THE UNIVERSITY OF CALGARY

A Prototype of Combined Induction

by

Sui-ky Ringo Ling

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

APRIL, 1987

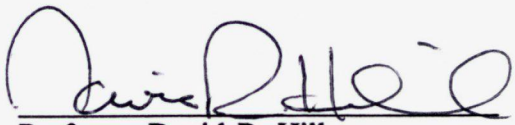
ISBN 0-315-38024-1

The University Of Calgary

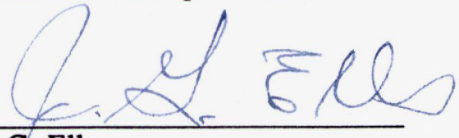
Faculty Of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, " A Prototype of Combined Induction " submitted by Sui-ky Ringo Ling in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor,
Dr. John Kendall
Department of Computer Science



Professor David R. Hill
Department of Computer Science



Dr. J. G. Ells
Department of Psychology

April 20, 1987

Abstract

One of the major bottlenecks of building knowledge-based systems is the process of acquiring domain specific knowledge. Machine learning has been suggested as one of the solutions to the problem. This thesis describes an experimental prototype which uses a combination of analytical and empirical machine learning techniques, to infer domain specific rules from solutions generated by basic rules.

The prototype is targeted to the domains where there are basic rules, but the basic rules are insufficient to infer specific rules of any degree of generality. Analytical induction is used first to exploit any available background knowledge to narrow down the search space before empirical induction. During empirical induction, the prototype minimizes the user's burden of generating instances by exploiting past rejected solutions as a source of negative instances, and using the database of the current domain to generate new instances.

Sample testing with the prototype indicated the advantage of using analytical induction to narrow down the search space before empirical induction. Further improvements are required for this prototype on its empirical induction. In addition, future research is needed in the organization of specific rules, and in establishing criteria to select appropriate techniques.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. John Kendall for his support and patience. He gave me freedom to pursue my own interest, and encouragement at time when I was in doubt.

Thanks to Dr. Ian Witten and Dr. Bruce MacDonald for their critique which made my work more concrete and precise.

Thanks to Allan Dewar for helping me with Prolog and to Brian Schack for useful discussions on my work. Thanks also to Mike Bonham for sharing his thesis typesetting environment and his experience.

Thanks to the graduate students and faculty in the Department of Computer Science at the University of Calgary. They have created such a good environment that I almost want to stay there forever.

Finally, I would like to thank my wife, Janice, for her love and support, and my daughter, Jessica, for pushing me to finish my thesis.

Table of Contents

List of Figures

CHAPTER 1

Introduction

There is a growing popularity of the expert system approach to solve problems in many areas such as medical diagnosis, well-log analysis and circuit design. One of the major problems in building an expert system is to acquire the many heuristics for the system: that is, the problem of knowledge acquisition. Machine learning has been suggested as one of the possible solutions to this problem. This thesis examines the idea of combining the analytical and empirical learning techniques for inferring specific rules for a particular type of knowledge domain.

1.1. Expert Systems

An expert system is characterized as a program developed to solve a problem for which an expert is normally required. The problem domain requires a certain amount of specialized knowledge. The expert system is usually applied to a domain where there is no well-defined algorithmic solution. Sometimes, even if an algorithmic solution exists, the solution often requires expensive computational power.

One of the most popular types of expert system is the rule-based system which consists of three major components, namely:

(1) A knowledge base which contains the explicit domain knowledge, encoded in the form of "IF condition THEN action" rules,

(2) A working memory which contains the current descriptions of the problem,

(3) An interpreter, which selects the appropriate rules from the knowledge base and modifies the descriptions in the working memory.

A rule-based system works by first matching the current descriptions of a problem in the working memory with the conditions of the rules in the knowledge base. The interpreter then selects one or more rules whose conditions match the descriptions and executes the actions of the rules to modify the descriptions. This cycle is repeated for the modified descriptions until no more rules are applicable or until the modified descriptions represent

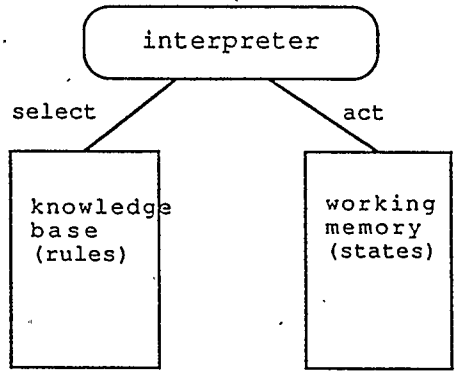


Figure 1.1: A rule-based expert system.

a solution.

One of the key characteristics of a rule-based expert system is the separation of domain specific rules in the knowledge base from the application of these rules by the interpreter. The rules represent all the domain knowledge necessary for solving the problem. It is these rules which give the computational power to an expert system. Constructing a rule-based expert system essentially means encoding these rules in the knowledge base. However, the process of encoding rules is time-consuming, tedious and expensive.

1.2. Knowledge Acquisition Problem

To encode expertise into a rule-based expert system, a knowledge engineer must first familiarize himself with the application area and gain a minimum amount of background knowledge. He then interviews an expert and tries to capture his expertise for problem solving in his field. The knowledge engineer then organizes the expertise into several blocks and represents them in the form of "if-then" rules in the knowledge base.

The major difficulty in this process is that part of the expertise is not in the form of textbook facts, but heuristics: the informal judgement rules that the expert acquires through his experience. These heuristics are seldom thought about concretely. As they are generally "rules of thumb", they also seldom appear in textbooks or journal articles. Worst of all is that the expert himself often has difficulties in verbalizing the heuristics. To capture them, an expert presents examples of how he uses his heuristics in solving specific problems. Then a knowledge engineer observes these examples, and tries to induce the heuristic rules from the examples. This thesis examines this part

of the knowledge acquisition process: inferring specific rules from examples.

The following is a case showing the importance of specific rules, and the difficulties of acquiring them. The case does not involve an expert system, but a proof assistance system. It shows that the knowledge acquisition problem does not arise only in the area of building expert systems, but also in other areas such as verification and proof systems.

One of the major parts of VLSI design is to verify the design of a chip. A HOL (Higher Order Logic) proof system has been developed for this purpose [Gordon 1985] and been used in the verification of an example chip [Joyce and Birtwistle 1985]. The HOL system is a manual-guide proof assistance system. A verification engineer enters the specification of a chip and its proposed implementation. He then tries to prove that the implementation of the circuit is equivalent to its intended specification. There are sufficient basic axioms in the HOL system to allow the construction of such proofs. In proving a circuit, the engineer has to select the right axioms at each step of his proof. The difficulty of proving a circuit correct is in the selection of the right axioms in the right sequence and this requires the experience of the engineer and his understanding of the circuit. In addition, the proof may be repeated for different circuits even though these circuits may be quite similar. For example, proving the design of a 2-bit adder by joining two 1-bit adders together is similar to proving a 16-bit adder from joining sixteen 1-bit adders.

The HOL system is capable of proving any circuit given sufficient basic axioms. However, its performance will be greatly enhanced if there are some derived axioms which handle some of the common proofs. For example, if the sequence of proving a 2-bit adder circuit can be captured and generalized,

then the derived axiom can be directly applied to prove another n-bit adder without going through the same proof. In other words, a derived axiom is a compiled proof sequence for a particular type of circuit. Its usefulness is by its direct application to those particular type of circuits, bypassing the basic proof sequence. Right now, for these derived axioms to be in the HOL system, they have to be hand-coded. The verification engineer has to recall its sequence, generalize the specification and its implementation, and put it in the system. It is an added problem that these verification engineers are scarce and they attain their expertise by proving a lot of circuits themselves.

In summary, the knowledge acquisition problem has been recognized as one of the major problems in the application of expert systems [Feigenbaum 1982]. The problem becomes even more serious in some areas where knowledge is scattered, hard to get and under constant evolution, such as VLSI circuit design [Stefik *et al.* 1981].

1.3. The scope of the thesis

Many approaches have been used to ease the knowledge acquisition problem. These include building explanation facilities, structuring the interviewing process of an expert, and so on. One solution is to build a computer program which constructs rules from examples given by an expert. This type of program falls under the area of machine learning, currently an active part of artificial intelligence research.

This thesis applies some machine learning techniques to one part of the knowledge acquisition problem: inferring rules from examples. The thesis examines the idea of combining analytical and empirical learning techniques to infer specific rules from examples or instances which are generated by the

basic rules of a domain.

