose of this section is to demonstrate how the order of training instances and the given constructors (and their order) influnce LIVE's learning performance on discovering new hidden features. Since tasks in this environment is to predict the correct outcome, LIVE's problem solving components, although inseparable from the system as a whole, are not in action. Experiments are conducted in a larger set of experiments generated randomly; but only those most informative are included here.

The Testing Conditions: Number of Disks: 4 Initial State: ((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG1) (ON DISK4 PEG1) Exploration Plan: ((MPICK DISK1 PEG1) (MPUT DISK1 PEG2))						
Goal Expression	# of Steps	CPU Sec.	# of Explore	# of Surprise	# of Experim	# of Rules
(ON 4 3)(ON 3 3) (ON 2 3)(ON 1 3)	116(14)	1035.49	2	4	1	4
(ON 1 2)(ON 2 1) (ON 3 1)(ON 4 2)	127(20)	1125.58	2	4	1	4
The Testing Conditions: Number of Disks: 5 Initial State: ((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG1) (ON DISK4 PEG1) (ON DISK5 PEG1) Exploration Plan: ((MPICK DISK1 PEG1) (MPUT DISK1 PEG2))						
(ON 1 3)(ON 2 3) (ON 3 3)(ON 4 3) (ON 5 3)	340(16)	4630.75	2	4	1	4

Table 8.3: Experiments with different number of disks.

8.2.1 Experiments with Different Orders of Training Instances

Table 8.4 is a summary of the particular run described in Section 7.3. The constructors are listed as the testing condition. The format of the table is the following: the first column of the table are the indexes of the training tasks; the second column is the state seen by LIVE; the third is LIVE's prediction; the fourth is the reason why LIVE made the prediction; the fifth is the result of the prediction, and finally, the last column is what LIVE learned from the mistakes. In the the fourth column of the table, the "reasons" are represented by the predicates and they should be read together with the objects and their order in the state column (the second column). For instance, the reason (W>) in row 4 should be read as (W> L R) (because L precedes R in the state 4), while the reason (W>) in row 5 should be read as (W> R L) (because R precedes L in state 5). The predicate (W×D>) is a shorthand for the relational predicate "torque greater".

As we can see from Table 8.4, the sequence of training tasks serves pretty well for the

Testing Conditions: Constructor Functions: (×, +) Constructor Relations: (>, =).					
#	State	Prediction	Reason	Result	Learned
1	((L 3 1 1)(R 3 1 1))	None	None	None	Always
2	((L 3 1 1)(R 2 1 1))	Balance	Always	Wrong	(W>)
3	((L 3 3 1)(R 3 2 1))	Balance	(W=)	Wrong	(D>)
4	((L 3 3 1)(R 2 4 1))	Left	(W>)	Correct	
5	((R 4 1 1)(L 2 3 1))	Right	(W>)	Wrong	(W×D>)
6	((L 2 2 1)(R 4 1 1))	Balance	¬(W×D>)	Correct	
7	((R 3 1 1)(L 2 1 1))	Right	(W×D>)	Correct	
8	((R 3 3 1)(L 3 2 1))	Right	(W×D>)	Correct	
9	((R 3 3 1)(L 2 4 1))	Right	(W×D>)	Correct	
10	((R 2 3 1)(L 4 1 1))	Right	(W×D>)	Correct	
11	((L 4 1 1)(R 2 2 1))	Balance	¬(W×D>)	Correct	
12	((L 3 3 1)(R 2 4 1))	Left	(W×D>)	Correct	
13	((L 2 3 1)(R 4 1 1))	Left	(W×D>)	Correct	
14	((L 2 2 1)(R 4 1 1))	Balance	¬(W×D>)	Correct	
15	((L 4 1 1)(R 2 2 1))	Balance	¬(W×D>)	Correct	

Table 8.4: The effects of instance order.

purpose of discovering the torque feature W×D and the relation W×D>. Although LIVE first believes that weight or distance are adequate for predicting the results, it soon meets a surprise (row 5) where it is forced to define the torque feature and its relation (see Section 7.3 for details).

However, such a "good" surprise can come earlier or later in the training. Whether and when LIVE will discovery the torque feature and its relation mainly depends on whether and when such surprises happen. For instance, in Table 8.5, LIVE is so lucky that the second training task is one of these surprises (when comparing to the first experiment, no explanation can be found) and the torque feature is discovered immediately. On the other hand, in Table 8.6, LIVE does not find the torque feature until task 9.

8.2.2 Experiments with Different Orders of Constructors

As we point out in Section 7.6, the weakest point of LIVE's discovering method is that the necessary constructors must be given in a more or less preferred order. The experiments in

Testing Conditions: Constructor Functions: (×, +) Constructor Relations: (>, =).					
#	State	Prediction	Reason	Result	Learned
1	((L 2 4 1)(R 3 3 1))	None	None	None	Always
2	((L 2 2 1)(R 4 1 1))	Right	Always	Wrong	¬(W×D>)
3	((L 3 1 1)(R 3 1 1))	Balance	¬(W×D>)	Correct	
4	((L 3 3 1)(R 3 2 1))	Left	(W×D>)	Correct	
5	((R 3 3 1)(L 3 2 1))	Right	(W×D>)	Correct	
6	((L 3 1 1)(R 2 1 1))	Left	(W×D>)	Correct	
7	((L 2 3 1)(R 4 1 1))	Left	(W×D>)	Correct	
8	((R 3 1 1)(L 2 1 1))	Right	(W×D>)	Correct	
9	((R 2 2 1)(L 4 1 1))	Balance	¬(W×D>)	Correct	

Table 8.5: When good surprises come earlier.

Testing Conditions: Constructor Functions: (×, +) Constructor Relations: (>, =).					
#	State	Prediction	Reason	Result	Learned
1	((L 3 2 1)(R 3 3 1))	None	None	None	Always
2	((L 3 1 1)(R 3 1 1))	Right	Always	Wrong	(D=)
3	((R 2 4 1)(L 3 3 1))	Right	(D>)	Wrong	(W>)
4	((L 3 1 1)(R 2 1 1))	Balance	(D=)	Wrong	¬(W=)
5	((L 2 3 1)(R 4 1 1))	Left	(¬W=)(D>)	Correct	
6	((L 3 3 1)(R 4 1 1))	Left	(¬W=)(D>)	Correct	
7	((R 2 3 1)(L 4 1 1))	Right	(¬W=)(D>)	Correct	
8	((R 2 5 1)(L 4 2 1))	Right	(¬W=)(D>)	Correct	
9	((L 2 2 1)(R 4 1 1))	Left	(¬W=)(D>)	Wrong	¬(W×D>)
10	((L 2 1 1)(R 3 1 1))	Right	(W×D>)	Correct	

Table 8.6: When good surprises come later.

this section show the importance of that point. We will first give LIVE the same constructors as in Table 8.6 but in a different order, then we will see how LIVE acts when a somewhat larger set of constructors is given.

Table 8.7 shows a set of experiments that conducted under the same testing conditions as that in the last section except the order of the constructor relations. Instead of giving

as (> =), the relations are now given in a reversed order, (= >).

As we can see in Table 8.7, because relation = precedes >, LIVE considers = before > when a hidden feature W×D is defined. Therefore, at row 7, W×D= is constructed instead of W×D>. As a result, LIVE continues to make mistakes on the following experiments (row 8 – 11) until another unresolvable surprise (row 11) happens and W×D> is defined.

Similarly, if we switch the order of constructor functions, as show in Table 8.8, then LIVE will construct the hidden feature W+D (at row 7) before it discovers W×D (at row 15), and consequently spend longer time for learning the torque concept.

Testing Conditions:
Constructor Functions: (×, +)
Constructor Relations: (=, >).

#	State	Prediction	Reason	Result	Learned
1	((L 3 2 1)(R 3 3 1))	None	None	None	Always
2	((L 3 1 1)(R 3 1 1))	Right	Always	Wrong	(D=)
3	((L 2 2 1)(R 4 1 1))	Left	(D>)	Wrong	(W>)
4	((L 3 3 1)(R 2 4 1))	Balance	(W>)(¬D>)	Wrong	(¬W>)
5	((L 3 1 1)(R 2 1 1))	Left	(W>)	Wrong	(D=)
6	((L 2 3 1)(R 4 1 1))	Left	(D>)	Correct	
7	((L 2 2 1)(R 4 1 1))	Left	(D>)	Wrong	(W×D=)
8	((L 3 1 1)(R 2 1 1))	Left	¬(W×D=)	Correct	
9	((L 2 1 1)(R 3 1 1))	Left	¬(W×D=)	Wrong	(W>)
10	((R 3 3 1)(L 3 2 1))	Left	¬(W×D=)	Wrong	(W=)(D>)
11	((R 3 3 1)(L 2 4 1))	Balance	(W>)(¬D>)	Wrong	(W×D>)
12	((R 2 3 1)(L 4 1 1))	Right	(W×D>)	Correct	
13	((R 2 2 1)(L 4 1 1))	Balance	¬(W×D>)	Correct	
14	((L 3 3 1)(R 2 4 1))	Left	(W×D>)	Correct	
15	((L 2 3 1)(R 4 1 1))	Left	(W×D>)	Correct	
16	((L 2 2 1)(R 4 1 1))	Balance	¬(W×D>)	Correct	
17	((L 4 1 1)(R 2 2 1))	Balance	¬(W×D>)	Correct	

Table 8.7: Experiments with different order of constructor relations.

In general, when the order of constructors is not favorable for constructing the desired features and relations first, LIVE will spend a longer time in searching. Sometimes, when the training instances do not provide the good surprises, LIVE will define many more useless hidden features and even fail to find any hidden features that can discriminate two

Testing Conditions: Constructor Functions: (+, ×) Constructor Relations: (>, =).					
#	State	Prediction	Reason	Result	Learned
1	((L 3 2 1)(R 3 3 1))	None	None	None	Always
2	((L 3 1 1)(R 3 1 1))	Right	Always	Wrong	(D=)
3	((L 2 2 1)(R 4 1 1))	Left	(D>)	Wrong	(W>)
4	((L 3 3 1)(R 2 4 1))	Balance	(W>)(¬D>)	Wrong	(¬W>)
5	((L 3 1 1)(R 2 1 1))	Left	(W>)	Wrong	(D=)
6	((L 2 3 1)(R 4 1 1))	Left	(D>)	Correct	
7	((L 2 2 1)(R 4 1 1))	Left	(D>)	Wrong	(W+D>)
8	((L 3 1 1)(R 2 1 1))	Balance	(W+D>)	Wrong	(D=)
9	((R 3 1 1)(L 2 1 1))	Right	(D=)(W+D>)	Correct	
10	((R 3 3 1)(L 3 2 1))	Left	¬(W+D>)	Wrong	(W=)(D>)
11	((R 3 3 1)(L 2 4 1))	Balance	(W>)(¬D>)	Wrong	(D>)(¬W>)
12	((L 1 4 1)(R 2 3 1))	Right	(D>)(¬W>)	Correct	
13	((R 2 2 1)(L 4 1 1))	Right	(D>)(¬W>)	Wrong	(W+D>)
14	((L 3 3 1)(R 2 4 1))	Left	¬(W+D>)	Correct	
15	((R 3 1 1)(L 2 2 1))	Right	¬(W+D>)	Wrong	(W×D>)
16	((L 2 2 1)(R 4 1 1))	Balance	¬(W×D>)	Correct	
17	((L 3 2 1)(R 2 2 1))	Left	(W×D>)	Correct	

Table 8.8: Experiments with different order of constructor functions.

undistinguishable states, as shown in step 13, Table 8.9.

Note that the only difference between Table 8.9 and Table 8.8 is the task 12. The reason LIVE fails in Table 8.9 is that when LIVE is surprised at step 15, whose state is ((x 2 2)(y 3 1)), step 12, whose state is ((x 2 3)(y 4 1)), is brought in for comparison. Unfortunately, with all the given constructors and features (including W+D and W×D), LIVE cannot find any new features that can discriminate these two states. One possible solution to this problem is to keep multiple hypothetical hidden features so that LIVE can backtrack when earlier hidden feature cannot discriminate the states that cause surprises.

Testing Conditions:
Constructor Functions: (+, ×)
Constructor Relations: (>, =).

#	State	Prediction	Reason	Result	Learned
1	((L 3 2 1)(R 3 3 1))	None	None	None	Always
2	((L 3 1 1)(R 3 1 1))	Right	Always	Wrong	(D=)
3	((L 2 2 1)(R 4 1 1))	Left	(D>)	Wrong	(W>)
4	((L 3 3 1)(R 2 4 1))	Balance	(W>)(¬D>)	Wrong	(¬W>)
5	((L 3 1 1)(R 2 1 1))	Left	(W>)	Wrong	(D=)
6	((L 2 3 1)(R 4 1 1))	Left	(D>)	Correct	
7	((L 2 2 1)(R 4 1 1))	Left	(D>)	Wrong	(W+D>)
8	((L 3 1 1)(R 2 1 1))	Balance	(W+D>)	Wrong	(D=)
9	((R 3 1 1)(L 2 1 1))	Right	(D=)(W+D>)	Correct	
10	((R 3 3 1)(L 3 2 1))	Left	¬(W+D>)	Wrong	(W=)(D>)
11	((R 3 3 1)(L 2 4 1))	Balance	(W>)(¬D>)	Wrong	(D>)(¬W>)
12	((R 2 3 1)(L 4 1 1))	Right	(D>)(¬W>)	Correct	
13	((R 2 2 1)(L 4 1 1))	Right	(D>)(¬W>)	Wrong	(W+D>)
14	((L 3 3 1)(R 2 4 1))	Left	¬(W+D>)	Correct	
15	((R 3 1 1)(L 2 2 1))	Right	¬(W+D>)	Wrong	FAIL
16	((L 2 2 1)(R 4 1 1))				
17	((L 3 2 1)(R 2 2 1))				

Table 8.9: Experiments where LIVE fails.

8.2.3 Experiments with Larger Set of Constructors

Just as the order of given constructors will effect LIVE's learning performance, so does the number of construtors (assume they are not in the favorable order). In this section, we give LIVE a larger set of constructors to try.

With the constructors given in the favorable order, shown as in Table 8.10, LIVE's learning performance is not effected. The reader can see that the experiments in Table 8.10 are given exactly as the those in Table 8.4; and even though the set of given constructors is much larger here, LIVE's learning results are consistant with Table 8.4.

Testing Conditions:
Constructor Functions: ($\times$, $+$, x^y, $-$, max, min);
Constructor Relations: ($>$, $=$, $<$, $\leq$, $\geq$).

#	State	Prediction	Reason	Result	Learned
1	((L 3 1 1)(R 3 1 1))	None	None	None	Always
2	((L 3 1 1)(R 2 1 1))	Balance	Always	Wrong	(W>)
3	((L 3 3 1)(R 3 2 1))	Balance	(W=)	Wrong	(D>)
4	((L 3 3 1)(R 2 4 1))	Left	(W>)	Correct	
5	((R 4 1 1)(L 2 3 1))	Right	(W>)	Wrong	(W×D>)
6	((L 2 2 1)(R 4 1 1))	Balance	¬(W×D>)	Correct	
7	((R 3 1 1)(L 2 1 1))	Right	(W×D>)	Correct	
8	((R 3 3 1)(L 3 2 1))	Right	(W×D>)	Correct	
9	((R 3 3 1)(L 2 4 1))	Right	(W×D>)	Correct	
10	((R 2 3 1)(L 4 1 1))	Right	(W×D>)	Correct	
11	((L 4 1 1)(R 2 2 1))	Balance	¬(W×D>)	Correct	
12	((L 3 3 1)(R 2 4 1))	Left	(W×D>)	Correct	
13	((L 2 3 1)(R 4 1 1))	Left	(W×D>)	Correct	
14	((L 2 2 1)(R 4 1 1))	Balance	¬(W×D>)	Correct	
15	((L 4 1 1)(R 2 2 1))	Balance	¬(W×D>)	Correct	

Table 8.10: Experiments with larger set of constructors.