The analytical learning techniques are knowledge-intensive. They make use of background knowledge to maximize generalization from a single example. The empirical learning techniques are data-intensive. They make use of syntactic comparison between examples to find the generalization. While these techniques are useful for different types of domains, there is a need to combine these techniques [Lebowitz 1986].

One possible combination is to apply analytical techniques *before* any empirical techniques. Although a domain may not have sufficient constraints to allow analytical techniques to infer a rule from a single example, these techniques can make use of the available background constraints and knowledge to constrain the generalization space. By the time empirical techniques are employed to complete the remaining generalization, the space has usually been narrowed down so that fewer examples and less computational effort may be required to reach the target generalization or rule.

There are many parts of the knowledge acquisition problem, and inferring rules from examples is only one part. This thesis does not address the other parts of the problem such as organization of rules and the selection of examples to be presented. This does not mean that the other parts are easy or insignificant; these parts are as important as that of inferring rules from examples. In fact, the results of this thesis indicate that the organization of rules may affect how a future rule may be inferred. However, the problem of inferring rules from examples is itself a difficult subject already and inclusion of other parts of the problem would make the study intractable.

1.4. Summary of results

Based on the ideas presented in this thesis, a prototype has been built on top of a C-Prolog interpreter. The prototype infers specific rules in the form of Prolog clauses. The domain has some basic rules. Some instances are generated from these basic rules and they are captured by the prototype. The prototype then infers specific rules from these instances.

In this thesis, the process of inferring a specific rule from an instance in Prolog consists of two stages:

(1) Deciding upon the constants and possible shared variables in the specific rule.

(2) Deciding upon the remaining constants and variables in the rule.

The prototype applies analytical techniques in the first stage to decide on possible constants and shared variables by tracing how an instance is derived from the basic rules. In the second stage, the prototype uses the empirical techniques to compare past and generated instances to determine the remaining constants and variables.

Sample tests with the prototype indicate the advantage of constraining the generalization using the analytical techniques followed by the empirical ones. However, further improvements are needed to make the prototype a more practical tool for future use.

1.5. Outline of the Thesis

Chapters 2 to 6 covers the remaining part of this thesis.

Chapter 2 introduces two types of learning techniques in the area of induction: learning from examples or instances. The first type is empirical

learning which is illustrated by the Arch program [Winston 1975] and the Version Space [Mitchell 1982]. The second type is analytical learning illustrated by LEAP [Mitchell, Mahadevan and Steinberg 1985]. Finally, an example is presented to show one possible combination of these two techniques [Lebowitz 1986].

Chapter 3 introduces the main idea of the thesis and the design of the prototype. The prototype has two stages in generalizing a rule from an instance. The first stage is to decide on any possible constants and shared variables in a rule by tracing how an instance is constructed from the basic rules of a domain. The second stage has two parts. The first part exploits possible past instances to specialize the rule. The second part generates new instances and uses user feedback on these instances to refine the rule.

Chapter 4 describes the implementation of the prototype and reports its current status.

Chapter 5 discusses the evaluation of the prototype, both the design and implementation. It describes the limitations of the current implementation, and two problems found in this thesis: the effect of the organization of specific rules on the learning of future rules, and deciding when to apply the empirical learning techniques.

Chapter 6 is the conclusion which summarizes the work of this thesis and suggests further research.

CHAPTER 2

Literature Review

This chapter gives the background to machine learning and discusses some previous systems of induction in the context of knowledge acquisition. The chapter begins with a general overview of machine learning, and then looks at one area of machine learning: induction. Section 2.2 describes one type of induction, empirical learning, based on the influential work of Winston [Winston 1975]. Section 2.3 explains another type, analytical learning based on the work of Mitchell [Mitchell, Mahadevan and Steinberg 1985]. Section 2.4 describes one possible combination of the two types of learning based on the work of Lebowitz [Lebowitz 1986]. Finally, section 2.5 provides a summary.

2.1. An Overview

Machine learning has been an important part of Artificial Intelligence research since its early days. The ability to learn is recognized as one of the essential characteristics of an intelligent system [Simon 1980] and constructing a "learning" computer program is advocated as one of the means of understanding this ability [Simon 1983]. However, learning is found to be very hard to capture in programs and hard to explain [Winston 1984]. Also it involves many of the problems of artificial intelligence, such as searching, perception, and knowledge representation, which are still under intense investigation [Norman 1980]. Not surprisingly, the performance of present learning systems is still primitive compared to the human being.

In spite of these difficulties, machine learning has recently attracted a considerable amount of attention due to the present success of expert systems and their potential application. Machine learning offers a possible solution to the problem of knowledge acquisition by eliminating the tedious manual process of transferring knowledge from human to program.

There are many ways of looking at machine learning. One common approach is to classify machine learning according to the learning strategy that a program uses [Carbonell, Michalski and Mitchell 1983, Michalski 1986, Dietterich 1982]. According to this approach, machine learning can be classified as follows:

Rote learning:

This area of learning is simplest in terms of the learning complexity. The program just remembers all the positive input instances so that they can be used later. This type of program is not adaptable to a complex changing environment because a stored instance can only be used later under an identical situation. The processing requirement is simple because there is no transformation on the input instances other than memorizing them. The program may have to organize the memory of these instances efficiently if the number of instances is large. Since the program just remembers exactly the input instances, it relies on its environment to provide correct, noise free instances. An example of such program is Samuel's checkers-playing program [Samuel 1959].

Induction (Learning from examples):

This is perhaps the most studied area of machine learning. The program accepts a set of classified specific instances (positive or negative or both)

of some concept, procedure or rules. Based on these input instances, the program infers features which characterize the target concept, procedure or rule. The major part of the program is the induction process where many heuristics and approaches are used. The program is more adaptable than a rote-learning program because it can apply its generalized concept or rule again in a similar but not necessarily identical situation. The program usually assumes a teacher in the environment to classify those input instances. Preferably, these instances are noise-free although some programs can handle errors in training instances. Some examples of this type of learning program are discussed later in this chapter.

Learning from experimentation and discovery:

This area of learning is the most complex of the three areas. Usually, the task involves a large search space in the inference process. Besides, the program has to classify the input instances itself, or even construct some instances, in order to test hypothesis related to a concept. Lenat's AM and Eurisko programs are classical examples in this area.

While the boundaries between these areas of machine learning are not very well-defined and precise, this classification provides one basis for examining the machine learning research. This thesis concentrates on the second area: induction.

2.1.1. Induction: learning from instances

The essential task of learning by induction is to construct the features of a concept which exist in all positive instances but not in any of the negative instances. In addition, this concept description must be broad enough to

cover not only all the positive instances that have been presented, but also some possible unobserved positive instances. This requirement is important in distinguishing learning by induction from rote learning. If the concept description only covers all the positive instances that the program has seen, then it is just another form of rote learning, memorizing all instances presented.

This requirement leads to a serious problem in induction process. Given a set of instances, positive, negative, and a combination of both, there is potentially an infinite number of concept descriptions that are consistent with the set of observed instances. Consider an example of finding a description to cover the following two instances [Utgoff 1986].

(3,4) is a positive instance

(6,5) is a negative instance

One of the possible concept descriptions is "an ordered pair of numbers where the first is numerically less than the second." However, there are also other alternative descriptions which are consistent with these two instances. They are:

(1) An ordered pair of numbers in which the first is an odd integer

(2) An ordered pair of numbers in which the second is an even integer

(3) An ordered pair of numbers in which the first is an odd integer and the second is an even integer

(4) An ordered pair of numbers in which the first is an odd integer or the second is an even integer

(5) An ordered pair of numbers the binary sum of which has a 1 in the 4's place

(6) An ordered pair of numbers the decimal sum of which has a 0 in the 10's place

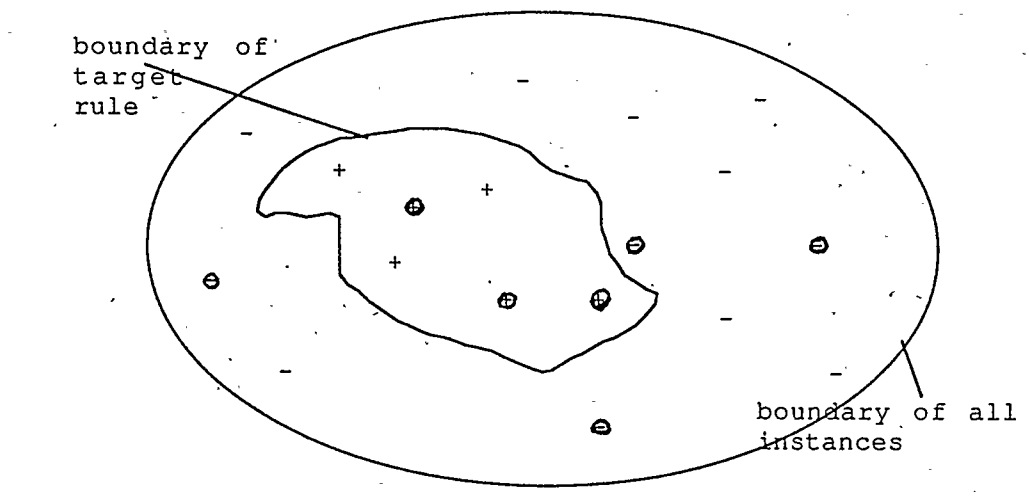
(7) A pair of numbers the sum of which is 7

(8) An ordered pair of numbers in which the second is 1 more than the first

(9) An ordered pair of numbers in which the first is not 1 more than the second

This simple example can have many possible target descriptions. For a more complex example, the space of all possible descriptions can be enormous. In fact, for a domain of N instances, there are 2 to the power N possible distinct target descriptions [Utgoff 1986].

The process of induction can be pictured as finding the target concept's boundary as shown in figure 2.1. Mitchell in his paper "generalization as search" [Mitchell 1982] put the induction process in the perspective of searching through the space of possible target concepts. A target concept is found if its boundary covers all observed positive instances and excludes any negative instances. Hopefully, the boundary can also cover the unobserved positive instances. However this induction process can be underconstrained and complex [Andreae 1985]. The search for a target concept can be combinatorially explosive if the space is large and the instances presented are few in number. In theory, a target concept can be found under this situation given sufficient time and resources. In practice, the search must be efficiently focused by constraining the search space, by presenting sufficient instances, or by a combination of both. The constraints and instances are the two major



observed positive instances ⊕
unobserved positive instances +
observed negative instances ⊖
unobserved negative instances −

Figure 2.1: The process of induction.

factors which characterize a spectrum of learning techniques within the area of learning by induction. At one end lies empirical learning which primarily relies on the presented instances to guide its search. At the other end of the spectrum lies analytical learning which primarily relies on the constraints of the background knowledge. The next two sections discuss these two types of learning and present examples of them.

2.2. Empirical Learning

In this section, the empirical learning is illustrated with two examples: the Arch program and the Version Space program.

2.2.1. The Arch Program

The Arch program was Patrick Winston's Ph.D work [Winston 1975]. The basic idea was to learn a simple concept description of an arch in a toy world. It is one of the pioneer programs of how to learn symbolic description. The program compares the positive and negative instances as shown in figure 2.2 and infers the concept of an arch as a parallelepiped object supported by two separate bricks.

The program takes the first instance, which must be a positive instance, to be the current target description. Then it compares the current target description and each instance in the input sequence in succession. If the next instance is positive, it generalizes the difference between the instance and the current concept. If the instance is negative, it specializes the difference. The concept of an instance is represented as a network of nodes. Generalization and specialization essentially involve manipulation of links between the nodes and climbing a generalization tree.

There are several noteworthy features of this program. Its learning ability relies primarily on the syntactical comparison between the structures of the instances and the current concept. Consequently, the program requires at least two distinct instances in order to learn anything. Otherwise, the concept is just the same as the first (and the only) instance.

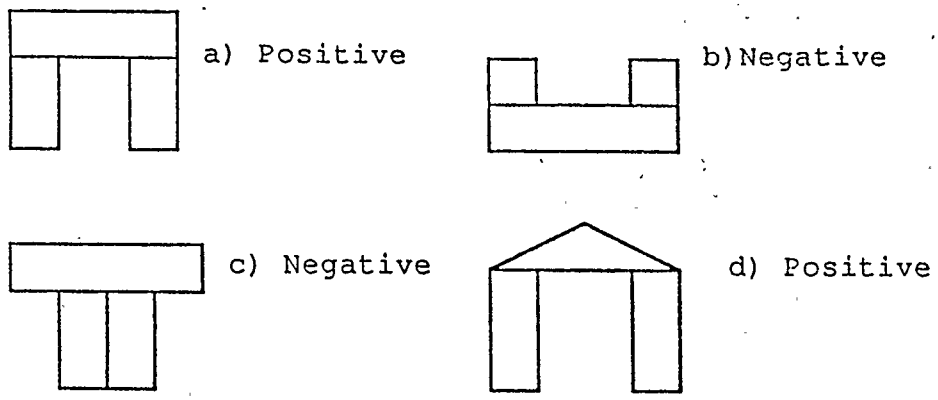


Figure 2.2: A sequence of instances for learning about arches.

The program exhibits incremental learning behavior. It takes instances one by one and modifies the concept one step at a time. Winston implied that this mode of learning was the predominant learning mode of humans [Winston 1984].

The program introduces a type of input instances called the *near-miss* negative instances. A near-miss instance is a negative instance which only differs from the current concept by *one essential feature.* This near-miss instance is used to focus the essential discriminant during the specialization process.

In case of negative instances with multiple differences, there are many ways of specialization. To handle these type of negative instances, the program adopts a depth-first search strategy and backtracks when a contradiction occurs. This requires the program to keep track of all past negative instances.

Winston later modified his Arch program and presented it as the "W procedure" [Winston 1984]. He eliminated the backtracking by arguing that inconsistency was difficult to debug. The best way to avoid debugging inconsistency and backtracking was to prevent mistakes in the first place by being conservative. He argued the important role of a co-operative teacher in presenting instances in good pedagogical order to a learner. He also suggested that a learner should be conservative in accepting the instances. His ideas on the importance of a co-operative teacher and his orderly presentation of instances leads to the study of another class of constraints, the felicity conditions, by Kurt VanLehn [VanLehn 1983, VanLehn 1987].

2.2.2. The Version Space

Mitchell [Mitchell 1982] provided a framework for looking at different data-driven learning strategies by casting them as searching through a space of possible concept descriptions. In addition, by noticing that all concept descriptions can be partially ordered according to their degree of generality, he proposed "the Version Space" as a compact representation of all possible concept descriptions which are consistent with the observed positive and negative instances.

Essentially all possible target descriptions can be stored in a partially ordered lattice. In this lattice, the most general description is at the top of

the lattice while the most specific descriptions are at the bottom of the lattice. The search space of all possible descriptions is bounded by the most general and the most specific descriptions. While the space may contain a large number of descriptions, it is sufficient to define them by the space boundaries: the most general and the most specific descriptions. An example of this lattice is shown in figure 2.3. This simple example involves descriptions of a group of people by their two features, colour of their hair and their height. The most general description is a group of people with any hair colour and any height. The most specific descriptions are different groups of people with different combinations of colour and height.

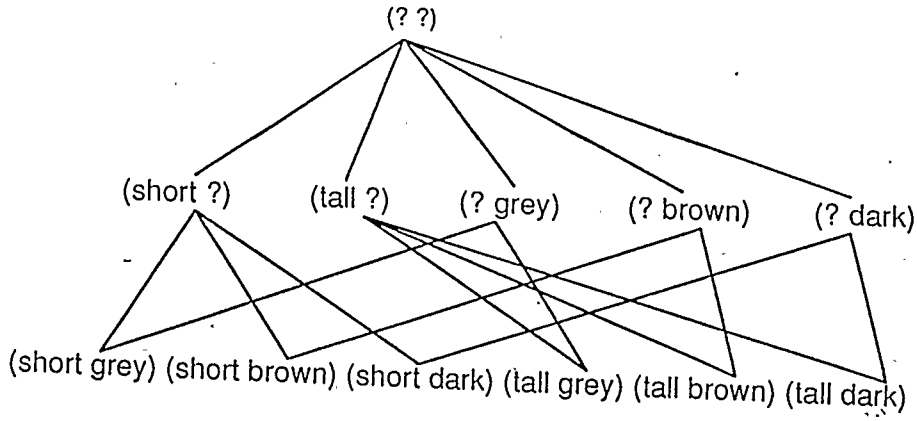


Figure 2.3: Representing a Version Space.

Mitchell's learning algorithm essentially makes use of this version space. Finding a target description in the space consists of moving the most general and the most specific boundaries until they converge to a particular description in the space. Initially, the boundaries cover the whole search space. When a positive instance is observed, the algorithm updates the current boundaries by eliminating those descriptions which are too specific to contain the instance. When a negative instance is presented, the algorithm updates the boundaries by eliminating those too general descriptions which contain the instance.

An example of finding a description of a group of people with any hair colour and short is presented in figure 2.4.

G --the set of the most general descriptions
S --the set of the most specific descriptions
X?--any Capital letter followed by '?' stands for variable
type-- type of instances, positive or negative

instance	type	S	G
(short, grey)	+	(short,grey)	(H?,C?)
(tall, brown)	-	(short,grey)	(H?,grey) (short,C?) (H?,dark)
(short, brown)	+	(short,C?)	(short,C?)