However, if the constructors are not given in a favorable order, then LIVE is swallowed in the vast space of possible hidden features, and often fails to find any useful features. Table 8.11 shows that when the constructor functions are given in a very bad order, LIVE is desperate to find the right hidden feature. In 15 steps, the system has defined features like max(W D), W^D. Continue running cause other features like $(W^D)^{max(WD)}$ and $(W +$

$D)^{(W+D)}$ to be defined. Since $\times$ is way back in the constructor list, LIVE does not find the feature torque in these experiments.

Testing Conditions: Constructor Functions: (max, min, x^y, $-$, $+$, $\times$) Constructor Relations: ($>$, $=$, $<$, $\leq$, $\geq$).				
#	State	Prediction	Reason	Result
hline 1	((L 3 1)(R 3 1))	None	()	—
2	((L 3 1)(R 2 1))	Balance	()	Wrong
3	((L 3 3 1)(R 3 2 1))	Balance	(W)=	Wrong
4	((L 3 3 1)(R 2 4 1))	Left	(W)>	Correct
5	((L 2 3 1)(R 4 1))	Right	(W)>	Wrong
6	((L 2 2 1)(R 4 1))	Right	(maxDW)>	Wrong
7	((R 3 1)(L 2 1))	Balance	¬(maxDW)>	Wrong
8	((R 3 3 1)(L 3 2 1))	Balance	(W)=	Wrong
9	((R 3 3 1)(L 2 4 1))	Left	(maxDW)>	Wrong
10	((R 2 3 1)(L 4 1))	Balance	(W)>	Wrong
11	((R 2 2 1)(L 4 1))	Right	(D)>	Wrong
12	((L 3 3 1)(R 2 4 1))	Right	(W^D)>	Wrong
13	((L 2 3 1)(R 4 1))	Right	(W^D)>	Wrong
14	((L 2 2 1)(R 4 1))	Balance	(W)>	Correct
15	((R 2 2 1)(L 4 1))	Balance	(W)>	Correct

Table 8.11: Many constructors in a nonfavorable order.

8.3 LIVE's Discovery in Time-Dependent Environments

At the present, Mendel's plant-hybridization experiment is the only natural time-dependent environment in which LIVE has been tested and the details have been reported in Section 7.4. (The implementation of the environment is in Appendix A.3.) It is unfortunate that we have not found other natural environments that have similar properties. Also, we do not have other variaties within the pea hybridization that can be used to test LIVE further because LIVE's current ability can only deal with features that are independent in inheritance. For instance, although we can give LIVE more features of pea plants such as the length of stems and the position of the flowers, if they are independent in inheritance then the tests will be no more than a collection of the similar runs described in Section 7.4.

However, we have done some theoretical analysis based on the assumptions that the environments are finite deterministic automatons and the learner's actions have known inverse actions. For example, move-backward is a inverse of move-forward. We have designed a simpler but formal version of LIVE's algorithm and tested it on some simple domains. We have proved the correctness of the algorithm using Valiant's Probably Approximately Correct technique [57].

8.3.1 The Learner L^* and its Theoretical Analysis

Figure 8.1 is L^*, a simplified version of LIVE's algorithm. Here, we define an *environment* $\mathcal{E}$ to be a generalization of the Moore automaton. It is a 5-*tuple* $(Q, A, F, \delta, \lambda)$, where Q is the set of states, A is the set of the learner's actions, F is the set of features perceivable by the learner, δ is the transition function from $Q \times A$ to Q, and λ is a mapping from Q to F specifying what is perceivable from each state. We assume that an environments is strongly connected and all its states are distinguishable, i.e. if two states are different, then exists a sequence of actions such that the results of applying the sequence on the two states are perceivably different. We define a model M of an environment to be the same type of machine as $\mathcal{E}$, except that δ is the transition function from $Q \times A$ to Q^*, and can be possibly non-deterministic during the learning process. We say that a learner has a *perfect model* M of its environment $\mathcal{E}$, if M is isomorphic to $\mathcal{E}$. The goal of our algorithm is to identify the necessary hidden features and use them to construct a perfect model of the environment.

Compared to LIVE, the transition function δ here is the internal model and the idea of inducing hidden feature is the same except L^* uses experiments to determine the values of the hidden features defined previously. Whenever an action a_i causes a surprise in a state q_c, a new binary state variable f_{new} is defined to split q_c into two distinguishable states. A new experiment is also defined to distinguish the two states in the future. When q_c is observed or inferred again, the learner knows that it is not a single state. To see whether an observation q_c is q_c0 or q_c1, $Experiment(q_c)$ is performed by executing the action a_i and observe the outcome o. If $o \in P$, we conclude the state is q_c0, if $o = q_v$, we conclude the state is q_c1, otherwise, the learner is surprised again and another hidden feature will be defined.

The main loop of L^* randomly selects a sequence B of actions. Before an action $a_i \in B$ is performed, a set of predicted states P is made based on the observation q_c from the

Initialization: Let $A = \{a_1, a_2, \ldots, a_m\}$ be m actions, and $F = \{f_1, f_2, \ldots, f_n\}$ be n features that can be sensed. Let Q be $\{q_c\}$, where q_c is what can be sensed from the current state. Let $\delta = \{\}$ and experiments $E = \{\}$;

Repeat: Randomly select a sequence B of actions from A;
For each $a_i \in B$, do
If $\langle \delta(q_c, a_i) = R \rangle \in \delta$, then set the *prediction* $P = R$, otherwise $P = \{\}$;
Execute action a_i, and obtain an observation q_v;
While $experiment(q_v) \in E$,
then $q_v \leftarrow experiment(q_v)$ and undo the actions of the experiment;
If $P = \{\}$, then insert $\delta(q_c, a_i) = \{q_v\}$ into δ and insert q_v into Q if $q_v \notin Q$
else if $q_v \in P$, then replace $\delta(q_c, a_i) = P$ by $\delta(q_c, a_i) = \{q_v\}$ in δ
else if $q_v \notin P$ *(surprise)*
then define a new feature and update Q, E and δ as follows:

$$f_{new} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{when } \delta(q_c, a_i) = P \\ 1 & \text{when } \delta(q_c, a_i) = \{q_v\}, \end{cases}$$

$$experiment\,(q_c) \stackrel{\text{def}}{=} \begin{cases} q_c0 & \text{if executing } a_i \text{ results in a state in } P \\ q_c1 & \text{if executing } a_i \text{ results in } q_v \end{cases}$$

$$Q \Longleftarrow \text{replace } q_c \in Q \text{ by } q_c0 \text{ and } q_c1$$

$$E \Longleftarrow \text{insert } experiment\,(q_c) \text{ into } E$$

$$\delta \Longleftarrow \text{removeall } \delta(q_c, a_i) = O \text{ from } \delta$$
$$\text{insert } \delta(q_c0, a_i) = P, \text{ and } \delta(q_c1, a_i) = \{q_v\} \text{ into } \delta$$
$$\text{replaceall } \delta(o, a_j) = q_c \text{ by } \delta(o, a_j) = \{q_c0, q_c1\} \text{ where} o \in O$$

If $q_c = q_v \neq P$, then $q_c \leftarrow experiment\,(q_v)$, otherwise $q_c \leftarrow q_v$;
Until for a large number $\mathcal{K}$ of loops, we always have $\{q_v\} = P$.

Figure 8.1: The formal learning algorithm L^*.

current state and the learner's internal model δ. After the action is taken, the learner will make another observation q_v which contains all the values of the features in F. If there exists an $experiment(q_v)$ (i.e. q_v has associated some hidden features that are not directly visible), then the experiment is carried out to determine the values of those hidden features. After an experiment is done, we undo the action of the experiment by executing its inverse actions, which are known by assumption. The reason we need to undo an experiment is that

sometimes multiple experiments must be carried out to determine multiple hidden feature values, in which case the undo serves as a synchronization between experiments.

If there is no prediction, we simply remember the outcome and build a new transition in δ. If the prediction is non-deterministic (has more than one value because some states were split in the past) and the outcome is one of them, then we replace the non-deterministic predictions with the outcome so that it will be deterministic in the future. If the outcome of the action surprises us, i.e. it falls outside our prediction, then we define a hidden feature f_{new} and a new experiment $experiment(q_c)$. Note that each time a hidden feature is defined, the number of states of the internal model is increased by one. The algorithm terminates when its predictions have been consecutively correct for a long time.

Figure 8.2: The little prince environment.

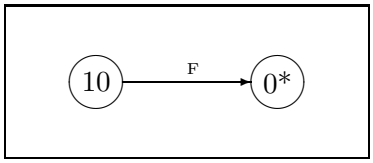
To see how L^* works, consider a simple environment called "the Little Prince" show in Figure 8.2. It is first studied in AI by Rivest and Schpire [**?**]. The prince is on a tiny planet where a rose and a volcano lie on the east and the west side. The little prince has three actions: move forward, move backward and turn around, and two senses: see the rose and see the volcano. Let us use state variables f_1 and f_2 to denote the visible states: $f_1 f_2 = 10$ means seeing the volcano; $f_1 f_2 = 11$ means seeing the rose; and $f_1 f_2 = 0\star$ means seeing nothing. The invisible features here are the prince's "facing direction" and whether his is at North pole or South pole. If we induce a new state variable f_3 together with f_2, we can represent the environment perfectly as an eight state machine shown in Figure 8.3.

If we follow the notation used in L^* and use env to represent the real state in the

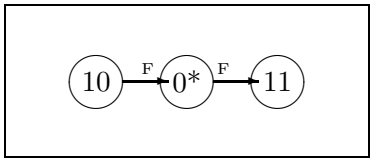
Figure 8.3: A perfect model for the little prince environment.

environment and $q_v^?$ as the observed state before experiments, then L^* learns from this environment as follows:

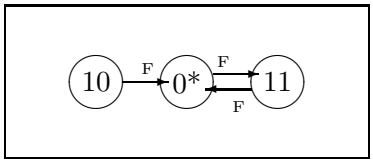
Learning begins with any of the visible states. Without lost of generality, let us assume the starting state be 101, or "seeing volcano" (10 internally).
(Step 1) q_c=10; a=F(orward); P=nil; $q_v^?$=0*; q_v=0*; env=001; Saying in English, the little prince moves forward from where he can see volcano, and he does not know what will show next (P=nil) because he never does this before. When he sees nothing, he remembers that moving forward from where he sees volcano he will see nothing.



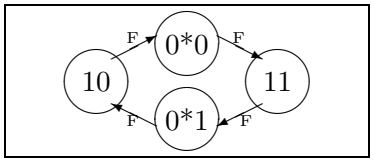
(Step 2) q_c=0*; a=F(orward); P=nil; $q_v^?$=11; q_v=11; env=111; Saying in English, the little prince moves forward from where he can see nothing, and he does not know what will show next (P=nil) because he never does this before. When he sees the rose, he remembers that moving forward from "seeing nothing" he will see the rose.



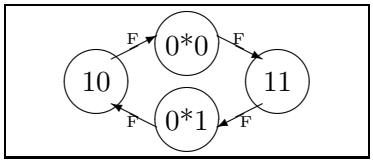
(Step 3) q_c=11; a=F; P=nil; $q_v^?$=0*; q_v=0*; env=011;



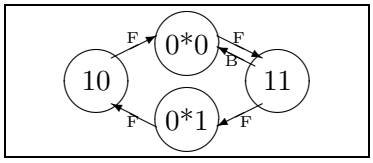
(Step 4) q_c=0*; a=F; P=11; $q_v^?$=10; q_v=10; env=101; this is a surprise because $p \neq ob$. In other words, the little prince expects to see the rose but he sees the volcano instead. At this point, we know that state 0* we see now is different from the state 0* we saw earlier, so a new state variable $f_3 = \begin{cases} 0 & \text{when apply F on 0* see 11} \\ 1 & \text{when apply F on 0* see 10} \end{cases}$ is defined to discriminate the 0* into two: 0*0 and 0*1;



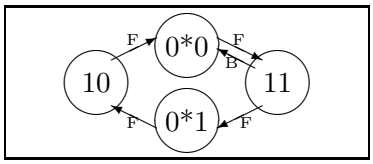
(Step 5, 6) q_c=10; a=F; $P = \{0 * 0\}$; $q_v^?$=0*; q_v=0*?; env=001; Sine we have two states look like 0* now, we must to do experiments to find out which this 0*? is. So we perform action F. Since we see 11 after F, by definition, we conclude 0*?=0*0 and q_v=11 (and env=111);



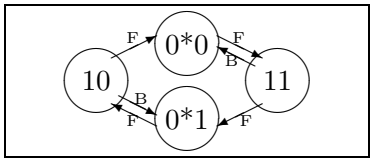
(Step 7, 8) q_c=11; a=B(ackward); P=nil; $q_v^?$=0*; q_v=0*?; q_c=001; determine 0*? by experimenting F and observing 11, conclude 0*?=0*0 and q_v=11 (and env=111);



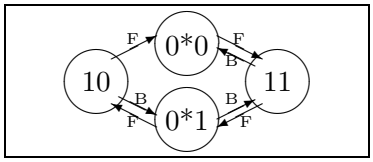
(Step 9, 10) q_c=11; a=F; P={0*1}; $q_v^?$=0*; q_v=0*?; env=011; perform F see 10 so 0*?=0*1 and q_c=10 (and env=101);



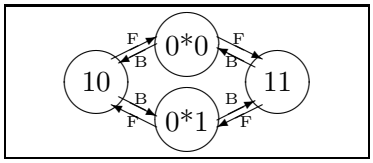
(Step 11, 12) q_c=10; a=B; P=nil; $q_v^?$=0*; q_v=0*?; env=011; perform F see 10 so 0*?=0*1 and q_c=10 (and env=101);



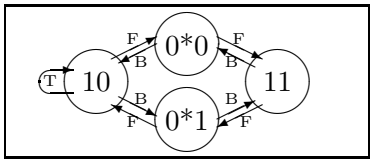
(Step 13) q_c=10; a=B; P=0*1; $q_v^?$=0*; q_v=0*1; env=011;
(Step 14) q_c=0*1; a=B; P=nil; $q_v^?$=11; q_v=11; env=111;



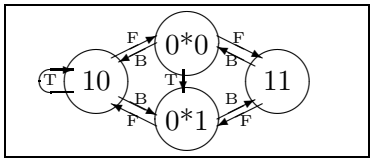
(Step 15) q_c=11; a=B; P=0*0; $q_v^?$=0*; q_v=0*0; env=001;
(Step 16) q_c=0*0; a=B; P=nil; $q_v^?$=10; q_v=10; env=101;



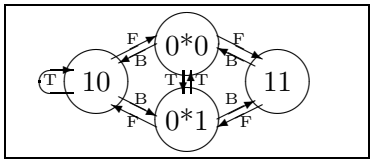
(Step 17) q_c=10; a=T(urn-around); P=nil; $q_v^?$=10; q_v=10; env=100;



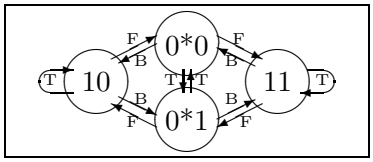
(Step 18) q_c=10; a=F; P=0*0; $q_v^?$=0*; q_v=0*0;env=010;
(Step 19, 20) q_c=0*0; a=T; P=nil; $q_v^?$=0*; q_v=0*?; env=011; perform F see 10, conclude 0*?=0*1 q_v=10 (and env=101);



(Step 21) q_c=10; a=B; P=0*1; $q_v^?$=0*; q_v=0*1; env=011;
(Step 22, 23) q_c=0*1; a=T; P=nil; $q_v^?$=0*; q_v=0*?; env=010; perform F see 11, conclude 0*?=0*0 q_v=11 (and env=110);



(Step 24) q_c=11; a=T; P=nil; $q_v^?$=11; q_v=11; env=111;



Up to this point, L^* has learned an internal model that is equivalent to the environment shown in Figure 8.3. Therefore, from now on, it will predict perfectly what are the outcomes of any action sequence although it does not know the learning has been completed. Notice that the model has only four states and is a minimum state machine equivalent to the eight states machine in Figure 8.3. The learning procedure takes 24 steps.