Figure 2.4: A Version Space example.

Unlike the Arch program, the version space program uses a search similar to the breadth-first search to update its boundaries. However, there are similarities between the two programs. They both use a number of positive and negative instances to converge to a target description and prune away those irrelevant ones. There is some domain knowledge built into the generalization hierarchy in the Arch program and in the version space. The knowledge helps the programs to converge to a target concept quickly without seeing all the possible instances. For example, in figure 2.4, the two positive instances (short, grey) and (short, brown) cause the concept to converge as (short, any_colour). In the generalization hierarchy of figure 2.3, any_colour is defined as a generalization of both grey, brown and dark colour. Therefore the concept of (short, any_colour) also covers the unobserved positive instance (short, dark). If the concept of any_colour is defined differently, then the concept (short, dark) covers different unobserved positive instances. It is the implicit bias of the built-in domain knowledge in the generalization hierarchy which induces a concept to cover *both observed and unobserved* positive instances. If the bias is inappropriate, it can prevent the system from ever inferring correct generalizations. If the bias is appropriate, it can provide the basis for important inductive leaps beyond information directly available from the training instances [Mitchell 1982]. The study of "bias" is an active area of machine learning research [Utgoff 1986]. While bias is useful in induction, it is usually not sufficient for a program to reach a target concept. A program still has to rely on some instances to prune away any irrelevant concept.

In summary, empirical learning depends on the relationship between instances and the implicit bias in the generalization hierarchy to reach a target concept. This type of learning technique usually requires a number of

instances. Besides the domain knowledge in the generalization hierarchy, empirical learning is relatively independent of the context of the domain. It does not consider how an instance is generated, and why the instances presented are classified as positive or negative. It is characterized as being empirical, data or instance-intensive. The next section presents another type of learning technique which relies more on the domain knowledge and less on the instances.

2.3. Analytical learning

Analytical learning requires more background knowledge from a domain to learn a concept. It has the advantage of using very few instances given sufficient knowledge. Lebowitz has given a scenario example (not yet implemented) of how the concept of "arch" would be learned in the analytical learning [Lebowitz 1986]. In his example, the program would require understanding of some prior concepts such as the concept of gravity, supports etc, and the description of structures. The program might use the concept of gravity and supports to analyze the structural description of a positive instance. It figured out that two equal height supports were necessary to support a lintel in the air but the other factors such as the colour and shape of the lintel were not important. It did not need further instances to show that two equal height supports were important.

The remaining part of this section presents several systems using this type of analytical technique in the context of the acquisition of heuristics.

2.3.1. LEAP

LEAP is a learning apprentice system for VLSI design developed at Rutgers University [Mitchell, Mahadevan and Steinberg 1985, Mahadevan 1985]. Its purpose is to acquire specific heuristic rules for verifying logic circuit designs. Mitchell characterized a type of learning apprentice system through the example of LEAP. It is an interactive knowledge acquisition system which accumulates heuristic rules by observing and analyzing the solutions of an expert through his *normal* use of the system. There is no explicit "training mode" for the system. The implication is that a co-operative teacher is not required, and that the system is more suitable than other programs such as the "Arch" as a knowledge acquisition tool for an expert system.

LEAP works together with another expert system VEXED [Mitchell, Steinberg and Shulman 1985] which is a problem solving component for the VLSI design. Given a design problem, VEXED tries to come up with a solution using its existing heuristic rules. If VEXED fails to come up with any rule or the implementation rule is not satisfactory to the user, an expert can override the decision of VEXED and supply his[1] own solution. At this point, LEAP begins to capture the expert's solution and generalize the solution into a new heuristic rule. The new rule will be used by VEXED when a future similar problem arises.

For example, the system is given the problem of implementing a function specified in figure 2.5. One of the possible implementations is to use three NOR circuits joined together as shown in figure 2.5. This involves verifying

[1] For simplicity of expressions, his, him and he are used to mean "his or her", "him or her" and "he or she".

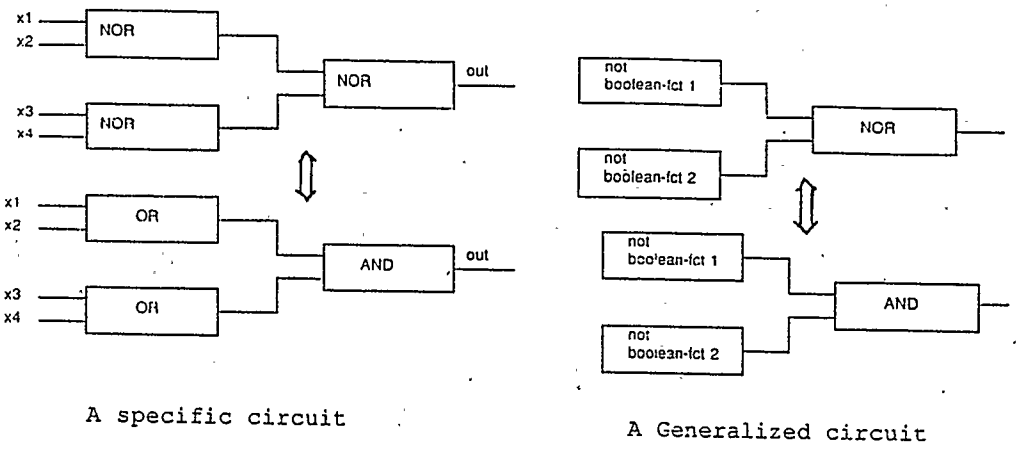


Figure 2.5: A circuit and it's generalized design for LEAP.

that the behavior of the implementation is equivalent to the required specification. The verification of this design involves using the basic De-Morgan's Law and Remove-Double-Negation operators already defined in the system. The verification sequence is shown in figure 2.6. LEAP captures this sequence and uses a technique called constraint back-propagation [Mitchell 1983, Utgoff 1986] to generalize the steps in the sequence.

Constraint back-propagation is the main generalization technique used in LEAP. It was developed in the previous system LEX2 [Mitchell 1983] to deduce the domain of an operator sequence or macro-operator that produces

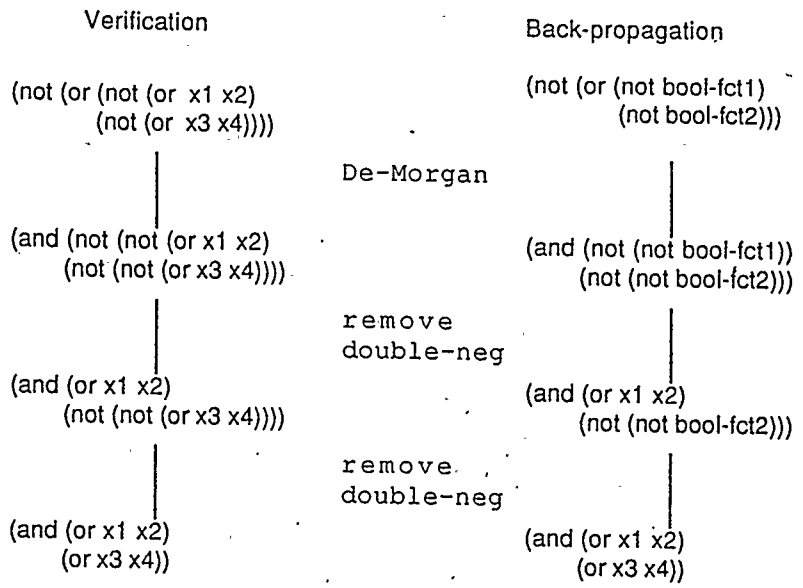


Figure 2.6: Verifying and Generalizing a circuit for LEAP.

some constrained range of states. Unlike the empirical generalization which examines the relations between instances, the constraint back-propagation examines how a positive instance is constructed from the basic operators within a particular domain.

A solution sequence can be interpreted as a transformation from an initial problem state to a final solution state through a number of intermediate states. Each basic operator is a mapping from one state (the domain) to another (the range) with constraints to restrict the operator's

domain and range. Consider a simple case of applying a single operator to an initial state to produce a final state. If a subset A of the range of the operator represents a class of solved states, then this subset can be propagated backward through the operator to find out the subset B of the domain which produces the subset A. The subset B represents a constrained domain of the operator such that application of the operator results in a group of solved state. If there is a sequence of operators, then the same operation can be applied recursively starting from the final state and working it backward until it reaches the initial state. A simple example of this process is shown in figure 2.7. The detail algorithm is shown in the work of Utgoff [Utgoff 1986].