We now prove the correctness and termination of L^*. The proof is parallel to Anguiln's, but we don't assume that the learner is always started from some known initial state. Once, L^* is started, it has to autonomously keep track of where it is.

Correctness of L^*. Suppose that the number $\mathcal{K}$ is provided by an oracle such that any failure of prediction will be found within $\mathcal{K}$ loops of consecutively successful predictions, then when L^* terminates its model M will be perfect for its environment $\mathcal{E}$.

To see this is true, suppose L^* terminates. Then there must be a mapping ϕ from M to $\mathcal{E}$ that guarantees all predictions will be correct. Suppose now M is not isomorphic to $\mathcal{E}$ under ϕ. Let δ_e be the transition function of $\mathcal{E}$ and δ_m be the transition function of M, then there exist a state q in M and a sequence of actions $\{a_1, a_2, \ldots, a_t\}$ that $\phi(\delta_m(q, a_1 \cdots a_t)) \neq \delta_e(\phi(q), a_1 \cdots a_t)$. This is a clearly a prediction failure, which contradicts the assumption.

Termination of L^*. To see that L^* terminates, we need to prove a lemma first.

Lemma 1 *If the perfect model M of an environment $\mathcal{E}$ learned by L^* has n states, then $\mathcal{E}$ must have at least n states.*

Let $\mathcal{E} = (Q, A, F, \delta, \lambda)$, ϕ be a mapping from M to $\mathcal{E}$. Suppose s_1 and s_2 are two distinct state in M, then $\phi(s_1)$ and $\phi(s_2)$ must be distinct states in $\mathcal{E}$. To see that this is true, note that either $\lambda(\phi(s_1)) \neq \lambda(\phi(s_2))$ (they look different), in which case $\phi(s_1) \neq \phi(s_2)$ because a single state cannot have different appearances, or $\lambda(\phi(s_1)) = \lambda(\phi(s_2))$ (they look identical), in this case there must be a sequence of actions $b = a_1, a_2, \cdots a_t$ such that $\lambda[\delta(\phi(s_1), b)] \neq \lambda[\delta(\phi(s_2), b)]$ (otherwise s_1 and s_2 will never be split), so $\phi(s_1) \neq \phi(s_2)$. In either case, we have $s_1 \neq s_2 \Rightarrow \phi(s_1) \neq \phi(s_2)$, proving Lemma 1.

Now suppose that n is the number of states in the environment $\mathcal{E}$. We show that the number of distinct states in M increases monotonically up to a limit of n as L^* runs. The proof of this statement is trivial because the number of states increases by one either when a state $q_v^?$ is added into Q because there is no prediction and $q_v \notin Q$, or when an old state q_c is split into two new states q_c0 and q_c1 because $\delta(q_c, a) \neq q_v$ for some action a (when a surprise occurs). Thus, L^* correctly learns the perfect model of its environment after creating $n - 1$ new states. Now we prove the complexity of L^*.

Time analysis of L^*. Let n be the number of states of an unknown environment $\mathcal{E}$, and m be the number of the actions in A. Let $\mathcal{K}$ be the number assumed earlier, i.e. L^* terminates if its predictions are consecutively correct for $\mathcal{K}$ times. From Lemma 1 we know that L^* will increase the number of states in the model at most $n - 1$ times. Since some of the incorrect predictions (such as null predictions or correct non-deterministic predictions) do not cause the number of states to be increased, we must now find out how many such non-increase-state incorrect predictions we can have for the whole run of L^*. Suppose that M has i states at some time, before the number of states becomes $i + 1$, L^* can make at most $i \cdot m$ non-increase-state incorrect predictions since there are only i states and m actions. Thus during the period that the number of states increased from i to $i + 1$, L^* can go through the main iteration at most $i\,m\,\mathcal{K}$ times, because any surprise will be found within $\mathcal{K}$ predictions. Therefore the total number of iterations before L^* terminates is at most

$$\sum_{i=1}^{n} i\,m\,\mathcal{K} = O(n^2\,m\,\mathcal{K}).$$

In the proof above, we assumed that the termination parameter $\mathcal{K}$ is provided by an oracle and is large enough to find any possible prediction failure. But how big is this number exactly? To answer this question, we postulate a stochastic approximation of $\mathcal{K}$ analogous to that proposed by Valiant [57] and used by Angluin [3].

For the stochastic approximation, we assume that there is some probability distribution P_r on the set of all sequences over the action set A, and P_r is unknown to the learner. In addition, we assume that the learner is given, at the start of the computation, two positive number between 0 and 1, the *accuracy* ϵ and the *confidence* ξ.

Let ϵ be a positive number between 0 and 1, $\mathcal{E} = (Q, A, F, \delta, \lambda)$ and $M = (Q', A, F, \delta', \lambda)$ be the environment and its model. Let ϕ be the mapping from M to $\mathcal{E}$ created by L^*. We say that M is an ϵ-*approximation* of $\mathcal{E}$ provided that at any state $q \in Q'$:

$$\sum_w P_r(w) \leq \epsilon, \text{ where } w \in A^* \text{ and } \phi(\delta'(q, w)) \neq \delta(\phi(q), w).$$

Thus, if M is an ϵ-*approximation* of $\mathcal{E}$, the probability for L^* to find a prediction failure after a random sequence of actions is at most ϵ. Using this definition, we have the following theorem.

Theorem 1 *Let*

$$\mathcal{K} \geq \frac{1}{\epsilon}(\log \frac{1}{\xi}).$$

If L^* terminates after $\mathcal{K}$ consecutively successful predictions, the probability that the model M learned by L^* is an ϵ-approximation of $\mathcal{E}$ is at least $1 - \xi$.

To see that this holds, note that according to the definition of ϵ-approximation, the probability of having a single successful prediction is at least $1 - \epsilon$. If $\mathcal{K}$ actions are tested, the probability that all predictions are correct is at least $(1 - \epsilon)^{\mathcal{K}}$. In other words, the probability that M is not an ϵ-*approximation* of $\mathcal{E}$ is at most $(1-\epsilon)^{\mathcal{K}}$. We wish $(1-\epsilon)^{\mathcal{K}} \leq \xi$ so that we can have at least $1 - \xi$ confidence for M to be an ϵ-*approximation* of $\mathcal{E}$. Thus:

$$\begin{aligned}
(1-\epsilon)^{\mathcal{K}} &\leq \xi \\
\mathcal{K}\log(1-\epsilon) &\leq \log\xi \\
\mathcal{K}(-\epsilon) &\leq -\log\frac{1}{\xi} \\
\mathcal{K} &\geq \frac{1}{\epsilon}\log\frac{1}{\xi}.
\end{aligned}$$

We have successfully tested the learner L^* on several types of environments, including the little prince world, the *n-bit* register world [**?**], and randomly generated Moore machines. In the register world [**?**], the learner is able to read the leftmost bit of an *n-bit* register and

allow to rotate the register left or right (with wraparound) or to flip the bit it sees. Note that an *n-bit* register is a machine has 2^n states. In the randomly generated Moore machine environments, the learner is given a set of actions and allowed to sense one bit from each state. Since some environments may not have reverse actions at all, we provide the algorithm with a *remember/reset* button (equivalent to knowing reverse actions) that can be used to remember a state before an experiment and return to that state after the experiment. Such randomly generated environments are usually hard to learn for humans.

8.3.2 Issues of Learning from Time-Dependent Environments

When experimenting LIVE's learning method in the pea hybridization experiment, two new issues arised and they seem common for any time-dependent enviroments. We list them below and point out how LIVE is adjusted correspondingly.

First, as we point out in Section 7.6, in order to assimilate newly defined hidden features in a time-dependent environment, at least two functions must be determiend: the utilize function U() and the inherit function I(). To do so, examples for each of the functions must be collected, and in the experiments of pea hybridization LIVE has received helps from some given domain dependent knowledge. In particular, the system has been told that the first generation peas are purebred so that their hidden features are equal to their color. Fortunately, since such additional knowledge is used only for collecting examples, their presence will not effect LIVE's performance in other domains (of course, for different domains, different addition knowledge may required). A lesson learned from this is that domain dependent knowledge may be necessary for LIVE to discover hidden features in some time-dependent environments.

The second and maybe the most important issue is that the representation of "history" in LIVE is determined by the nature of the time (linear or branching [12]) in the enviroment. In the Tower of Hanoi environment, time is linear because a state at any time t_i is the result of the state and the action at time t_{i-1}; while in the pea hybridization experiments, the time is branching because a state at time t_i may not necessarily depend on time t_{i-1} but on its ancestors states through the inheritance line. In order to perserve the inheritance line, LIVE is adjusted to remember, for each element of history, not only the state (the parents), the action and the variable bindings, but also the result state (the children) after the action. Fourthmore, instead of remember just one history element for each rule, LIVE

now remembers a complete history back to a certain point (20, at the present). Thus, when searching for the parents of a pea, LIVE can go through the remembered history and find a hisotry element whose result state contains the pea. Of course, there are more efficient ways to remember the inheritance line, but we find the current one is the most compatible in order to have LIVE work in both tiem-dependent and time-independent environments.

8.4 LIVE in the Hand-Eye Version of Tower of Hanoi

To illustrate LIVE's performance in the Hand-Eye version of the Tower of Hanoi environment, we have included a problem solving trace in this section. The definition of the environment is in Section 2.1 and its implementation is in Appendix A.4. Since most of the variety of the Tower of Hanoi have been tested in Section 8.1, this section is not to present a large number of experiments but to show how lower level percepts and actions are used to solve the puzzle.

In this problem, LIVE is given the following initial state: As we can see, there are

```
((PEG-A CYLINDER 3.1 38.0 32.0 8.0) (PEG-B CYLINDER 3.1 -38.0 32.0 8.0) (PEG-C CYLIN-
DER 3.1 0.0 40.0 8.0) (DISK-LARGE DISK 3.0 -38.0 32.0 2.001) (DISK-MEDIUM DISK 2.5 0.0
40.0 2.001) (DISK-SMALL DISK 2.0 0.0 40.0 4.002) (HAND CYLINDER 3.5 0.0 40.0 23.0))
```

seven objects in this environment: three disks, three pegs and LIVE's hand. Objects are represented as the feature vectors (*objID shape size direction distance altitute*) as described in Section 5.2.

The goals in this problem are the following: Although the goals are given as feature

```
((DISK-LARGE DISK 3.0 -38.0 32.0 (¡ 20)) ; GOAL-L (DISK-MEDIUM DISK 2.5 -38.0 32.0 (¡ 20))
; GOAL-M (DISK-SMALL DISK 2.0 -38.0 32.0 (¡ 20))) ; GOAL-S
```

vectors, they are not real objects. Therefore, for each of these goals, LIVE has assigned a name so that the goals can be compared to the real disks for differences.

Given these goals in the initial state, LIVE first computes the differences between the goal and the current state, which yields the following list:

```
(DISK-MEDIUM DISK 2.5 -38.0 32.0 (¡ 20)) ; GOAL-M (DISK-SMALL DISK 2.0 -38.0 32.0 (¡ 20))
; GOAL-S
```

However, since LIVE does not know anything about the environment, no rule can be

found to construct any solution plans. So LIVE proposes an exploration plan and explores the environment as follows (some details of how predicates and rules are created can be found in Section 5.2):

(pick): This action picks up the DISK-SMALL which happens under the hand at the moment. Since the altitude of the disk is changed, LIVE creates a new predicate ALTITUDE= and a rule about picking up an object. (turn -10): This action changes the direction of HAND, so that LIVE creates a new predicate DIR= and a rule about turning HAND (holding an object) away from some objects (e.g. PEG-C). (slide 5): This action changes the distance of HAND, so that LIVE creates a new predicate DIST= and a rule about sliding HAND (holding an ojbect) away from some objects. (slide -5): LIVE creates a new rule about sliding HAND (holding an object) back to some objects. (turn 10): LIVE creates a new rule about turning HAND (holding an object) back to some objects. (drop): This action drops the object in HAND, so that LIVE creates a new rule about droping an object.

(turn 10): LIVE is surprised that this action does not change other objects's direction as it did in the last application, so the rule about turning is split into two rules: turn-away-with-obj and turn-away-without-obj. (slide -5): LIVE is surprised that this action does not change other objects' distance as it did in the last application, so the rule about turning is split into two rules: slide-away-with-obj and slide-away-without-obj. (slide 5): LIVE creates a new rule about sliding an empty HAND back to some objects. (turn -10): LIVE creates a new rule about turning an empty HAND back to some objects.

After these explorations, LIVE plans the following solution for the given goals: In order

(DISK-MEDIUM must DIR= to GOAL-M) ; to be achieved by turn-back-with-obj (DISK-MEDIUM must DIST= to GOAL-M) ; to be achieved by slide-back-with-obj (DISK-SMALL must DIR= to GOAL-S) ; to be achieved by turn-back-with-obj (DISK-SMALL must DIST= to GOAL-S) ; to be achieved by slide-back-with-obj

to apply turn-back-with-obj to turn DISK-MEDIUM to GOAL-M, the rule proposes the following goals: In order to pick up DISK-MEDIUM, the pick rule proposes these new goals:

(DISK-MEDIUM must ALTITUDE= to HAND) ; to be achieved by pick (DISK-MEDIUM must DIST= to GOAL-M) ; to be achieved by slide-with-obj

So LIVE first turns its HAND from GOAL-M, then turns its HAND to DISK-MEDIUM.

(HAND must not DIR= to GOAL-M) ; to be achieved by turn-without-obj (HAND must DIR= to DISK-MEDIUM) ; to be achieved by turn-without-obj

But when it does pick up DISK-MEDIUM, it is surprised that DISK-MEDIUM is not in

the hand by DISK-SMALL. Comparing this failure to a previous success, LIVE creates a new predicate SIZE¿ and finds that the reason for the failure is that there is a smaller disk at the same spot as the disk to be picked up[1].

After this surprise, a new solution plan is constructed: To reach the first goal, LIVE

(No smaller disk, such as DISK-SMALL, should ; to be achieved by turn-away-with-obj have the same direction and distance as DISK-MEDIUM) ; to turn DISK-SMALL away. (DISK-MEDIUM must ALTITUDE= to HAND) ; to be achieved by pick (DISK-MEDIUM must DIST= to GOAL-M) ; to be achieved by slide-back-with-obj (DISK-MEDIUM must DIR= GOAL-M) ; to be achieved by turn-back-with-obj (DISK-SMALL must DIST= to GOAL-S) ; to be achieved by slide-back-with-obj (DISK-SMALL must DIR= to GOAL-S) ; to be achieved by turn-back-with-obj (DISK-SMALL must ALTITUDE= to GOAL-S) ; to be achieved by drop

turns its hand (holding DISK-SMALL) away from DISK-MEDIUM. To achieve the second goal, one new goal is proposed by the rule of pick: The rule of turn-back-without-obj in

(HAND must DIR= to DISK-MEDIUM) ; to be achieved by turn-back-without-obj

turn requires the following conditions:

(DISK-SMALL must not DIST= to DISK-MEDIUM) ; to be achieved by slide-away-with-obj (DISK-SMALL must not ALTITUDE= to HAND) ; to be achieved by drop

So LIVE slides DISK-SMALL away from DISK-MEDIUM and does the drop action. Unfortunately, LIVE is surprised that after the drop action, DISK-SMALL is still in its hand! (This is because there is no thing under the hand.) Comparing this failure to an early success of drop, LIVE learns that in order to drop a disk successfully, something must be under the hand.

After this surprise, a new plan is contructed as follows: To reach the first goal, LIVE

(HAND must DIR= to DISK-MEDIUM) ; to be achieved by turn-back-with-obj (DISK-MEDIUMmust ALTITUDE= to HAND) ; to be achieved by pick (DISK-MEDIUM must DIST= to GOAL-M) ; to be achieved by slide-back-with-obj (DISK-MEDIUM must DIR= GOAL-M) ; to be achieved by turn-back-with-obj (DISK-SMALL must DIST= to GOAL-S) ; to be achieved by slide-back-with-obj (DISK-SMALL must DIR= to GOAL-S) ; to be achieved by turn-back-with-obj (DISK-SMALL must ALTITUDE= to GOAL-S) ; to be achieved by drop

turns its hand (holding DISK-SMALL) away from DISK-MEDIUM. To achieve the second goal, one new goal is proposed by the rule of pick: So LIVE turns to DISK-MEDIUM, and

[1]LIVE is lucky here to find the right reason in one step, this is because the failure and the success are in

(HAND must DIR= to DISK-MEDIUM) ; to be achieved by turn-back-without-obj

performs a pick action. Unfortunately, the pick is failed because HAND is not at the same distance as DISK-MEDIUM (not mention that DISK-SMALL is still in HAND). LIVE learns from this mistake that in order to pick up an object successfully, HAND's distance must be equal to the object's distance. A new plan is constructed as follows: To reach the first

(HAND must DIST= to DISK-MEDIUM) ; to be achieved by slide-back-with-obj (DISK-MEDIUM must ALTITUDE= to HAND) ; to be achieved by pick (DISK-MEDIUM must DIST= to GOAL-M) ; to be achieved by slide-back-with-obj (DISK-MEDIUM must DIR= GOAL-M) ; to be achieved by turn-back-with-obj (DISK-SMALL must DIST= to GOAL-S) ; to be achieved by slide-back-with-obj (DISK-SMALL must DIR= to GOAL-S) ; to be achieved by turn-back-with-obj (DISK-SMALL must ALTITUDE= to GOAL-S) ; to be achieved by drop

goal, LIVE slides its hand (holding DISK-SMALL) back to DISK-MEDIUM. To achieve the second goal, one new goal is proposed by the rule of pick:

(HAND must DIR= to DISK-MEDIUM) ; to be achieved by turn-back-without-obj

So LIVE turns its hand to DISK-MEDIUM. But, in oder to pick up DISK-MEDIUM, the pick rule now requires the following condition: Thus LIVE turns DISK-SMALL away from

(No smaller disk, such as DISK-SMALL, should have ; to be achieved by turn-away-with-obj the same direction and distance as DISK-MEDIUM) ; to turn DISK-SMALL away.

DISK-MEDIUM. However, in order to pick up DISK-MEDIUM, the pick rule also requires the following condition: In order to apply the rule turn-back-without-obj, the following

(HAND must DIR= to DISK-MEDIUM) ; to be achieved by turn-back-without-obj

conditions must be true:

(DISK-SMALL must not DIST= to DISK-MEDIUM))) ; to be achieved by slide-away-with-obj (DISK-SMALL must not ALTITUDE= to HAND))) ; to be achieved by drop

So LIVE slides DISK-SMALL away from DISK-MEDIUM. But when applying the drop rule, the following condition[2] must be true: Thus LIVE turns and slides its HAND to PEG-A,

the same state, only their bindings are different.

[2]Here, PEG-A is choosen by the "other peg" heuristic, see Section 4.4.3

(HAND must DIR= to PEG-A) ; to be achieved by turn-back-with-obj (HAND must DIST= to PEG-A) ; to be achieved by slide-back-with-obj

and drops DISK-SMALL there. Then, following the rest of the plan, LIVE turns and slides its hand back to DISK-MEDIUM's location, picks it up, and slides and turns to GOAL-M's location, and drops DISK-MEDIUM there.

After that, DISK-SMALL is moved to the location of GOAL-S in a similar way, and LIVE solves all the given goals.

Chapter 9

Conclusion

This chapter provides a summary of the research, a statement of the results, and a list of the future work.

9.1 Summary

Although learning from environments is a very common phenomenon in human learning, it is not well understood from AI point of view what the requirements are and where the difficulties lie. This thesis formalizes the phenomenon as a problem of inferring correlations between the learner's innate actions and percepts in the environment when attempting at some given goals. Based on the previous research on unifying problem solving and rule induction, a theory for learning from environments is proposed to view the instance space as a sequence of instances linked by actions, and the rule space as correlations between complex concepts and constructed actions.

An architecture for learning from environments has been constructed and it has three major components in addition to the problem solver:

- Rule Creation: constructing new and the most general rules when actions are performed during the explorations of the environment.
- Rule Revision: using the method of complementary discrimination to improve the incomplete rules when they fail to predict the outcomes of their actions.
- Hidden Feature Discovery: constructing new features based on given percepts, actions and constructors, when a surprise cannot be explained by the discrimination method.

The implemented system LIVE has demonstrated that the three components and the capability of problem solving can be integrated, and problem representations can be created through interactions with the unfamiliar environments. Applications of LIVE to the hidden-rule Tower of Hanoi environments, the child development Balance Beam experiments, and Mendel's pea-hybridization experiments have shown some encouraging results.

9.2 The Results

Following is a list of the major results of this thesis:

- Learning from environments is formalized as a problem of inferring the laws of the environment based on the learner's actions and percepts for achieving some given goals. This provides an approach to create problem spaces through interactions with the environments (in contrast to understanding written problem instructions). A theory of such learning is also proposed by advancing Simon and Lea's dual space theory to include actions as well.

- An architecture for learning from environments has been developed in the research to integrate exploration, learning, problem solving, execution, experimentation, and discovery.

- The complementary discrimination learning method developed for rule induction has shown some advantages for incrementally learning both conjunctive and disjunctive rules. Comparison to selected existing learning methods has revealed both its strength and weakness.

- Discovering hidden features from environments has been addressed as a natural extension to the complementary discrimination learning method. This method has provided a basic approach to the problem of learning with incomplete concept description language (also known as "new terms" or "constructive induction").

- The thesis has classified environments as time-dependent and time-independent, and identified the hidden features that must be defined recursively through actions as a special kind of theoretical terms in Philosophy of Science.

9.3 The Future Work

9.3.1 Improving the Learning of Search Control Knowledge

Learning from environments involves both domain knowledge and search control knowledge. However, there is a certain kind of search control knowledge that the current LIVE system is not able to learn. For example, in solving the hidden-rule Tower of Hanoi puzzle, the system will not learn the concept "the other peg" which is necessary to solve the problem efficiently. One possible approach to this problem is to have LIVE make not only "short term" predictions (predicting the outcomes of a single action) but also "long term" predictions (predicting the outcomes of a sequence of actions). Thus, when an earlier choice has caused one way to fail and the other to succeed, discrimination can be used to find the differences and form the correct problem solving strategy.

9.3.2 Learning Rules with Constructed Actions

Although the theory of learning from environments has proposed to learn rules with constructed actions in terms of primitives, the current system has not implemented that capability. The most obvious example in the Tower of Hanoi environment is to learn a macro action (a pick followed by a put) for moving one disk from one peg to another.

Complex actions can be constructed not only by compositing the actions sequentially, but by other means as well. For example, actions can be executed in parallel. To pick up an object, a robot can simultaneously move its arm and open its hand. To learn this kind of rules, LIVE's learning objectives must include not only the achievement of the given goals, but also the efficiency of such achievements. Learning should be invoked (even if there is no surprise) to make problem solving faster and more efficient. Many existing learning systems, such as SOAR [21] and PRODIGY [30], have done this task well, LIVE shall incorporate their techniques in the future.

9.3.3 Further Studies on the Discovery of Hidden Features

In this thesis, hidden feature discovery is triggered when the discrimination cannot find any differences between a success and a failure. However, to claim that two states in the real world do not have any difference is computationally intractable because there are enormous information that can be perceived. (This can be viewed as another version of the frame

problem.) Therefore, the assumption that "all the information that can be perceived by the learner are perceived" will have to be relaxed, and better mechanisms for focusing attention before discovering new features must be developed in the future.

In addition, more experiments must be done to understand the problem better. One possible domain is to predict how color blind is inherited in human, which involves not only discovering the genes of color blind but also their relations to human sexuality.

9.3.4 Theoretical Studies for Learning from the Environment

Learning from environments is closely related to the problems studied in the theoretical machine learning literature, such as Inductive Inference and System Identification. One thing we have learned from this research is that learning can have different criteria and that can have impacts on the complexity of the learning tasks. Many interesting learning problems were proved to be intractable because the criteria is to learn the whole environment. Recently, the needs for new criteria has been realized and some researcher [57, 39] have been proposed their solutions. From our point of view, it will be interesting to see how the complexity of learning will change when "solving problems" becomes the criterion of learning.

9.3.5 Building a Learning Robot

As we pointed out in the introduction, studying learning from environments is one way to bridge the gap between the high level problem solving and the low level perception and control. Thus, the next step of this research is to push down the given percepts and actions to lower levels. For example, feature vectors may be constructed from some spacial representation [33], and action commands may be replaced by the controlling actions of a mobile robot.

Bibliography

[1] J. Amsterdam. Extending the valiant learning model. In *Proceedings of the Fifth International Conference on Machine Learning*. Morgan Kaufmann, 1988.

[2] D Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39:337–350, 1978.

[3] D. Angluin. Learning regular sets from queries and counter-examples. *Information and Computation*, 75(2):87–106, November 1987.

[4] D. Angluin and C.H. Smith. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, September 1983.

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

[6] A. Barr and E.A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 3. William Kaufmann, Inc., 1982.

[7] P. Brazdil. *A Model for Error Detection and Correction*. PhD thesis, University of Edinburgh, 1981.

[8] A. Bundy, B. Silver, and D. Plummer. An analytical comparison of some rule-learning programs. *Artificial Intelligence*, 27:137–181, 1985.

[9] J.G. Carbonell and Y. Gil. Learning by experimentation. In *Proceedings of 4th International Workshop on Machine Learning*, 1987.

[10] T.G. Dietterich and R.S. Michalski. A comparative review of selected methods for learning from examples. In *Machine Learning*. Morgan Kaufmann, 1983.

[11] G.L. Drescher. A mechanism for early piagetian learning. In *Proceedings of Fifth National Conference on Artificial Intelligence*. MIT Press, 1987.

[12] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of The ACM*, 33(1), 1986.

[13] G.W. Ernst and A. Newell. GPS*: A case study in generality and problem-solving*. Academic Press, New York, 1969.

[14] B. Falkenhainer. The utility of difference-based reasoning. In *Proceedings of Sixth National Conference on Artificial Intelligence*. MIT Press, 1988.

[15] E.A. Feigenbaum and H.A. Simon. EPAM-like models of recognition and learning. *Cognitive Science*, 8, 1984.

[16] E.M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[17] J.R. Hayes and H.A. Simon. Understanding written problem instructions. In *Knowledge and Cognition*. Lawrence Erlbaum, 1974.

[18] B. Inhelder and J. Piaget. *The Growth of Logical Thinking from Childhood to Adolescence*. Basic Books, New York, 1958.

[19] B. Koslowski and J. Bruner. Learning to use a lever. *Child Development*, 43:790–799, 1972.

[20] K. Kotovsky, J.R. Hayes, and H.A. Simon. Why are some problems hard? evidence from tower of hanio. *Cognitive Psychology*, 17, 1985.

[21] J.E. Laird, P.S. Rosenbloom, and A. Newell. Chunking in SOAR: the anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[22] P. Langley. Learning search strategies through discrimination. *Int. J. Man-Machine Studies*, 18, 1983.

[23] P. Langley. Learning to search: from weak method to domain-specific heuristics. *Cognitive Science*, 9:217–260, 1985.

[24] P. Langley. A general theory of discrimination learning. In *Production System Models of Learning and Development*. MIT Press, 1987.

[25] P. Langley, H.A. Simon, and G.L. Bradshaw. Rediscovering chemistry with the bacon system. In *Machine Learning*. Morgan Kaufmann, 1983.

[26] P. Langley, H.A. Simon, G.L. Bradshaw, and Zytkow J.M. *Scientific Discovery—Computational Explorations of the Creative Processes*. The MIT Press, 1987.

[27] D. Lenat. *AM: an AI Approach to Discovery in Mathematics as Heuristic Search*. PhD thesis, Computer Science Department, Stanford University, 1976.

[28] G. Mendel. Experiments in plant-hybridization (reprinted). In J.A. Peters, editor, *Classic Papers in Genetics*. Prentice-Hall, 1865.

[29] S.N. Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, 1988.

[30] S.N. Minton, J.G. Carbonell, O. Etzioni, C.A. Knoblock, and D.R. Kuokka. Acquiring search control rules: Explanation-based learning in the prodigy system. In *Proceedings of the fourth Machine Learning Workshop*, Irvine, CA, 1987.

[31] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[32] T.M. Mitchell, P.E. Utgoff, and R.B. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning*. Morgan Kaufmann, 1983.

[33] H. Moravec. Certainty grids for sensor fusion in mobile robots. *AI Magazine*, Summer 1988.

[34] E. Nagel. *John Stuart Mill's Philosophy of the Scientific Method*. Hafner, 1950.

[35] A. Newell. *Unified Theories of Cognition, William James Lecture*. Harvard University, 1987.

[36] J. Piaget. *The Origins of Intelligence in Children*. Norton, New York, 1952.

[37] J. Piaget. *The Construction of Reality in the Child*. Ballantine, New York, 1954.

[38] J.R. Quinlan. Generating production rules from decision trees. In *Proceedings of 10th IJCAI*, 1987.

[39] R.L. Rivest and R. Sloan. Learning complicated concepts reliably and usefully. In *Proceedings of Sixth National Conference on Artificial Intelligence*. Morgan Kaufmann, 1988.

[40] J.C. Schlimmer. *Concept Acquisition through Representation Adjustment*. PhD thesis, University of California, Irvine, 1987.

[41] E. Shapiro. An algorithm that infers theories from facts. In *Proceedings Seventh IJCAI*, pages 446–451. IJCAI, Vancouver, BC, 1981.

[42] W.M. Shen. An algorithm that infers novel features from experiments. Technical Report ???, Carnegie Mellon University, xxx 1988.

[43] W.M. Shen. Functional transformation in AI discovery systems. *Artificial Intelligence*, 41(3):257–272, 1990.

[44] Y. Shoham. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence*. PhD thesis, Yale University, 1986.

[45] J. Shrager. *Instructionless Learning: Discovery of the Mental Model of a Complex Device*. PhD thesis, Carnegie Mellon University, 1985.

[46] R.S. Siegler. How knowledge influences learning. *American Scientist*, 71:631–638, 1983.

[47] H. A. Simon. *The Sciences of the Artificial*. MIT Press, 1981.

[48] H.A. Simon. The axiomatization of physical theories. *Philosophy of Science*, 37:16–26, 1970.

[49] H.A. Simon. The functional equivalence of problem solving skills. *Cognitive Psychology*, 7, 1975.

[50] H.A. Simon and J.R. Hayes. The understanding process: Problem isomorphs. *Cognitive Psychology*, 8, 1976.

[51] H.A. Simon and G. Lea. Problem solving and rule induction: A unified view. In *Knowledge and Cognition*. Erlbaum, Hillsdale, N.J., 1974.

[52] D.M. Steier, J.E. Laird, A. Newell, et al. Varieties of learning in SOAR. In *Proceedings of the Fourth International Machine Learning Workshop*, Irvine, CA, 1987.

[53] P. Suppes. *Introduction to Logic*. Van Nostrand, Princeton, New Jersey, 1957.

[54] G.J. Sussman. *A Computer Model of Skill Acquisition*. American Elsevier, New York, 1975.

[55] A. Tarski. *Some Methodological Investigations on the Definability of Concepts*, chapter 10. The Clarendon Press, Oxford, 1956.

[56] P.E. Utgoff. Shift of bias for inductive concept learning. In *Machine Learning II*. Morgan Kaufmann, 1986.

[57] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[58] S.A. Vere. Multilevel counterfactuals for generalizations of relational concepts and productions. *Artificial Intelligence*, 14, 1980.

[59] D.A. Waterman. Adaptive production systems. In *Proceedings of IJCAI-83*, 1975.

[60] P.H. Winston. Learning structural descriptions from examples. In *The psychology of computer vision*. MacGraw-Hill, 1975.