There are several features about the LEAP which also characterizes a typical analytical learning system. One important feature about the system is that it can make use of only *one* positive instance to deduce a heuristic rule. The heuristic rule is not restricted to solving one particular example, but is generalized to solve a specific group of similar examples. In the example shown in figure 2.6, the specific example involves 4 input signals, but the generalized rule applies to any condition that matches a boolean function.

The ability to do such powerful generalization from a single instance stems from examining how a positive instance is constructed. In the case of the LEAP example, it means the verification process of showing how a particular design of 3 NOR gates to meet the functional requirement of a circuit with two OR gates and a AND gate. This points to a second requirement. Analytical learning must have sufficient domain knowledge to explain the construction of an instance. In LEAP, it means the existence of a

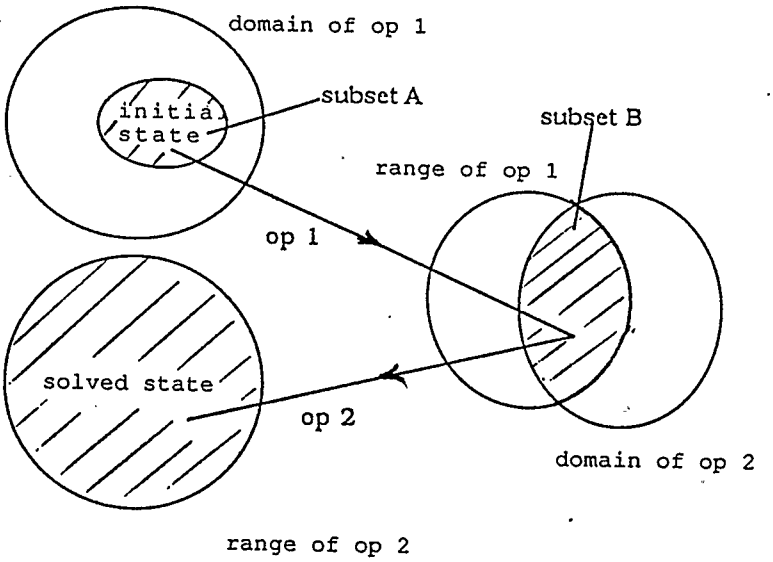


Figure 2.7: The constraint back-propagation method.

number of basic operators such as De-Morgan's law and other information required to narrow down a group of solved states. Because of the requirement for background information, analytical learning cannot be applied to an arbitrary problem domain. It usually requires a problem domain with a strong enough theory to explain and validate the training instances, such as digital circuit design and mathematical integration problems [Mitchell, Mahadevan and Steinberg 1985, Mahadevan 1985]. In fact, even in the integration domain, Utgoff reported some difficulties in learning certain concepts because the formalism could not express certain context-sensitive

relationships [Utgoff 1986].

Because analytical learning usually requires verification or explanation of a positive instance before any generalization, it is more robust than empirical learning in handling possible errors in the input instances. If an instance contains an error, the explanation process will fail and prevent the generalization process from proceeding.

The constraint back-propagation technique is one of the techniques in analytical learning. Other techniques are also being developed, such as a schemata to understand a situation [Lebowitz 1986] and a proof tree to generalize a circuit design structure [Ellman 1985].

In summary, analytical learning is able to generalize a heuristic rule or a concept from a single instance by using a great deal of background knowledge. Instead of examining the differences between instances, the generalization comes from examining how an instance is constructed. This type of learning is characterized as analytical, knowledge-intensive. The next section discusses one possible combination of the two types of learning.

2.4. Combining two types of learning

The previous two sections examined two types of learning. One of the key conditions under which a type of learning can be applied is the existence of sufficient domain knowledge. Analytical learning is suitable for a domain where there is a substantial amount of background knowledge. On the other hand, empirical learning is needed when the background knowledge is lacking. However, there are a number of domains which lie between these two extreme conditions. Some domains have a certain amount of background knowledge but the knowledge is not sufficient to allow the use of the analytical learning

only. On the other hand, using only empirical learning seems to neglect the existence of background knowledge. This section describes an example of combining these types of learning through the UNIMEM program [Lebowitz 1986], which used the empirical analysis to guide the analytical generalization.

UNIMEM is a program that takes the description of a situation and tries to build an explanation scheme to account for the situation. One domain for this program is to explain the US congressional voting records. The input information is the voting records of U.S. congressmen and the characteristics of the states and districts that they represent. The task of the program is to build an explanation of how a congressman's voting record relates to his other votes (a congressman who opposes cutting the MX missile also opposes general cuts in defense spending) or to the features of his district (a congressman from a low-income district supports the increase in social spending). There are a number of simple rules in the domain. Each simple rule relates a set of conditions (causes) to an observed behavior (results). These simple rules are rules of thumb and they are general approximations. They represent a tentative model of the domain. The explanation scheme is built by relating those relevant simple rules into a structure to explain the voting behavior of a congressman.

Building such a structure using only analytical learning is computationally expensive as there are many basic rules, and also a number of possible features (over 30) to consider for each explanation. Among those features, some of them may be the causes while other may be results due to other features. Identification of the causes among those features is not trivial.

For example, it might be that districts with high farm property values are thought to have oil reserves and hence their congressmen would vote to limit any profit tax on oil reserves found on property. Conversely, it might be that voting to limit the profit tax on reserve actually causes the farm value to be high, as potential investors would know oil profits would not be subject to high taxes.

Lebowitz suggested using the idea of predictability to identify those features which are causes. Predictive features are those which exist uniquely in a given situation and they are most likely to be the causes. This argument follows from the observation that non-predictive features occur in many situations, and are associated with many different combinations of other features. Hence, they do not predict a single outcome. For example, if a situation is made up of two features, A and B, and A only occurs in one situation, and B in many, B cannot cause A. If B did cause A, A would appear in all the other situations that B was in.

After identifying those potential predictive features, the system starts to match those features with the conditions of the basic rules and tries to build up a structure to explain the remaining features. Lebowitz has reported the use of this predictability to prevent using irrelevant features in building an explanation. He also found that some of those features which were supposed to be predictive in the simple rules could in fact be explained by other predictive features. Hence the result could be used to debug the initial set of simple rules. The essential result from this work was that predictive ability provided significant control over the process of building up an explanation. The program did not have to use brute force and try every possible

explanation rule sequence. Consequently, the efficiency of analytical learning was increased in an area where it could be combinatorially explosive.

2.5. Summary

This chapter has presented two types of machine learning. The empirical learning does not require too much domain knowledge and relies on a number of instances, positive and negative, to reach a target concept. It is suitable for the situation where domain knowledge is lacking. The analytical learning requires sufficient domain knowledge to reach a target concept and relies less on any input instances. These two types of learning characterize two ends of a spectrum in term of the requirement of domain knowledge. An example has been presented where both techniques are combined to handle a situation between the two ends of the spectrum. The next chapter presents the prototype in this thesis. The prototype employs the idea of using the analytical learning *before* the empirical learning.

CHAPTER 3

Design of the prototype

This chapter discusses in detail the design of a prototype for another approach to learning. The purpose of the prototype is to combine both analytical and empirical techniques in inducing a domain specific rule from an instance which is generated by some basic rules. Section 3.1 describes two types of rules for certain domains: basic and specific. Section 3.2 discusses the Prolog programming language as the representation of both instances and rules. Section 3.3 describes how an user interacts with the prototype, and sections 3.4 and 3.5 describe the two stages of induction: analytical and empirical. The final section 3.6 summarizes this chapter.

3.1. Basic and specific rules

Two types of knowledge are currently recognised as providing a basis for solving problems in a domain [Chandrasekaran and Mittal 1983, Rosenbloom *et al.* 1985]. Although the definition is not precise, they are generally referred to as deep and surface knowledge. In building an expert system, these types of knowledge are represented by two types of rules: basic and specific, respectively.