Appendix A

The Implementations of the Environments

A.1 The RPMA Environment

```
;;; This is a simulation of the RPMA environment. ;;; Only the symbols that listed in the export section can be used outside.
(provide 'environment) (in-package 'environment) (export '(env-initiate observe *time-depen* *relations-to-predict* *relations* on gr inhand disk1 disk2 disk3 peg1 peg2 peg3 *actions* mpick mput *cnst-funs* *cnst-rels* constructor fe-index *feature-counter*))
(defvar *actions* '((mpick d p) (mput d p))) (defvar *relations* '(on gr inhand)) (defvar *cnst-funs* '()) (defvar *cnst-rels* '()) (defvar *time-depen* t) (defvar *relations-to-predict* '()) (defvar *feature-counter* 0) (defvar *world* nil)
```

```
;;; ******************** Initiate and Observe ********************
(defun env-initiate () (setq *actions* '((mpick d p) (mput d p)) *relations* '(on gr inhand) *cnst-
funs* '() *cnst-rels* '() *time-depen* t *relations-to-predict* '() *feature-counter* 0) (setf (get 'disk1
'user::external-id) 'disk1) (setf (get 'disk2 'user::external-id) 'disk2) (setf (get 'disk3 'user::external-
id) 'disk3) (setf (get 'peg1 'user::external-id) 'peg1) (setf (get 'peg2 'user::external-id) 'peg2) (setf
(get 'peg3 'user::external-id) 'peg3) (setq *world* (copy-tree (list '(disk1 disk 1 1) '(disk2 disk 2 1)
'(disk3 disk 3 1) '(peg1 peg 0 1) '(peg2 peg 0 2) '(peg3 peg 0 3)))))
(defun observe () (list '(disk1) '(disk2) '(disk3) '(peg1) '(peg2) '(peg3) '(gr disk2 disk1) '(gr disk3
disk2) '(gr disk3 disk1) (on-or-inh (copy-list (elt *world* 0))) (on-or-inh (copy-list (elt *world* 1)))
(on-or-inh (copy-list (elt *world* 2)))))
;;; ******************** Pre-Defined Relations ********************
(defun on (x y) nil)
(setf (get 'on 'constructor) '=) (setf (get 'on 'fe-index) 0)
(defun gr (x y) nil)
(setf (get 'gr 'constructor) '¿) (setf (get 'gr 'fe-index) 0)
(defun inhand (x y) nil)
(setf (get 'inhand 'constructor) 'eql) (setf (get 'inhand 'fe-index) 0)
;;; ******************** Actions ********************
(defun mpick (disk peg) (format t " (let ((d (indexof disk)) (p (indexof peg))) (if (and (= (elt (elt
*world* p) 3) (elt (elt *world* d) 3)) (nothing-in-hand) (no-block d p)) (setf (elt (elt *world* d) 3)
0))))
(defun mput (disk peg) (format t " (let ((d (indexof disk)) (p (indexof peg))) (if (and (= 0 (elt (elt
*world* d) 3)) (no-block d p)) (setf (elt (elt *world* d) 3) (elt (elt *world* p) 3)))))
```

```
;;; ******************** Misc Functions ********************
(defun on-or-inh (d) (cond ((= 0 (elt d 3)) (list 'inhand (car d))) ((= 1 (elt d 3)) (list 'on (car d)
'peg1)) ((= 2 (elt d 3)) (list 'on (car d) 'peg2)) ((= 3 (elt d 3)) (list 'on (car d) 'peg3))))
(defun no-block (d p) (let ((peg (elt (elt *world* p) 3)) (size (elt (elt *world* d) 2))) (dolist (f
*world* t) (if (and (eql 'disk (elt f 1)) (= peg (elt f 3)) (¿ size (elt f 2))) (return nil)))))
(defun nothing-in-hand () (and (/= 0 (elt (elt *world* 0) 3)) (/= 0 (elt (elt *world* 1) 3)) (/= 0
(elt (elt *world* 2) 3))))
(defun indexof (dp) (case dp ('disk1 0) ('disk2 1) ('disk3 2) ('peg1 3) ('peg2 4) ('peg3 5)))
```

A.2 The Balance Beam Environment

```
;;; This is a simulation of the Balance Beam environment. ;;; Only the symbols that listed in the export section can be used outside.
(provide 'environment) (in-package 'environment) (export '(env-initiate observe *time-depen* *relations-to-predict* *relations* wg= wg¿ dist= dist¿ alt= alt¿ *actions* release *cnst-funs* *cnst-rels* constructor fe-index ; these two are properties of relations. *feature-counter* ok-to-proceed receive-updated-world))
(defvar *actions* '((release))) (defvar *relations* '(wg= wg¿ dist= dist¿ alt= alt¿)) (defvar *cnst-funs* '(* +)) (defvar *cnst-rels* '(¿ =))
(defvar *time-depen* nil) (defvar *relations-to-predict* '(alt= alt¿)) (defvar *feature-counter* 3)
(defvar *world* nil) (defvar *trial* -1) ; 1+ every time (observe) is called.
```

```
(defun env-initiate () (setq *actions* '((release)) *relations* '(wg= wg¿ dist= dist¿ alt= alt¿) *cnst-funs* '(* +) *cnst-rels* '(¿ =) *time-depen* nil *relations-to-predict* '(alt= alt¿) *feature-counter* 3 *trial* -1) (setq *world* (list '((L 3 1 1) (R 3 1 1)) '((L 3 1 1) (R 3 1 1)) '((L 3 1 1) (R 2 1 1)) '((L 3 1 0) (R 2 1 2)) '((L 3 3 1) (R 3 2 1)) '((L 3 3 0) (R 3 2 2)) '((L 3 3 1) (R 2 4 1)) '((L 3 3 0) (R 2 4 2)) '((L 2 3 1) (R 4 1 1)) '((L 2 3 0) (R 4 1 2)) '((L 2 2 1) (R 4 1 1)) '((L 2 2 1) (R 4 1 1)) '((R 3 1 1) (L 2 1 1)) '((R 3 1 0) (L 2 1 2)) '((R 3 3 1) (L 3 2 1)) '((R 3 3 0) (L 3 2 2)) '((R 3 3 1) (L 2 4 1)) '((R 3 3 0) (L 2 4 2)) '((R 2 3 1) (L 4 1 1)) '((R 2 3 0) (L 4 1 2)) '((R 2 2 1) (L 4 1 1)) '((R 2 2 1) (L 4 1 1)) '((L 3 3 1) (R 2 4 1)) '((L 3 3 0) (R 2 4 2)) '((L 2 3 1) (R 4 1 1)) '((L 2 3 0) (R 4 1 2)) '((L 2 2 1) (R 4 1 1)) '((L 2 2 1) (R 4 1 1)) '((L 2 2 1) (R 4 1 1)) '((L 2 2 1) (R 4 1 1)))))
(defun observe () (setq *trial* (1+ *trial*)) (copy-tree (elt *world* *trial*)))
```

```
;;; ******************** Pre-Defined Relations ********************
(defun wg= (x y) (unless (eql (car x) (car y)) (if (eql (elt x 1) (elt y 1)) (list 'wg= (car x) (car y)))))
(setf (get 'wg= 'constructor) '=) (setf (get 'wg= 'fe-index) 1)
(defun wg¿ (x y) (unless (eql (car x) (car y)) (if (¿ (cadr x) (cadr y)) (list 'wg¿ (car x) (car y)))))
(setf (get 'wg¿ 'constructor) '¿) (setf (get 'wg¿ 'fe-index) 1)
(defun dist= (x y) (unless (eql (car x) (car y)) (if (eql (elt x 2) (elt y 2)) (list 'dist= (car x) (car y)))))
(setf (get 'dist= 'constructor) '=) (setf (get 'dist= 'fe-index) 2)
(defun dist¿ (x y) (unless (eql (car x) (car y)) (if (¿ (elt x 2) (elt y 2)) (list 'dist¿ (car x) (car y)))))
(setf (get 'dist¿ 'constructor) '¿) (setf (get 'dist¿ 'fe-index) 2)
(defun alt= (x y) (unless (eql (car x) (car y)) (if (eql (elt x 3) (elt y 3)) (list 'alt= (car x) (car y)))))
(setf (get 'alt= 'constructor) '=) (setf (get 'alt= 'fe-index) 3)
(defun alt¿ (x y) (unless (eql (car x) (car y)) (if (¿ (elt x 3) (elt y 3)) (list 'alt¿ (car x) (car y)))))
(setf (get 'alt¿ 'constructor) '¿) (setf (get 'alt¿ 'fe-index) 3)
;;; ******************** Actions ********************
(defun release ())
```

A.3 The Pea-Hybridization Environment

;;; This is a simulation of Mendel's experiments. ;;; Only the symbols that listed in the export section can be used outside.
(provide 'environment) (in-package 'environment) (export '(env-initiate observe *time-depen* *discret-events* *purebreds* *predict-only* *relations-to-predict* *relations* *cnst-rels* *feature-counter* constructor fe-index ; these two are properties of relations. *actions* af *cnst-funs* e-max e-min e-distr))
(defvar *actions* '((af x y))) (defvar *relations* nil) (defvar *cnst-funs* '(e-max e-min e-distr)) (defvar *cnst-rels* '(= ¿)) (defvar *purebreds* ())
(defvar *time-depen* t) (defvar *predict-only* t) (defvar *discret-events* t) (defvar *relations-to-predict* nil) (defvar *feature-counter* 1)
(defvar *world* nil) (defvar *trial* -1) ; 1+ every time (observe) is called.
(defun observe () (setq *trial* (1+ *trial*)) (mapcar #'(lambda (x) (subseq x 0 2)) (copy-tree (elt *world* *trial*))))
;;; ******************** Actions ********************
(defun af (x y))
;;; ******************** Misc Functions ********************
(defun e-distr (a b) (if (and (listp a) (listp b) (= (length a) 2) (= (length b) 2)) `((,(car a) ,(car b)) (,(car a) ,(cadr b)) (,(cadr a) ,(car b)) (,(cadr a) ,(cadr b)))))
(defun e-max (&rest a) (if (every #'numberp a) (apply #'max a)))
(defun e-min (&rest a) (if (every #'numberp a) (apply #'min a)))

(defun env-initiate () (setq *actions* '((af x y)) *relations* nil *cnst-funs* '(e-max e-min e-distr) *cnst-rels* '(= ¿) *time-depen* t *predict-only* t *discret-events* t *relations-to-predict* nil *feature-counter* 1 *trial* -1 *purebreds* '(P1 P2 P3 P4 P5 P6 P7 P8)) (setq *world* (list '((P1 0 0 0) (P2 0 0 0)) '((P9 0 0 0) (P10 0 0 0) (P11 0 0 0) (P12 0 0 0)) '((P3 1 1 1) (P4 1 1 1)) '((P13 1 1 1) (P14 1 1 1) (P15 1 1 1) (P16 1 1 1)) '((P5 0 0 0) (P6 1 1 1)) '((P17 1 0 1) (P18 1 0 1) (P19 1 0 1) (P20 1 0 1)) '((P7 1 1 1) (P8 0 0 0)) '((P21 1 1 0) (P22 1 1 0) (P23 1 1 0) (P24 1 1 0))
'((P9 0 0 0) (P10 0 0 0)) '((P25 0 0 0) (P26 0 0 0) (P27 0 0 0) (P28 0 0 0)) '((P13 1 1 1) (P14 1 1 1)) '((P29 1 1 1) (P30 1 1 1) (P31 1 1 1) (P32 1 1 1)) '((P17 1 0 1) (P18 1 0 1)) '((P33 0 0 0) (P34 1 0 1) (P35 1 1 0) (P36 1 1 1)) '((P21 1 1 0) (P22 1 1 0)) '((P37 1 1 1) (P38 1 1 0) (P39 1 0 1) (P40 0 0 0))
'((P33 0 0 0) (P40 0 0 0)) '((P41 0 0 0) (P42 0 0 0) (P43 0 0 0) (P44 0 0 0)) '((P34 1 0 1) (P38 1 1 0)) '((P45 1 0 1) (P46 0 0 0) (P47 1 1 1) (P48 1 1 0)) '((P35 1 1 0) (P39 1 0 1)) '((P49 1 1 0) (P50 1 1 1) (P51 0 0 0) (P52 1 0 1)) '((P36 1 1 1) (P37 1 1 1)) '((P53 1 1 1) (P54 1 1 1) (P55 1 1 1) (P56 1 1 1)))))

A.4 The Hand-Eye Version of the Tower of Hanoi Environment

;;; This is a simulated world model of Tower of Hanoi environment. ;;; Only the symbols that listed in the export section can be used outside.
(provide 'environment) (in-package 'environment) (export '(env-initiate observe *time-depen* peg disk *relations* *actions* turn slide pick drop *cnst-funs* *cnst-rels* constructor fe-index *feature-counter*))
(defvar *actions* '((turn g) (slide d) (pick) (drop))) (defvar *relations* '()) (defvar *cnst-funs* '()) (defvar *cnst-rels* '()) (defvar *time-depen* t) (defvar *feature-counter* 6) (defvar *world* nil)
(defun env-initiate () (setq *world* (copy-tree '((0 cylinder 3.5 0.0 40.0 23.0) ; the hand (1 disk 2.0 0.0 40.0 4.002) ; disk1 (2 disk 2.5 0.0 40.0 2.001) ; disk2 (3 disk 3.0 38.0 32.0 2.001) ; disk3 (4 peg 3.1 0.0 40.0 8.0) ; peg-c (5 peg 3.1 -38.0 32.0 8.0) ; peg-b (6 peg 3.1 38.0 32.0 8.0))))) ; peg-a
(defun observe () (copy-tree *world*))

;;; ************************ Actions ************************
(defun turn (theta) (format t " (setf (fourth (car *world*)) (+ theta (fourth (car *world*)))) (let ((id (thing-in-hand))) (if id (setf (fourth (elt *world* id)) (+ theta (fourth (elt *world* id))))))))
(defun slide (dis) (format t " (setf (fifth (car *world*)) (+ dis (fifth (car *world*)))) (let ((id (thing-in-hand))) (if id (setf (fifth (elt *world* id)) (+ dis (fifth (elt *world* id))))))))
(defun pick () (format t " (unless (thing-in-hand) (let ((th (top-disk-under-hand))) (if th (setf (sixth (elt *world* th)) 23.0)))))
(defun drop () ; to drop onto a peg that has no smaller disks. (format t " (let (in-hand under alt) (when (and (setq in-hand (thing-in-hand)) (peg-under-hand)) (multiple-value-setq (under alt) (top-disk-under-hand)) (if (null under) (setf (sixth (elt *world* in-hand)) 2.001) (if (¡ (third (elt *world* in-hand)) (third (elt *world* under))) (setf (sixth (elt *world* in-hand)) (+ alt 2.001)))))))

;;; ************************ Misc Functions ************************
(defun thing-in-hand () (position 23.0 (list* nil (cdr *world*)) :key #'sixth))
(defun peg-under-hand () (let ((hand-dir (fourth (car *world*))) (hand-dis (fifth (car *world*))) (temp nil)) (do ((i 1 (1+ i))) ((¿ i 6) nil) (setq temp (elt *world* i)) (if (and (eq (second temp) 'peg) (= (third temp) 3.1) (= hand-dir (fourth temp)) (= hand-dis (fifth temp))) (return t)))))
(defun top-disk-under-hand () (let ((hand-dir (fourth (car *world*))) (hand-dis (fifth (car *world*))) (top-altitude -500) (top-disk nil)) (do ((i 1 (1+ i))) ((¿ i 6) (values top-disk top-altitude)) (if (and (eq (second (elt *world* i)) 'disk) (= hand-dir (fourth (elt *world* i))) (= hand-dis (fifth (elt *world* i))) (¿ (sixth (elt *world* i)) top-altitude) (¡ (sixth (elt *world* i)) 20)) ; not in hand (setq top-disk i top-altitude (sixth (elt *world* i)))))))

Appendix B

LIVE's Running Trace in the RPMA Environment

* (live)

****
****** ****** ****** Learning from Environment Based on Percepts and Actions ****** ****** ******
****

The Initial Rules: ( )
Here is what I see:
(ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1)
The Goals Given:
((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))
Planning for the remaining goals:
((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))
Differences towards the goal (ON DISK3 PEG3) are: (ON DISK3 PEG3)
Find rules for the differences and sort them ...
To achieve (ON DISK3 PEG3), Rules NIL can be used.

————— Think or Act (1) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (ON DISK3 PEG3)
Proposing explore plan ... ((MPICK DISK1 PEG1) (MPUT DISK1 PEG2)) Given the action (MPICK DISK1 PEG1), rules NIL can predict. Useful-Rule-Instance: NIL
Decide to =====¿ (MPICK DISK1 PEG1)
Pick DISK1 from PEG1. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Creating a new rule ...
**** BEGIN RULES ****
Index: 0 Sibling: NIL State: ((ON =X2018 =X2019) (NOTEX ((INHAND =X2018))) (=X2019) (=X2018)) Action: (MPICK =X2018 =X2019) Predict: ((INHAND =X2018) (NOTEX ((ON =X2018 =X2019))) (=X2019) (=X2018))

**** END RULES ****

Given the action (MPUT DISK1 PEG2), rules NIL can predict. Useful-Rule-Instance: NIL

Decide to =====¿ (MPUT DISK1 PEG2)

Put DISK1 on PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Creating a new rule ...

**** BEGIN RULES ****

Index: 0 Sibling: NIL State: ((ON =X2018 =X2019) (NOTEX ((INHAND =X2018))) (=X2019) (=X2018)) Action: (MPICK =X2018 =X2019) Predict: ((INHAND =X2018) (NOTEX ((ON =X2018 =X2019))) (=X2019) (=X2018))

Index: 1 Sibling: NIL State: ((INHAND =X2033) (NOTEX ((ON =X2033 =X2034))) (=X2034) (=X2033)) Action: (MPUT =X2033 =X2034) Predict: ((ON =X2033 =X2034) (NOTEX ((INHAND =X2033))) (=X2034) (=X2033))

**** END RULES ****

Planning for the remaining goals:

((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))

Differences towards the goal (ON DISK1 PEG1) are: (ON DISK1 PEG1)

Differences towards the goal (ON DISK3 PEG3) are: (ON DISK3 PEG3)

Find rules for the differences and sort them ...

To achieve (ON DISK1 PEG1), Rules (1) can be used. To achieve (ON DISK3 PEG3), Rules (1) can be used.

————— Think or Act (2) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK1 PEG1)

Useful-Rule-Instance: State: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2033 . DISK1) (=X2034 . PEG1))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((INHAND DISK1))

Find rules for the differences and sort them ...

To achieve (INHAND DISK1), Rules (0) can be used.

New plan for the differences: (INHAND DISK1) 0

————— Think or Act (3) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK1)

Rulei's action has free variable: =X2019 is bound to PEG2

Useful-Rule-Instance: State: ((ON DISK1 PEG2) (NOTEX ((INHAND DISK1))) (PEG2) (DISK1)) Action: (MPICK DISK1 PEG2) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG2))) (PEG2) (DISK1)) Index: 0 Bindings: ((=X2019 . PEG2) (=X2018 . DISK1))

Decide to =====¿ (MPICK DISK1 PEG2)

Pick DISK1 from PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (INHAND DISK1) accomplished!

————— Think or Act (4) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK1 PEG1)

Useful-Rule-Instance: State: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2033 . DISK1) (=X2034 . PEG1))

Decide to =====¿ (MPUT DISK1 PEG1)

Put DISK1 on PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK1 PEG1) accomplished!

————— Think or Act (5) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK3 PEG3)

Useful-Rule-Instance: State: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG3))) (PEG3) (DISK3)) Action: (MPUT DISK3 PEG3) Predict: ((ON DISK3 PEG3) (NOTEX ((INHAND DISK3))) (PEG3) (DISK3)) Index: 1 Bindings: ((=X2033 . DISK3) (=X2034 . PEG3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((INHAND DISK3))

Find rules for the differences and sort them ...

To achieve (INHAND DISK3), Rules (0) can be used.

New plan for the differences: (INHAND DISK3) 0

————— Think or Act (6) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Rulei's action has free variable: =X2019 is bound to PEG1

Useful-Rule-Instance: State: ((ON DISK3 PEG1) (NOTEX ((INHAND DISK3))) (PEG1) (DISK3)) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ (MPICK DISK3 PEG1)

Pick DISK3 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Wrong prediction: (((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) 0 ((=X2019 . PEG1) (=X2018 . DISK3)))

Explain the surprise by comparing with the last succeed application viewed by the current relations

(((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1)) (NIL NIL) (MPICK DISK1 PEG2) ((=X2019 . PEG2) (=X2018 . DISK1)))

Explanation 1: Relation differences inside bindings.

Rel-Then: NIL Rel-Now: NIL

Creating relations on the bound things .....

Explanation 2: Middle circle relation differences.

Rel-Then: ((GR DISK2 DISK1) (GR DISK3 DISK1)) Rel-Now: ((GR DISK3 DISK2) (GR DISK3 DISK1) (ON DISK1 PEG1) (ON DISK2 PEG1))

WHY: (((ON =X2112 =X2019)))

Spliting rule 0 with WHY.

**** BEGIN RULES ****

Index: 0 Sibling: 2 State: ((=X2018) (=X2019) (ON =X2018 =X2019) (NOTEX ((ON =X2112 =X2019))) (NOTEX ((INHAND =X2018)))) Action: (MPICK =X2018 =X2019) Predict: ((INHAND =X2018) (NOTEX ((ON =X2018 =X2019))) (=X2019) (=X2018))

Index: 1 Sibling: NIL State: ((INHAND =X2033) (NOTEX ((ON =X2033 =X2034))) (=X2034) (=X2033)) Action: (MPUT =X2033 =X2034) Predict: ((ON =X2033 =X2034) (NOTEX ((INHAND =X2033))) (=X2034) (=X2033))

Index: 2 Sibling: 0 State: ((NOTEX ((NOTEX ((ON =X2112 =X2019))))) (ON =X2018 =X2019) (NOTEX ((INHAND =X2018))) (=X2019) (=X2018)) Action: (MPICK =X2018 =X2019) Predict: ((ON =X2018 =X2019))

**** END RULES ****

Planning for the remaining goals:

((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))

Differences towards the goal (ON DISK3 PEG3) are: (ON DISK3 PEG3)

Find rules for the differences and sort them ...

To achieve (ON DISK3 PEG3), Rules (1) can be used.

————— Think or Act (7) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK3 PEG3)

Useful-Rule-Instance: State: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG3))) (PEG3) (DISK3)) Action: (MPUT DISK3 PEG3) Predict: ((ON DISK3 PEG3) (NOTEX ((INHAND DISK3))) (PEG3) (DISK3)) Index: 1 Bindings: ((=X2033 . DISK3) (=X2034 . PEG3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((INHAND DISK3))

Find rules for the differences and sort them ...

To achieve (INHAND DISK3), Rules (0) can be used.

New plan for the differences: (INHAND DISK3) 0

————— Think or Act (8) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Rulei's action has free variable: =X2019 is bound to PEG1

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((ON =X2112 PEG1))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((ON DISK2 PEG1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((ON DISK2 PEG1))), Rules (0) can be used.

New plan for the differences: (NOTEX ((ON DISK2 PEG1))) 0

————— Think or Act (9) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((ON DISK2 PEG1)))

Useful-Rule-Instance: State: ((DISK2) (PEG1) (ON DISK2 PEG1) (NOTEX ((ON =X2112 PEG1))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG1) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Index: 0 Bindings: ((=X2018 . DISK2) (=X2019 . PEG1))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((ON DISK3 PEG1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((ON DISK3 PEG1))), Rules (0) can be used.

New plan for the differences: (NOTEX ((ON DISK3 PEG1))) 0

————— Think or Act (10) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((ON DISK3 PEG1)))

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((ON =X2112 PEG1))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2018 . DISK3) (=X2019 . PEG1))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((ON DISK2 PEG1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((ON DISK2 PEG1))), Rules (0) can be used.

New plan for the differences: (NOTEX ((ON DISK2 PEG1))) 0

————— Think or Act (11) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((ON DISK2 PEG1)))

Useful-Rule-Instance: State: ((DISK2) (PEG1) (ON DISK2 PEG1) (NOTEX ((ON =X2112 PEG1))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG1) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Index: 0 Bindings: ((=X2018 . DISK2) (=X2019 . PEG1))

Decide to =====¿ Propose new subgoals.

Rule 0 causes SUBGOAL-LOOP Generating an experiment for it.

Generating an Experiment.

————— Think or Act (12) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (EXPERIMENT)

Useful-Rule-Instance: State: ((NOTEX ((NOTEX ((ON DISK3 PEG1))))) (ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Action: (MPICK DISK1 PEG1) Predict: ((ON DISK1 PEG1)) Index: 2 Bindings: ((=X2018 . DISK1) #1=(=X2019 . PEG1) #1# (=X2112 . DISK3))

Decide to =====¿ (MPICK DISK1 PEG1)

Pick DISK1 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Wrong prediction: (((ON DISK1 PEG1)) 2 ((=X2018 . DISK1) #1=(=X2019 . PEG1) #1# (=X2112 . DISK3)))

Explain the surprise by comparing with the last succeed application viewed by the current relations

(((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1)) (NIL NIL) (MPICK DISK3 PEG1) ((=X2112 . DISK2) (=X2019 . PEG1) (=X2018 . DISK3)))

Explanation 1: Relation differences inside bindings.

Rel-Then: NIL Rel-Now: NIL

Creating relations on the bound things .....

Explanation 2: Middle circle relation differences.

Rel-Then: ((GR DISK3 DISK2) (GR DISK3 DISK1) (ON DISK1 PEG1) (ON DISK2 PEG1)) Rel-Now: ((GR DISK2 DISK1) (GR DISK3 DISK1) (ON DISK2 PEG1) (ON DISK3 PEG1))

WHY: ((NOTEX ((GR =X2018 =X2112) (ON =X2112 =X2019))))

Update the wrong rule and complement its sibling.

**** BEGIN RULES ****

Index: 0 Sibling: 2 State: ((NOTEX ((GR =X2018 =X2112) (ON =X2112 =X2019))) (ON =X2018 =X2019) (NOTEX ((INHAND =X2018))) (=X2019) (=X2018)) Action: (MPICK =X2018 =X2019) Predict: ((INHAND =X2018) (NOTEX ((ON =X2018 =X2019))) (=X2019) (=X2018))

Index: 1 Sibling: NIL State: ((INHAND =X2033) (NOTEX ((ON =X2033 =X2034))) (=X2034) (=X2033)) Action: (MPUT =X2033 =X2034) Predict: ((ON =X2033 =X2034) (NOTEX ((INHAND =X2033))) (=X2034) (=X2033))

Index: 2 Sibling: 0 State: ((ON =X2112 =X2019) (GR =X2018 =X2112) (=X2018) (=X2019) (ON =X2018 =X2019) (NOTEX ((INHAND =X2018)))) Action: (MPICK =X2018 =X2019) Predict: ((ON =X2018 =X2019))

**** END RULES ****

Experiment done.

Planning for the remaining goals:

((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))

Differences towards the goal (ON DISK1 PEG1) are: (ON DISK1 PEG1)

Differences towards the goal (ON DISK3 PEG3) are: (ON DISK3 PEG3)

Find rules for the differences and sort them ...

To achieve (ON DISK1 PEG1), Rules (1) can be used. To achieve (ON DISK3 PEG3), Rules (1) can be used.

————— Think or Act (13) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK1 PEG1)

Useful-Rule-Instance: State: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2033 . DISK1) (=X2034 . PEG1))

Decide to =====¿ (MPUT DISK1 PEG1)

Put DISK1 on PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK1 PEG1) accomplished!

————— Think or Act (14) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK3 PEG3)

Useful-Rule-Instance: State: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG3))) (PEG3) (DISK3)) Action: (MPUT DISK3 PEG3) Predict: ((ON DISK3 PEG3) (NOTEX ((INHAND DISK3))) (PEG3) (DISK3)) Index: 1 Bindings: ((=X2033 . DISK3) (=X2034 . PEG3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((INHAND DISK3))

Find rules for the differences and sort them ...

To achieve (INHAND DISK3), Rules (0) can be used.

New plan for the differences: (INHAND DISK3) 0

———— Think or Act (15) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Rulei's action has free variable: =X2019 is bound to PEG1

Useful-Rule-Instance: State: ((NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (ON DISK3 PEG1) (NOTEX ((INHAND DISK3))) (PEG1) (DISK3)) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK3 DISK1))), Rules NIL can be used. To achieve (NOTEX ((ON DISK1 PEG1))), Rules (0) can be used.

New plan for the differences: (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))) 0

———— Think or Act (16) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1)))

Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG1))) (ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Action: (MPICK DISK1 PEG1) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Index: 0 Bindings: ((=X2018 . DISK1) (=X2019 . PEG1))

Decide to =====¿ (MPICK DISK1 PEG1)

Pick DISK1 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))) accomplished!

———— Think or Act (17) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Useful-Rule-Instance: State: ((NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (ON DISK3 PEG1) (NOTEX ((INHAND DISK3))) (PEG1) (DISK3)) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK3 DISK2))), Rules NIL can be used. To achieve (NOTEX ((ON DISK2 PEG1))), Rules (0) can be used.

New plan for the differences: (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1))) 0

———— Think or Act (18) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1)))

Useful-Rule-Instance: State: ((NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG1))) (ON DISK2 PEG1) (NOTEX ((INHAND DISK2))) (PEG1) (DISK2)) Action: (MPICK DISK2 PEG1) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Index: 0 Bindings: ((=X2018 . DISK2) (=X2019 . PEG1))

Decide to =====¿ (MPICK DISK2 PEG1)

Pick DISK2 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Wrong prediction: (((INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) 0 ((=X2018 . DISK2) (=X2019 . PEG1)))

Explain the surprise by comparing with the last succeed application viewed by the current relations

(((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1)) (NIL NIL) (MPICK DISK1 PEG1) ((=X2018 . DISK1) (=X2019 . PEG1)))

Explanation 1: Relation differences inside bindings.

Rel-Then: NIL Rel-Now: ((INHAND DISK1))

WHY: (((INHAND =X2218)))

Update the wrong rule and complement its sibling.

**** BEGIN RULES ****

Index: 0 Sibling: 2 State: ((=X2018) (=X2019) (ON =X2018 =X2019) (NOTEX ((GR =X2018 =X2112) (ON =X2112 =X2019))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND =X2018)))) Action: (MPICK =X2018 =X2019) Predict: ((INHAND =X2018) (NOTEX ((ON =X2018 =X2019))) (=X2019) (=X2018))

Index: 1 Sibling: NIL State: ((INHAND =X2033) (NOTEX ((ON =X2033 =X2034))) (=X2034) (=X2033)) Action: (MPUT =X2033 =X2034) Predict: ((ON =X2033 =X2034) (NOTEX ((INHAND =X2033))) (=X2034) (=X2033))

Index: 2 Sibling: 0 State: ((NOTEX ((NOTEX ((INHAND =X2218))) (NOTEX ((GR =X2018 =X2112) (ON =X2112 =X2019))))) (ON =X2018 =X2019) (NOTEX ((INHAND =X2018))) (=X2019) (=X2018)) Action: (MPICK =X2018 =X2019) Predict: ((ON =X2018 =X2019))

**** END RULES ****

Planning for the remaining goals:

((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))

Differences towards the goal (ON DISK1 PEG1) are: (ON DISK1 PEG1)

Differences towards the goal (ON DISK3 PEG3) are: (ON DISK3 PEG3)

Find rules for the differences and sort them ...

To achieve (ON DISK1 PEG1), Rules (1) can be used. To achieve (ON DISK3 PEG3), Rules (1) can be used.

———— Think or Act (19) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK1 PEG1)

Useful-Rule-Instance: State: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2033 . DISK1) (=X2034 . PEG1))

Decide to =====¿ (MPUT DISK1 PEG1)

Put DISK1 on PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK1 PEG1) accomplished!