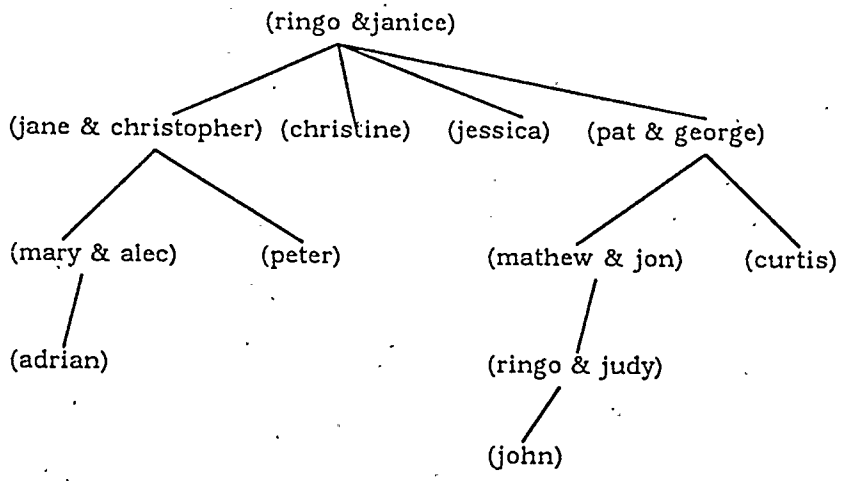
The basic rules of a domain represent the essential knowledge and they have a wide scope of applicability within the domain. The specific rules represent the knowledge derived from the basic rules. Each of these specific rules is restricted to a particular situation. These specific rules are usually more efficient than the basic rules since they relate the aspects of a task

directly to action consequences, bypassing the computational steps needed to apply the basic rules of the domain.

For example, in solving the mathematical integration problems in LEX [Mitchell, Utgoff and Banerji 1983], there are basic operators with conditions specifying where they can be legally applied. However, there are also heuristics, which specify the conditions where it is beneficial or useful to apply those operators. Each heuristic specifies a restricted subset of the legally applicable situations for an operator where application of it is most likely to lead to a solution.

One characteristic of many present-day expert systems is that they have a lot of domain specific rules which allow them to arrive at problem solutions quickly [Rosenbloom *et al.* 1985]. Therefore, there are a number of research efforts arrived at developing systems which acquire these specific rules or heuristics to enrich the computational power of an expert system, and to ease the bottleneck in the knowledge acquisition process [Langley 1985, Rosenbloom *et al.* 1985, Mitchell, Utgoff and Banerji 1983].

In what follows, a family-tree domain is chosen to illustrate the two types of rules and set the context for the purpose of the prototype which is: to infer specific rules from the instances which are generated by the basic rules. Although this thesis only uses a family-tree and a context-free grammar problem for demonstration, the approach embodied in the prototype is not restricted to solving these two problems, but is targeted at a broader class of problems. These problems are chosen because they are familiar examples in most Prolog textbooks [Sterling and Shapiro 1986, Clocksin and Mellish 1981, Bratko 1986].



Note: two persons in a bracket are a married couple.

```
parent(janice, christopher)
parent(janice, christine)
        :
parent(mathew, ringo)
parent(ringo, john)
```

Figure 3.1: A family tree and it's Prolog clauses.

Given a family-tree as shown in the figure 3.1, and a set of predicates which define the relationship of the nodes in the family-tree, the two clauses

cl(3.1)

```
related(X,Y) :- (parent(X,Y);parent(Y,X)).
```

cl(3.2)

```
related(X,Y) :- (parent(X,Z);parent(Z,X)), related(Z,Y).
```

are able to find if any two persons in the family-tree are related. The two clauses (3.1) and (3.2) are the basic rules for the family-tree problem. These basic rules completely define any solution to the problem if that solution exists. The basic rules are also flexible so that they can be applied to any different occurrence of the problem. They can be used to examine two nodes whether these two nodes are related through a single node or many intermediate nodes. While these two clauses are flexible, they may also be very inefficient when compared to specific rules such as,

cl(3.3)

```
related(X,Y) :- parent(X,Z),parent(Z,Y). /* grandparent*/
```

cl(3.4)

```
related(X,Y) :- parent(Z,X),parent(Z,Y)./* sibling */
```

For example, in searching whether "christopher"[1] and "christine" are related in figure 3.1, the rules (3.1) and (3.2) needs five instantiations to find a solution, while the more specific rule (3.4) needs only two instantiations. If these domain specific rules are applied to a right situation, they can be very efficient in the sense that they bypass a lot of unnecessary search.

Although these domain specific rules are efficient, each of them is restricted to a particular situation. Rule (3.4) is useful only to find out if "christopher" and "christine" are related through a sibling relationship. It fails to find out that "ringo" and "mary" are related because "ringo" and "mary" are related in a grandparent relationship. While the specific rules are useful for computational efficiency, the basic rules are needed in cases where

[1] Names should start with an upper-case letter but this is in conflict with the Prolog definition of constants as lower-case letters. Therefore, names are quoted and in lower-case to designate this as a constant in Prolog.

all the specific rules fail.

The prototype in this thesis endeavors to induce those specific rules such as (3.3) and (3.4) from basic rules such as (3.1) and (3.2). The prototype is initially given only the basic rules to solve any problem in a particular domain. As a result, the problem solving efficiency is low. As more problems are solved, useful specific rules are induced by the prototype. The system then relies more on the specific rules than the basic rules to handle future problems. Consequently, the prototype's problem solving efficiency increases.

One of the key characteristics used in the machine learning area is the classification of the target rules or concepts according to their degree of generality. A rule "A" is more general than a rule "B" if B can be obtained by substitution of certain variables in "A" with specific values. For example, rule (3.4) is more general than rule (3.5)

cl(3.5)

related(X,Y) :- parent(ringo,X),parent(ringo,Y). /*children of ringo*/

because (3.5) is an instance of (3.4) by instantiating the variable "Z" with the constant "ringo".

In this thesis, the prototype is required to learn specific rules with any degree of generality desired by a user. This requirement creates a problem. For a given instance, there are many possible target rules with different degrees of generality. The induction of target rules such as (3.3) and (3.4) can be achieved by using the analytical technique only to the extent that they have the same degree of generality as the basic rules. However, the analytical technique alone cannot infer a target rule such as (3.5) because that rule has a degree of generality more specific than that of the basic rules. Empirical

technique is required to complement the analytical ones.

3.2. Prolog as the implementation language

The Prolog programming language is used to implement the prototype and also as a representation language for both instances and rules. A Prolog program is a set of Horn clauses, which have the general form

A :- B1, B2, B3...Bn

where A, and the B's are atomic formulae [Shapiro 1982]. Each formula is a predicate consisting of a predicate symbol, called a functor, and optionally followed by a list of terms in parentheses, separated by commas. Each term can be a variable, denoted by a capital letter, or a constant, denoted by a lower-case letter, or a functor.

A Prolog clause can have both declarative and procedural interpretation [Kowalski 1979]. Declaratively, the above clause can be read as "A is the conjunction of the B's". Procedurally, it can also be interpreted as "to fulfill the goal of A, satisfy the goals of B1,B2..Bn".

Because of its dual interpretation, the Horn clause has been used as a common basis for representation in both concept-learning and rule-learning programs [Bundy, Silver and Plummer 1985]. For example, the clause

cl(3.3)

related(X,Y) :- parent(X,Z),parent(Z,Y).

represents the concept of grandparent. Procedurally, it can also be interpreted as a rule or a program to search whether the two nodes, X and Y, in the family-tree are related. To fulfill the declarative meaning of whether two people are related in a grandparent relationship, the program searches for

a common node which relates to the two nodes X and Y in the tree.

Most of the Prolog systems, such as the C-Prolog system, are implemented sequentially. A common strategy is to evaluate the goals from left to right. As a result, there is a difference in the declarative and the procedural interpretation when deciding whether two logic clauses are the same. Declaratively, the clause

cl(3.6)

related(X,Y) :- parent(Z,Y),parent(X,Z).

has the same meaning as the clause (3.3). Procedurally, these two clauses can be different. The difference is due to the sequential evaluation of clauses in the language. An example to illustrate the difference are the clauses for doing arithmetic addition.

cl(3.7)

sum(X,Y,Z,S) :- I is X+Y, S is I+Z.

cl(3.8)

sum(X,Y,Z,S) :- S is I+Z, I is X+Y.

Both clauses have the same declarative meaning. That is, the result of adding three numbers together can be obtained by adding two numbers to get an intermediate value, and then by adding the intermediate value to the remaining number. Given the query of sum(1,2,3,S), clause (3.7) succeeds with the S value returned as 6. Cl(3.8) fails because the variable I is undefined when the first goal of " S is I+Z " is evaluated.

In this thesis, unless it is explicitly stated, logic clauses are interpreted as procedural rules. An extra criterion is imposed if two rules are said to be the same. Two rules are the same if they have the same atomic formulae

arranged in the same order. According to this criterion, clauses (3.3) and (3.6) are considered as different rules even though they represent the same declarative meaning.

This criterion also allows the prototype to narrow down its search for target rules considerably. Consider the following example involving two separate instances:

(1) related(ringo,alec) :- parent(ringo,christopher), parent(christopher,alec).

(2) related(ringo,mathew) :- parent(ringo,pat), parent(pat,mathew).