———— Think or Act (20) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK3 PEG3)

Useful-Rule-Instance: State: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG3))) (PEG3) (DISK3)) Action: (MPUT DISK3 PEG3) Predict: ((ON DISK3 PEG3) (NOTEX ((INHAND DISK3))) (PEG3) (DISK3)) Index: 1 Bindings: ((=X2033 . DISK3) (=X2034 . PEG3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((INHAND DISK3))

Find rules for the differences and sort them ...

To achieve (INHAND DISK3), Rules (0) can be used.

New plan for the differences: (INHAND DISK3) 0

———— Think or Act (21) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Rulei's action has free variable: =X2019 is bound to PEG1

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK3 DISK1))), Rules NIL can be used. To achieve (NOTEX ((ON DISK1 PEG1))), Rules (0) can be used.

New plan for the differences: (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))) 0

———— Think or Act (22) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1)))

Useful-Rule-Instance: State: ((DISK1) (PEG1) (ON DISK1 PEG1) (NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK1)))) Action: (MPICK DISK1 PEG1) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Index: 0 Bindings: ((=X2018 . DISK1) (=X2019 . PEG1))

Decide to =====¿ (MPICK DISK1 PEG1)

Pick DISK1 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))) accomplished!

————— Think or Act (23) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1))) (NOTEX ((INHAND DISK1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK3 DISK2))), Rules NIL can be used. To achieve (NOTEX ((ON DISK2 PEG1))), Rules (0) can be used. To achieve (NOTEX ((INHAND DISK1))), Rules (1) can be used.

New plan for the differences: (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1))) 0 (NOTEX ((INHAND DISK1))) 1

————— Think or Act (24) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1)))

Useful-Rule-Instance: State: ((DISK2) (PEG1) (ON DISK2 PEG1) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG1) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Index: 0 Bindings: ((=X2018 . DISK2) (=X2019 . PEG1))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((INHAND DISK1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((INHAND DISK1))), Rules (1) can be used.

New plan for the differences: (NOTEX ((INHAND DISK1))) 1

————— Think or Act (25) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((INHAND DISK1)))

Rulei's action has free variable: =X2034 (ON DISK3 PEG3) is bound to PEG2

Useful-Rule-Instance: State: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG2))) (PEG2) (DISK1)) Action: (MPUT DISK1 PEG2) Predict: ((ON DISK1 PEG2) (NOTEX ((INHAND DISK1))) (PEG2) (DISK1)) Index: 1 Bindings: ((=X2034 . PEG2) (=X2033 . DISK1))

Decide to =====¿ (MPUT DISK1 PEG2)

Put DISK1 on PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK1 PEG2) accomplished! Goal (NOTEX ((INHAND DISK1))) accomplished!

————— Think or Act (26) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1)))

Useful-Rule-Instance: State: ((DISK2) (PEG1) (ON DISK2 PEG1) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG1) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Index: 0 Bindings: ((=X2018 . DISK2) (=X2019 . PEG1))

Decide to =====¿ (MPICK DISK2 PEG1)

Pick DISK2 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1))) accomplished! Goal (NOTEX ((INHAND DISK1))) accomplished!

————— Think or Act (27) —————

Seeing= ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((INHAND DISK2))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((INHAND DISK2))), Rules (1) can be used.

New plan for the differences: (NOTEX ((INHAND DISK2))) 1

————— Think or Act (28) —————

Seeing= ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((INHAND DISK2)))

Rulei's action has free variable: =X2034 (ON DISK3 PEG3) is bound to PEG2

Useful-Rule-Instance: State: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG2))) (PEG2) (DISK2)) Action: (MPUT DISK2 PEG2) Predict: ((ON DISK2 PEG2) (NOTEX ((INHAND DISK2))) (PEG2) (DISK2)) Index: 1 Bindings: ((=X2034 . PEG2) (=X2033 . DISK2))

Decide to =====¿ (MPUT DISK2 PEG2)

Put DISK2 on PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Wrong prediction: (((ON DISK2 PEG2) (NOTEX ((INHAND DISK2))) (PEG2) (DISK2)) 1 ((=X2034 . PEG2) (=X2033 . DISK2)))

Explain the surprise by comparing with the last succeed application viewed by the current relations

(((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1)) (NIL NIL) (MPUT DISK1 PEG2) ((=X2034 . PEG2) (=X2033 . DISK1)))

Explanation 1: Relation differences inside bindings.

Rel-Then: ((INHAND =X2033)) Rel-Now: ((INHAND =X2033))

Creating relations on the bound things .....

Explanation 2: Middle circle relation differences.

Rel-Then: ((GR DISK2 DISK1) (GR DISK3 DISK1)) Rel-Now: ((GR DISK2 DISK1) (GR DISK3 DISK2) (ON DISK1 PEG2))

WHY: (((ON =X2381 =X2034)))

Spliting rule 1 with WHY.

**** BEGIN RULES ****

Index: 0 Sibling: 2 State: ((=X2018) (=X2019) (ON =X2018 =X2019) (NOTEX ((GR =X2018 =X2112) (ON =X2112 =X2019))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND =X2018)))) Action: (MPICK =X2018 =X2019) Predict: ((INHAND =X2018) (NOTEX ((ON =X2018 =X2019))) (=X2019) (=X2018))

Index: 1 Sibling: 3 State: ((=X2033) (=X2034) (INHAND =X2033) (NOTEX ((ON =X2381 =X2034))) (NOTEX ((ON =X2033 =X2034)))) Action: (MPUT =X2033 =X2034) Predict: ((ON =X2033 =X2034) (NOTEX ((INHAND =X2033))) (=X2034) (=X2033))

Index: 2 Sibling: 0 State: ((NOTEX ((NOTEX ((INHAND =X2218))) (NOTEX ((GR =X2018 =X2112) (ON =X2112 =X2019))))) (ON =X2018 =X2019) (NOTEX ((INHAND =X2018))) (=X2019) (=X2018)) Action: (MPICK =X2018 =X2019) Predict: ((ON =X2018 =X2019))

Index: 3 Sibling: 1 State: ((NOTEX ((NOTEX ((ON =X2381 =X2034))))) (INHAND =X2033) (NOTEX ((ON =X2033 =X2034))) (=X2034) (=X2033)) Action: (MPUT =X2033 =X2034) Predict: ((INHAND =X2033))

**** END RULES ****

Planning for the remaining goals:

((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))

Differences towards the goal (ON DISK1 PEG1) are: (ON DISK1 PEG1)

Differences towards the goal (ON DISK2 PEG1) are: (ON DISK2 PEG1)

Differences towards the goal (ON DISK3 PEG3) are: (ON DISK3 PEG3)

Find rules for the differences and sort them ...

To achieve (ON DISK1 PEG1), Rules (1) can be used. To achieve (ON DISK2 PEG1), Rules (1) can be used. To achieve (ON DISK3 PEG3), Rules (1) can be used.

Rule 1 causes GOAL-CONFLICT Generating an experiment for it.

Generating an Experiment.....

————— Think or Act (29) —————

Seeing= ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (EXPERIMENT)

Useful-Rule-Instance: State: ((NOTEX ((NOTEX ((ON DISK3 PEG1))))) (INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Action: (MPUT DISK2 PEG1) Predict: ((INHAND DISK2)) Index: 3 Bindings: ((=X2033 . DISK2) #1=(=X2034 . PEG1) #1# (=X2381 . DISK3))

Decide to =====¿ (MPUT DISK2 PEG1)

Put DISK2 on PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Wrong prediction: (((INHAND DISK2)) 3 ((=X2033 . DISK2) #1=(=X2034 . PEG1) #1# (=X2381 . DISK3)))

Explain the surprise by comparing with the last succeed application viewed by the current relations

(((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1)) (NIL NIL) (MPUT DISK2 PEG2) ((=X2381 . DISK1) (=X2034 . PEG2) (=X2033 . DISK2)))

Explanation 1: Relation differences inside bindings.

Rel-Then: ((INHAND =X2033)) Rel-Now: ((INHAND =X2033))

Creating relations on the bound things .....

Explanation 2: Middle circle relation differences.

Rel-Then: ((GR DISK2 DISK1) (GR DISK3 DISK2) (ON DISK1 PEG2)) Rel-Now: ((GR DISK2 DISK1) (GR DISK3 DISK2) (ON DISK3 PEG1))

WHY: ((NOTEX ((GR =X2033 =X2381) (ON =X2381 =X2034))))

Update the wrong rule and complement its sibling.

**** BEGIN RULES ****

Index: 0 Sibling: 2 State: ((=X2018) (=X2019) (ON =X2018 =X2019) (NOTEX ((GR =X2018 =X2112) (ON =X2112 =X2019))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND =X2018)))) Action: (MPICK =X2018 =X2019) Predict: ((INHAND =X2018) (NOTEX ((ON =X2018 =X2019))) (=X2019) (=X2018))

Index: 1 Sibling: 3 State: ((NOTEX ((GR =X2033 =X2381) (ON =X2381 =X2034))) (INHAND =X2033) (NOTEX ((ON =X2033 =X2034))) (=X2034) (=X2033)) Action: (MPUT =X2033 =X2034) Predict: ((ON =X2033 =X2034) (NOTEX ((INHAND =X2033))) (=X2034) (=X2033))

Index: 2 Sibling: 0 State: ((NOTEX ((NOTEX ((INHAND =X2218))) (NOTEX ((GR =X2018 =X2112) (ON =X2112 =X2019))))) (ON =X2018 =X2019) (NOTEX ((INHAND =X2018))) (=X2019) (=X2018)) Action: (MPICK =X2018 =X2019) Predict: ((ON =X2018 =X2019))

Index: 3 Sibling: 1 State: ((ON =X2381 =X2034) (GR =X2033 =X2381) (=X2033) (=X2034) (INHAND =X2033) (NOTEX ((ON =X2033 =X2034)))) Action: (MPUT =X2033 =X2034) Predict: ((INHAND =X2033))

**** END RULES ****

Experiment done.

Planning for the remaining goals:

((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))

Differences towards the goal (ON DISK1 PEG1) are: (ON DISK1 PEG1)

Differences towards the goal (ON DISK3 PEG3) are: (ON DISK3 PEG3)

Find rules for the differences and sort them ...

To achieve (ON DISK1 PEG1), Rules (1) can be used. To achieve (ON DISK3 PEG3), Rules (1) can be used.

————— Think or Act (30) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK1 PEG1)

Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2381) (ON =X2381 PEG1))) (INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2033 . DISK1) (=X2034 . PEG1))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((INHAND DISK1))
Find rules for the differences and sort them ...
To achieve (INHAND DISK1), Rules (0) can be used.
New plan for the differences: (INHAND DISK1) 0

———— Think or Act (31) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (INHAND DISK1)
Rulei's action has free variable: =X2019 is bound to PEG2
Useful-Rule-Instance: State: ((DISK1) (PEG2) (ON DISK1 PEG2) (NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG2))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK1)))) Action: (MPICK DISK1 PEG2) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG2))) (PEG2) (DISK1)) Index: 0 Bindings: ((=X2019 . PEG2) (=X2018 . DISK1))
Decide to =====¿ (MPICK DISK1 PEG2)
Pick DISK1 from PEG2. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Goal (INHAND DISK1) accomplished!

———— Think or Act (32) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (ON DISK1 PEG1)
Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2381) (ON =X2381 PEG1))) (INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2033 . DISK1) (=X2034 . PEG1))
Decide to =====¿ (MPUT DISK1 PEG1)
Put DISK1 on PEG1. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Goal (ON DISK1 PEG1) accomplished!

———— Think or Act (33) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (ON DISK3 PEG3)
Useful-Rule-Instance: State: ((NOTEX ((GR DISK3 =X2381) (ON =X2381 PEG3))) (INHAND DISK3) (NOTEX ((ON DISK3 PEG3))) (PEG3) (DISK3)) Action: (MPUT DISK3 PEG3) Predict: ((ON DISK3 PEG3) (NOTEX ((INHAND DISK3))) (PEG3) (DISK3)) Index: 1 Bindings: ((=X2033 . DISK3) (=X2034 . PEG3))
Decide to =====¿ Propose new subgoals.
New differences to achieve:
((INHAND DISK3))
Find rules for the differences and sort them ...
To achieve (INHAND DISK3), Rules (0) can be used.
New plan for the differences: (INHAND DISK3) 0

————— Think or Act (34) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Rulei's action has free variable: =X2019 is bound to PEG1

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK3 DISK1))), Rules NIL can be used. To achieve (NOTEX ((ON DISK1 PEG1))), Rules (0) can be used.

New plan for the differences: (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))) 0

————— Think or Act (35) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1)))

Useful-Rule-Instance: State: ((DISK1) (PEG1) (ON DISK1 PEG1) (NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK1)))) Action: (MPICK DISK1 PEG1) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Index: 0 Bindings: ((=X2018 . DISK1) (=X2019 . PEG1))

Decide to =====¿ (MPICK DISK1 PEG1)

Pick DISK1 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))) accomplished!

————— Think or Act (36) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1))) (NOTEX ((INHAND DISK1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK3 DISK2))), Rules NIL can be used. To achieve (NOTEX ((ON DISK2 PEG1))), Rules (0) can be used. To achieve (NOTEX ((INHAND DISK1))), Rules (1) can be used.

New plan for the differences: (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1))) 0 (NOTEX ((INHAND DISK1))) 1

————— Think or Act (37) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1)))

Useful-Rule-Instance: State: ((DISK2) (PEG1) (ON DISK2 PEG1) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG1) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Index: 0 Bindings: ((=X2018 . DISK2) (=X2019 . PEG1))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((INHAND DISK1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((INHAND DISK1))), Rules (1) can be used.

New plan for the differences: (NOTEX ((INHAND DISK1))) 1

———— Think or Act (38) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((INHAND DISK1)))

Rulei's action has free variable: =X2034 (ON DISK3 PEG3) is bound to PEG2

Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2381) (ON =X2381 PEG2))) (INHAND DISK1) (NOTEX ((ON DISK1 PEG2))) (PEG2) (DISK1)) Action: (MPUT DISK1 PEG2) Predict: ((ON DISK1 PEG2) (NOTEX ((INHAND DISK1))) (PEG2) (DISK1)) Index: 1 Bindings: ((=X2034 . PEG2) (=X2033 . DISK1))

Decide to =====¿ (MPUT DISK1 PEG2)

Put DISK1 on PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK1 PEG2) accomplished! Goal (NOTEX ((INHAND DISK1))) accomplished!

———— Think or Act (39) ————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG1) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1)))

Useful-Rule-Instance: State: ((DISK2) (PEG1) (ON DISK2 PEG1) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG1) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Index: 0 Bindings: ((=X2018 . DISK2) (=X2019 . PEG1))

Decide to =====¿ (MPICK DISK2 PEG1)

Pick DISK2 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (NOTEX ((GR DISK3 DISK2) (ON DISK2 PEG1))) accomplished! Goal (NOTEX ((INHAND DISK1))) accomplished!

———— Think or Act (40) ————

Seeing= ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((INHAND DISK2))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((INHAND DISK2))), Rules (1) can be used.

New plan for the differences: (NOTEX ((INHAND DISK2))) 1

———— Think or Act (41) ————

Seeing= ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((INHAND DISK2)))

Rulei's action has free variable: =X2034 (ON DISK3 PEG3) is bound to PEG2

Useful-Rule-Instance: State: ((NOTEX ((GR DISK2 =X2381) (ON =X2381 PEG2))) (INHAND DISK2) (NOTEX ((ON DISK2 PEG2))) (PEG2) (DISK2)) Action: (MPUT DISK2 PEG2) Predict: ((ON DISK2 PEG2) (NOTEX ((INHAND DISK2))) (PEG2) (DISK2)) Index: 1 Bindings: ((=X2034 . PEG2) (=X2033 . DISK2))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK2 DISK1))), Rules NIL can be used. To achieve (NOTEX ((ON DISK1 PEG2))), Rules (0) can be used.

New plan for the differences: (NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2))) 0

———— Think or Act (42) ————

Seeing= ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2)))

Useful-Rule-Instance: State: ((DISK1) (PEG2) (ON DISK1 PEG2) (NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG2))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK1)))) Action: (MPICK DISK1 PEG2) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG2))) (PEG2) (DISK1)) Index: 0 Bindings: ((=X2018 . DISK1) (=X2019 . PEG2))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((INHAND DISK2))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((INHAND DISK2))), Rules (1) can be used.

New plan for the differences: (NOTEX ((INHAND DISK2))) 1

———— Think or Act (43) ————

Seeing= ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((INHAND DISK2)))

Rulei's action has free variable: =X2034 (ON DISK2 PEG2) is bound to PEG3

Useful-Rule-Instance: State: ((NOTEX ((GR DISK2 =X2381) (ON =X2381 PEG3))) (INHAND DISK2) (NOTEX ((ON DISK2 PEG3))) (PEG3) (DISK2)) Action: (MPUT DISK2 PEG3) Predict: ((ON DISK2 PEG3) (NOTEX ((INHAND DISK2))) (PEG3) (DISK2)) Index: 1 Bindings: ((=X2034 . PEG3) (=X2033 . DISK2))

Decide to =====¿ (MPUT DISK2 PEG3)

Put DISK2 on PEG3. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG3) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK2 PEG3) accomplished! Goal (NOTEX ((INHAND DISK2))) accomplished!

————— Think or Act (44) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG3) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2)))

Useful-Rule-Instance: State: ((DISK1) (PEG2) (ON DISK1 PEG2) (NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG2))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK1)))) Action: (MPICK DISK1 PEG2) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG2))) (PEG2) (DISK1)) Index: 0 Bindings: ((=X2018 . DISK1) (=X2019 . PEG2))

Decide to =====¿ (MPICK DISK1 PEG2)

Pick DISK1 from PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG3) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2))) accomplished!

————— Think or Act (45) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG3) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK2 PEG2)

Useful-Rule-Instance: State: ((NOTEX ((GR DISK2 =X2381) (ON =X2381 PEG2))) (INHAND DISK2) (NOTEX ((ON DISK2 PEG2))) (PEG2) (DISK2)) Action: (MPUT DISK2 PEG2) Predict: ((ON DISK2 PEG2) (NOTEX ((INHAND DISK2))) (PEG2) (DISK2)) Index: 1 Bindings: ((=X2034 . PEG2) (=X2033 . DISK2))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((INHAND DISK2))

Find rules for the differences and sort them ...

To achieve (INHAND DISK2), Rules (0) can be used.

New plan for the differences: (INHAND DISK2) 0

————— Think or Act (46) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG3) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK2)

Rulei's action has free variable: =X2019 is bound to PEG3

Useful-Rule-Instance: State: ((DISK2) (PEG3) (ON DISK2 PEG3) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG3))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG3) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG3))) (PEG3) (DISK2)) Index: 0 Bindings: ((=X2019 . PEG3) (=X2018 . DISK2))

Decide to =====¿ Propose new subgoals.

New differences to achieve:
((NOTEX ((INHAND DISK1))))
Find rules for the differences and sort them ...
To achieve (NOTEX ((INHAND DISK1))), Rules (1) can be used.
New plan for the differences: (NOTEX ((INHAND DISK1))) 1

————— Think or Act (47) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG3) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (NOTEX ((INHAND DISK1)))
Rulei's action has free variable: =X2034 (ON DISK2 PEG2) is bound to PEG1
Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2381) (ON =X2381 PEG1))) (INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2034 . PEG1) (=X2033 . DISK1))
Decide to =====¿ (MPUT DISK1 PEG1)
Put DISK1 on PEG1. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG1) (ON DISK2 PEG3) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Goal (ON DISK1 PEG1) accomplished! Goal (NOTEX ((INHAND DISK1))) accomplished!

————— Think or Act (48) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG3) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (INHAND DISK2)
Useful-Rule-Instance: State: ((DISK2) (PEG3) (ON DISK2 PEG3) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG3))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG3) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG3))) (PEG3) (DISK2)) Index: 0 Bindings: ((=X2019 . PEG3) (=X2018 . DISK2))
Decide to =====¿ (MPICK DISK2 PEG3)
Pick DISK2 from PEG3. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Goal (INHAND DISK2) accomplished!

————— Think or Act (49) —————

Seeing= ((ON DISK3 PEG1) (INHAND DISK2) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (ON DISK2 PEG2)
Useful-Rule-Instance: State: ((NOTEX ((GR DISK2 =X2381) (ON =X2381 PEG2))) (INHAND DISK2) (NOTEX ((ON DISK2 PEG2))) (PEG2) (DISK2)) Action: (MPUT DISK2 PEG2) Predict: ((ON DISK2 PEG2) (NOTEX ((INHAND DISK2))) (PEG2) (DISK2)) Index: 1 Bindings: ((=X2034 . PEG2) (=X2033 . DISK2))
Decide to =====¿ (MPUT DISK2 PEG2)
Put DISK2 on PEG2. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG1) (ON DISK2 PEG2) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Goal (ON DISK2 PEG2) accomplished! Goal (NOTEX ((INHAND DISK2))) accomplished!

————— Think or Act (50) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG2) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK3 DISK1))), Rules NIL can be used. To achieve (NOTEX ((ON DISK1 PEG1))), Rules (0) can be used.

New plan for the differences: (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))) 0

————— Think or Act (51) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG2) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1)))

Useful-Rule-Instance: State: ((DISK1) (PEG1) (ON DISK1 PEG1) (NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK1)))) Action: (MPICK DISK1 PEG1) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Index: 0 Bindings: ((=X2018 . DISK1) (=X2019 . PEG1))

Decide to =====¿ (MPICK DISK1 PEG1)

Pick DISK1 from PEG1. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG2) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (NOTEX ((GR DISK3 DISK1) (ON DISK1 PEG1))) accomplished!

————— Think or Act (52) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG2) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((INHAND DISK1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((INHAND DISK1))), Rules (1) can be used.

New plan for the differences: (NOTEX ((INHAND DISK1))) 1

————— Think or Act (53) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG2) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((INHAND DISK1)))

Rulei's action has free variable: =X2034 (ON DISK3 PEG3) is bound to PEG2

Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2381) (ON =X2381 PEG2))) (INHAND DISK1) (NOTEX ((ON DISK1 PEG2))) (PEG2) (DISK1)) Action: (MPUT DISK1 PEG2) Predict: ((ON DISK1 PEG2) (NOTEX ((INHAND DISK1))) (PEG2) (DISK1)) Index: 1 Bindings: ((=X2034 . PEG2) (=X2033 . DISK1))

Decide to =====¿ (MPUT DISK1 PEG2)

Put DISK1 on PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG1) (ON DISK2 PEG2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK1 PEG2) accomplished! Goal (NOTEX ((INHAND DISK1))) accomplished!

————— Think or Act (54) —————

Seeing= ((ON DISK3 PEG1) (ON DISK2 PEG2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK3)

Useful-Rule-Instance: State: ((DISK3) (PEG1) (ON DISK3 PEG1) (NOTEX ((GR DISK3 =X2112) (ON =X2112 PEG1))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK3)))) Action: (MPICK DISK3 PEG1) Predict: ((INHAND DISK3) (NOTEX ((ON DISK3 PEG1))) (PEG1) (DISK3)) Index: 0 Bindings: ((=X2019 . PEG1) (=X2018 . DISK3))

Decide to =====¿ (MPICK DISK3 PEG1)

Pick DISK3 from PEG1. Waiting for responses from the environment ...

Observe: ((INHAND DISK3) (ON DISK2 PEG2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (INHAND DISK3) accomplished!

————— Think or Act (55) —————

Seeing= ((INHAND DISK3) (ON DISK2 PEG2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK3 PEG3)

Useful-Rule-Instance: State: ((NOTEX ((GR DISK3 =X2381) (ON =X2381 PEG3))) (INHAND DISK3) (NOTEX ((ON DISK3 PEG3))) (PEG3) (DISK3)) Action: (MPUT DISK3 PEG3) Predict: ((ON DISK3 PEG3) (NOTEX ((INHAND DISK3))) (PEG3) (DISK3)) Index: 1 Bindings: ((=X2033 . DISK3) (=X2034 . PEG3))

Decide to =====¿ (MPUT DISK3 PEG3)

Put DISK3 on PEG3. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG3) (ON DISK2 PEG2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK3 PEG3) accomplished!

Planning for the remaining goals:

((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))

Differences towards the goal (ON DISK1 PEG1) are: (ON DISK1 PEG1)

Differences towards the goal (ON DISK2 PEG1) are: (ON DISK2 PEG1)

Find rules for the differences and sort them ...

To achieve (ON DISK1 PEG1), Rules (1) can be used. To achieve (ON DISK2 PEG1), Rules (1) can be used.

————— Think or Act (56) —————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK2 PEG1)

Useful-Rule-Instance: State: ((NOTEX ((GR DISK2 =X2381) (ON =X2381 PEG1))) (INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Action: (MPUT DISK2 PEG1) Predict: ((ON DISK2 PEG1) (NOTEX ((INHAND DISK2))) (PEG1) (DISK2)) Index: 1 Bindings: ((=X2033 . DISK2) (=X2034 . PEG1))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((INHAND DISK2))

Find rules for the differences and sort them ...

To achieve (INHAND DISK2), Rules (0) can be used.

New plan for the differences: (INHAND DISK2) 0

————— Think or Act (57) —————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK2)

Rulei's action has free variable: =X2019 is bound to PEG2

Useful-Rule-Instance: State: ((DISK2) (PEG2) (ON DISK2 PEG2) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG2))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG2) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG2))) (PEG2) (DISK2)) Index: 0 Bindings: ((=X2019 . PEG2) (=X2018 . DISK2))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((GR DISK2 DISK1))), Rules NIL can be used. To achieve (NOTEX ((ON DISK1 PEG2))), Rules (0) can be used.

New plan for the differences: (NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2))) 0

————— Think or Act (58) —————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG2) (ON DISK1 PEG2) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2)))

Useful-Rule-Instance: State: ((DISK1) (PEG2) (ON DISK1 PEG2) (NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG2))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK1)))) Action: (MPICK DISK1 PEG2) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG2))) (PEG2) (DISK1)) Index: 0 Bindings: ((=X2018 . DISK1) (=X2019 . PEG2))

Decide to =====¿ (MPICK DISK1 PEG2)

Pick DISK1 from PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG3) (ON DISK2 PEG2) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (NOTEX ((GR DISK2 DISK1) (ON DISK1 PEG2))) accomplished!

————— Think or Act (59) —————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG2) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK2)

Useful-Rule-Instance: State: ((DISK2) (PEG2) (ON DISK2 PEG2) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG2))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG2) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG2))) (PEG2) (DISK2)) Index: 0 Bindings: ((=X2019 . PEG2) (=X2018 . DISK2))

Decide to =====¿ Propose new subgoals.

New differences to achieve:

((NOTEX ((INHAND DISK1))))

Find rules for the differences and sort them ...

To achieve (NOTEX ((INHAND DISK1))), Rules (1) can be used.

New plan for the differences: (NOTEX ((INHAND DISK1))) 1

———— Think or Act (60) ————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG2) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (NOTEX ((INHAND DISK1)))

Rulei's action has free variable: =X2034 (ON DISK2 PEG1) is bound to PEG3

Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2381) (ON =X2381 PEG3))) (INHAND DISK1) (NOTEX ((ON DISK1 PEG3))) (PEG3) (DISK1)) Action: (MPUT DISK1 PEG3) Predict: ((ON DISK1 PEG3) (NOTEX ((INHAND DISK1))) (PEG3) (DISK1)) Index: 1 Bindings: ((=X2034 . PEG3) (=X2033 . DISK1))

Decide to =====¿ (MPUT DISK1 PEG3)

Put DISK1 on PEG3. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG3) (ON DISK2 PEG2) (ON DISK1 PEG3) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (ON DISK1 PEG3) accomplished! Goal (NOTEX ((INHAND DISK1))) accomplished!

———— Think or Act (61) ————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG2) (ON DISK1 PEG3) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (INHAND DISK2)

Useful-Rule-Instance: State: ((DISK2) (PEG2) (ON DISK2 PEG2) (NOTEX ((GR DISK2 =X2112) (ON =X2112 PEG2))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK2)))) Action: (MPICK DISK2 PEG2) Predict: ((INHAND DISK2) (NOTEX ((ON DISK2 PEG2))) (PEG2) (DISK2)) Index: 0 Bindings: ((=X2019 . PEG2) (=X2018 . DISK2))

Decide to =====¿ (MPICK DISK2 PEG2)

Pick DISK2 from PEG2. Waiting for responses from the environment ...

Observe: ((ON DISK3 PEG3) (INHAND DISK2) (ON DISK1 PEG3) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

Goal (INHAND DISK2) accomplished!

———— Think or Act (62) ————

Seeing= ((ON DISK3 PEG3) (INHAND DISK2) (ON DISK1 PEG3) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))

CurrentDiff= (ON DISK2 PEG1)

Useful-Rule-Instance: State: ((NOTEX ((GR DISK2 =X2381) (ON =X2381 PEG1))) (INHAND DISK2) (NOTEX ((ON DISK2 PEG1))) (PEG1) (DISK2)) Action: (MPUT DISK2 PEG1) Predict: ((ON DISK2 PEG1) (NOTEX ((INHAND DISK2))) (PEG1) (DISK2)) Index: 1 Bindings: ((=X2033 . DISK2) (=X2034 . PEG1))

Decide to =====¿ (MPUT DISK2 PEG1)

Put DISK2 on PEG1. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG3) (ON DISK2 PEG1) (ON DISK1 PEG3) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Goal (ON DISK2 PEG1) accomplished!

————— Think or Act (63) —————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG1) (ON DISK1 PEG3) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (ON DISK1 PEG1)
Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2381) (ON =X2381 PEG1))) (INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2033 . DISK1) (=X2034 . PEG1))
Decide to =====¿ Propose new subgoals.
New differences to achieve:
((INHAND DISK1))
Find rules for the differences and sort them ...
To achieve (INHAND DISK1), Rules (0) can be used.
New plan for the differences: (INHAND DISK1) 0

————— Think or Act (64) —————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG1) (ON DISK1 PEG3) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (INHAND DISK1)
Rulei's action has free variable: =X2019 is bound to PEG3
Useful-Rule-Instance: State: ((DISK1) (PEG3) (ON DISK1 PEG3) (NOTEX ((GR DISK1 =X2112) (ON =X2112 PEG3))) (NOTEX ((INHAND =X2218))) (NOTEX ((INHAND DISK1)))) Action: (MPICK DISK1 PEG3) Predict: ((INHAND DISK1) (NOTEX ((ON DISK1 PEG3))) (PEG3) (DISK1)) Index: 0 Bindings: ((=X2019 . PEG3) (=X2018 . DISK1))
Decide to =====¿ (MPICK DISK1 PEG3)
Pick DISK1 from PEG3. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG3) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Goal (INHAND DISK1) accomplished!

————— Think or Act (65) —————

Seeing= ((ON DISK3 PEG3) (ON DISK2 PEG1) (INHAND DISK1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
CurrentDiff= (ON DISK1 PEG1)
Useful-Rule-Instance: State: ((NOTEX ((GR DISK1 =X2381) (ON =X2381 PEG1))) (INHAND DISK1) (NOTEX ((ON DISK1 PEG1))) (PEG1) (DISK1)) Action: (MPUT DISK1 PEG1) Predict: ((ON DISK1 PEG1) (NOTEX ((INHAND DISK1))) (PEG1) (DISK1)) Index: 1 Bindings: ((=X2033 . DISK1) (=X2034 . PEG1))
Decide to =====¿ (MPUT DISK1 PEG1)
Put DISK1 on PEG1. Waiting for responses from the environment ...
Observe: ((ON DISK3 PEG3) (ON DISK2 PEG1) (ON DISK1 PEG1) (GR DISK3 DISK1) (GR DISK3 DISK2) (GR DISK2 DISK1) (PEG3) (PEG2) (PEG1) (DISK3) (DISK2) (DISK1))
Goal (ON DISK1 PEG1) accomplished!
Planning for the remaining goals:

((ON DISK1 PEG1) (ON DISK2 PEG1) (ON DISK3 PEG3))
Time Report on clock *GTIME*: 348.61 sec.
All given goals accomplished!

*