To infer a rule which covers these two instances, the prototype only has to evaluate the same goals in the same sequence according to the above criterion. Consequently, the prototype only needs to consider the two possible pairings: "parent(ringo,christopher)" with "parent(ringo,pat)"; and "parent(christopher,alec)" with "parent(pat,mathew)". However, without the criterion, the prototype also has to consider two extra possible pairings of "parent(ringo,christopher)" with "parent(pat,mathew)" and "parent(christopher,alec)" with "parent(ringo,pat)". Given two instances with each one having N goals, the number of possible combinations would be N! (N factorial) without the constraint of the criterion. With this constraint, only one combination needs to be considered.

Finally, an instance[2] is defined as a single item of input to a learning program. An instance of a rule is obtained by instantiation of all variables of the rule with specific constants. For example, "related(christopher,christine) :- parent(ringo,christopher), parent(ringo,christine)" is a positive instance of the

[2] The word "instance" is to replace the commonly used word "example" in order to avoid confusion over the various usages of the word "example"

rule "related(X,Y):- parent(ringo,X), parent(ringo,Y)", but the "related(christopher,christine):-parent(janice,christopher), parent(janice,christine)" is a negative instance of the rule.

3.3. User interaction

Environment plays an important role in providing the input information that a learning system needs, and the user is a major part of that environment. This prototype is an "interactive rule acquisition system". One of its requirements is to minimize the burden put on a user to generate positive and negative instances for the system.

A major part of the input information comprises the positive and negative instances. But there is extra information which can reduce the complexity and difficulty of the learning tasks. One of the sources of extra information is the way the instances are presented to a learning program. There is a range of possibilities for how the extra information is encoded through the presentation of instances. For example, the ARCH program [Winston 1975], assumes a co-operative teacher. The teacher provides instances free of any noise, classifies the instances as positive or negative, and presents them in good pedagogical order. Winston also suggested the use of near-miss negative instances to help a learning program to identify those essential features easily and narrow down to the target concept quickly. SIERRA [VanLehn 1987] is another program which makes use of extra information encoded in the sequence of instances to ease its induction of disjunction and invisible objects. In SIERRA, the instances are partitioned into lessons, and the lessons are sequenced. Each lesson can only introduce one disjunctive feature. Also, lessons are organized so that detailed work is

shown first, followed by optimized work. VanLehn suggested these two major restrictions of the sequence of instances, which then facilitates the induction which otherwise would be extremely difficult. He argued the importance of using the constraints from the presentation of instances and termed these as felicity conditions.

Although the strategies adopted make the learning tasks much easier in the above programs, the users are expected to do extra work in organizing and providing instances to the programs. If too much effort is required by the user in presenting instances to an induction program, there is concern that the practical uses of the program to ease the knowledge acquisition problem may be limited. The benefits of automatic induction of rules may be offset by the requirement for too much effort by the user.

Mitchell has suggested the use of another type of learning apprentice system such as LEAP [Mitchell, Mahadevan and Steinberg 1985]. In contrast to ARCH, LEAP does not require an explicit "teaching mode", it acquires specific rules through the normal use of the system by the user.

The present prototype is closer to LEAP in its requirement for instances. It requires a positive instance as the initial input. The positive instance is a solution to a particular problem, and is derived using the basic rules during the problem solving stage. The system induces a rule from the instance with a degree of generality equivalent to that of the basic rule. If the user is not satisfied with the degree of generality of the induced rule, he can invoke the next step to refine the rule. The refinement relies on the empirical technique. Instead of asking the user to provide instances, the system extracts negative instances from the past solution trace, if there are any. Then, it generates

instances from the existing database and requests the user to classify them. Based on the classification by the user, the system returns a refined rule.

The details of the process are as follows:

Problem Solving Stage

The user first invokes the system to find a solution to a problem. The system may produce several solutions. The user picks the first acceptable solution and uses it as a positive instance. The system collects any solutions preceding the acceptable one in a solution trace. The user then invokes the next stage with the positive instance.

Analytical Induction

The system uses the analytical technique to generalize the positive instance into a rule. The degree of generality of the induced rule is equivalent to that of the basic rules in the system.

Empirical Induction

If the user wants the induced rule to be more specific, he then invokes this stage. The system tries to find useful negative instances from the solution trace. A useful negative instance is one which has a similar structure to the rule being processed, but with one discriminating feature. After that, the system generates instances based on the existing database model and asks the user for a classification of each instance. The user answers "yes" to a positive instance, and "no" to a negative one. After a sequence of instances, the system returns a more specialized rule.

Two major problems were found in building this prototype. The first one was in choosing problems for which the induced rules were useful. There are

numerous problems and solutions for a given domain. Some of the problems are interesting and typical, while others rarely occur. We do not want the system to infer rules for every problem. We want the system to induce only those specific rules which solve the typical and commonly occurring problems. The prototype does not know which problem is common or typical and has to rely on the user to decide. A learning system, which learns by itself, such as AM [Davis and Lenat 1982] has to confront this problem. Another major problem was to decide the degree of generality of the induced rules required by the user. Again, the prototype has to rely on the user's choice.

The prototype takes advantage of its interactive nature to determine the user's choice on the above problem. When the user invokes the second stage of analytical induction, the user communicates to the prototype that the instance is a solution to a common problem. The prototype also assumes that the user wants a more specialized rule when he invokes the stage of empirical induction. In general, an interactive learning system can often provide more chances for the system to infer extra information from a user than a system which learns by itself.

3.4. Analytical Induction

The idea behind the prototype developed for this thesis is to exploit any available background knowledge as much as possible. When a domain can provide sufficient background knowledge, it is possible for the analytical generalization to induce a rule from a single instance [Mitchell 1983]. However, when there are insufficient constraints, this prototype first uses the available constraints to guide the analytical generalization, and then uses empirical techniques to deal with the area where constraint is lacking.

Consider a positive instance such as

cl(3.8)

related(christopher,christine) :- parent(ringo,christopher), parent(ringo,christine).

Possible target rules for this instance are:

cl(3.9)

related(X,Y):- parent(Z1,X), parent(Z2,Y).

cl(3.4)

related(X,Y):- parent(Z,X), parent(Z,Y).

cl(3.5)

related(X,Y):- parent(ringo,X), parent(ringo,Y).

In this prototype, the process of induction from a positive instance consists of two stages:

(a) Deciding the relationship between the variables and constants. Does "ringo" in the instance (3.8) bind to two separate variables as in the target rule (3.9), or bind to the same variable as in the rule (3.4)?

(b) Deciding whether the constants in the instance can be turned into variables. Is "ringo" in the instance (3.8) a constant in the rule (3.5), or just an instantiation of the variable Z in the rule (3.4)?

The following two sub-sections consider the induction in two cases.

3.4.1. Case I

Consider an instance which relates the top node "ringo" to the bottom node "john" in the right-hand side of the family-tree in the figure 3.1. There are two separate nodes called "ringo" which exist in the path from the top

node "ringo" to the bottom node "john". One is the top node "ringo". Another one is the one immediately above the bottom node "john". As a result, the instance becomes

cl(3.10)

related(ringo,john) :- parent(ringo,john), parent(mathew,ringo), parent(george,mathew), parent(ringo,george).

The proper rule for this instance should be:

cl(3.11)

related(R,J) :- parent(R1,J),parent(M,R1),parent(G,M),parent(R,G).

If the prototype is given only the instance cl(3.10) without any knowledge of how the instance is derived, the prototype cannot directly induce the rule (3.11) from the instance (3.10).

If the prototype assumes that each constant with the same value comes from a unique variable in the target rule, then the rule

cl(3.12)

related(R,J) :- parent(R,J),parent(M,R),parent(G,M),parent(R,G).

is induced which is not correct as it neglects the existence of a separate variable R1. On the other hand, if every constant, regardless of whether it shares the same value with any other, is assumed to come from a separate variable, then a rule of the form

cl(3.13)

related(R,J) :- parent(R1,J1),parent(M,R2),parent(G,M),parent(R3,G).

is induced which is too general. The rule (3.13) ignores the shared variables in the clause. It ignores that R1 and R2 are the same, and so are R and R3.

If the prototype does not have any background knowledge, it has to rely on empirical techniques. It can use the cl(3.13) as the upper bound (the most general form) and the instance cl(3.10) as the lower bound (the most specific form). By having a lot of positive and negative instances, the prototype would eventually arrive at the proper rule cl(3.11). But this means that the user has to produce a lot of instances to guide the prototype.

If the prototype has the history of how the instance cl(3.10) was derived from the basic rules during the problem solving stage, then it is able to infer the relationship between each constant in the instance with each variable in the target rule without relying on the empirical technique. In addition, if each basic rule is applied properly during the problem solving stage, then it can infer the proper rule cl(3.11) from just a single instance(3.10).

3.4.2. Case II

Analytical generalization can allow the induction of a rule from a single instance if the prototype not only knows how the instance is derived from the basic rules but also knows that each basic rule is applied correctly during the problem solving stage. The second assumption may not hold true all the time. In this prototype, the system is given some basic rules in order to solve a wide variety of problems in a given domain. The condition of each basic rule is quite open so that each basic rule can fit a wide variety of situations. As a result, the basic rules are quite general. If the prototype relies on these rules as the basis of generalization, and the application of these rules during the problem solving stage is not constrained, then it is possible to induce a rule which is over-general. Furthermore, there is a requirement that the user may want to induce a rule of arbitrary generality.

For example, consider the instance

cl(3.14)

related(christopher,christine) :- parent(ringo,christopher), parent(ringo,christine)

The analytical generalization would induce this instance cl(3.14) into a rule such as:

cl(3.15)

related(X,Y) :- parent(Z,X),parent(Z,Y) /* sibling */

However, another possible generalization of cl(3.14) is the rule

cl(3.16)

related(X,Y) :- parent(ringo,X),parent(ringo,Y) /* children of ringo */

The instance of cl(3.14) does not provide other information for the prototype to decide which one, cl(3.15) or cl(3.16), is the target rule. The conditions of the basic rules cl(3.1) and cl(3.2) are insufficient to allow the analytical generalization to decide that "ringo" in the instance cl(3.14) should be a constant instead of a variable. During the problem solving stage, the instantiation for the predicate parent(ringo,christopher) is "parent(X,Y)" instead of "parent(ringo,Y)", the generalization process just infers "ringo" to be a variable. Therefore rule cl(3.15) is induced instead of rule cl(3.16).

To enable analytical generalization to induce rule cl(3.16) instead of cl(3.15), two extra conditions must be provided by the system. Instead of only two general basic rules cl(3.5) and cl(3.6), the prototype needs an extra set of rules such as parent(ringo,Y). During the problem solving stage, the system must be able to use "parent(ringo,Y)" instead of the predicate "parent(X,Y)" to generate the instance (3.14).

Building an extra set of rules is a major problem. The difficulty lies in deciding which set of rules should be included. By including an extra set of rules, the system would assume the types of specific rules that a user would eventually like to induce in the future. This assumption is too much to make. Also, the extra set of rules actually belong to specific type rules. Including the extra set would contradict the original purpose of inducing the specific rules from the basic one.

Even if this extra set of rules exists in the system, there is no guarantee that this set of rules should be used instead of the general basic one during the problem solving stage. The user just issues the question "relate(christopher,christine)" to the system and there is no further information provided with the question. If the database arranges the extra set of rules in front of the basic ones, the system may pick "parent(ringo,Y)" in this example. While it may be suitable for this case, it may be undesirable in another case where the rule (3.15) is indeed the target one. The strategy of placing this set of rules in front of the basic one shifts the learning bias of the prototype. The bias would be to prefer rules like cl(3.16) over the other ones like cl(3.15).

It appears that using only the analytical generalization for this case is not enough. On the other hand, relying only on the empirical technique runs into the problem of needing a lot of positive and negative instances as discussed in the section 3.4.

While the basic rules are not sufficiently constrained to enable analytical generalization from a single instance, they can be used to infer any constants and shared variables in the target rule. In this example, the basic rules

cannot be used to decide whether "ringo" in the instance comes from the variable "Z" in the target rule, but it can decide that there is a common shared variable "Z" or a common shared constant "ringo" between the two subgoals. Because of that, the analytical generalization eliminates one possible target rule "parent(Z1,X), parent(Z2, Y)".

The target rule can be imagined as lying within the version space bounded by a pair of most specific and most general rules.[3] In this example, without the stage of the analytical generalization, the prototype has to begin its empirical generalization with the most general boundary of "parent(Z1,X), parent(Z2,Y)", and the most specific boundary of "parent(ringo,christopher), parent(ringo,christine)". If the stage of analytical generalization exists, the prototype can start its stage of empirical generalization with the most general boundary "parent(Z,X), parent(Z,Y)" instead of "parent(Z1,X), parent(Z2,X)". The most general boundary has been lowered. By bringing down the most general boundary to a lower level, the analytical generalization narrows down the version space for the next stage of empirical generalization. The whole scenario is shown in the figure 3.2

If the rule cl(3.16) is the target rule for this example, extra instances are necessary in the empirical generalization. Of course, one useful source of instances is the user. However, there are two other possible sources of instances. The next section discusses the empirical induction and how those instances can be extracted and generated.

---

[3] The following explanation is described using the Version Space approach for convenience. However, the current implementation of this prototype has not yet incorporated the Version space method for its empirical techniques.

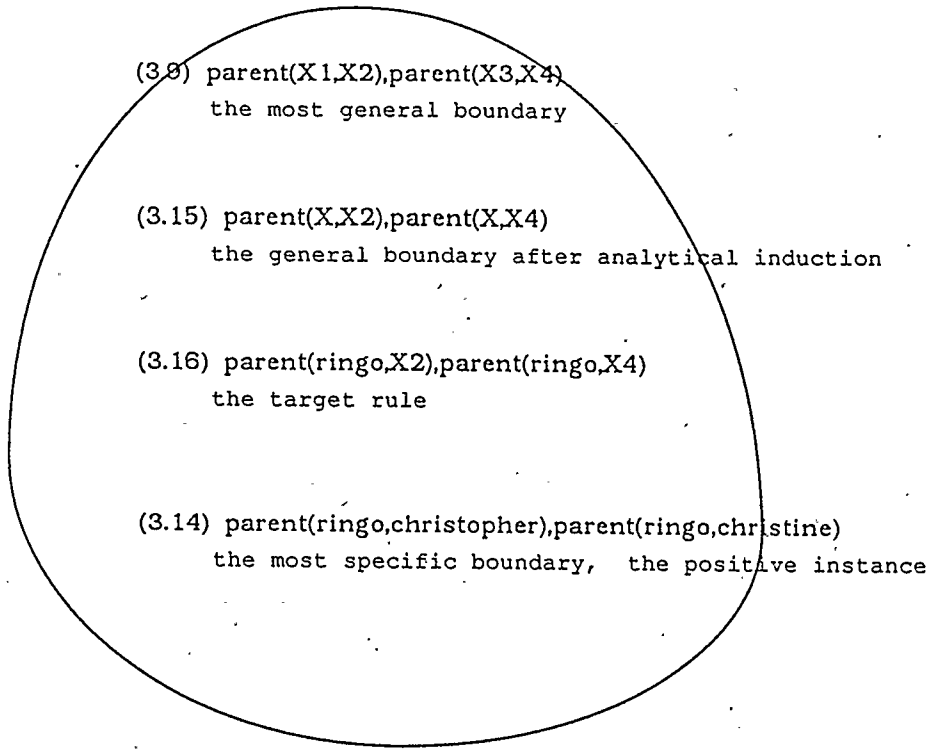


Figure 3.2: The Version Space of a family tree example.

3.5. Empirical Induction

Empirical induction has two steps: extracting negative instances and generating instances. They are described in the following two sub-sections.

3.5.1. Extracting Instances

The general condition of the basic rules cl(3.1) and cl(3.2) not only create problems during the analytical generalization, they also create problems

during the problem solving stage. Because their conditions are general, these basic rules may need several trials before they can get to the right solution. Given the database as defined in the figure (3.1), the first solution to the question ":-related(christopher,christine)" is the instance

cl(3.17)

related(christopher,christine) :- parent(janice,christopher), parent(janice,christine).

If the user is only interested in finding whether they are related through "ringo" as opposed to anyone else, then he is going to reject this instance as an incorrect solution. The system has to search for another solution until it finds the one "related(christopher,christine) :- parent(ringo,christopher), parent(ringo,christine)". If the instance cl(3.17)--related through janice-- is captured and stored, then it can become a useful negative instance for empirical induction. The instance (3.17) gives the justification that the variable X in the cl(3.15) should be specialized into the constant "ringo". Otherwise, if X was indeed the variable, then the instance cl(3.17) should also be acceptable as a solution instead of being rejected. There are three problems related to using the solution trace as the source of negative instances for empirical induction.

An assumption is made that the user is only interested in finding the correct solution, and no alternative solution. As soon as he has found the correct solution, he stops the system from generating further solutions. This assumption is necessary for the system to decide that the last solution in the solution trace is a positive instance and any solution prior to it is a negative one. If alternative solutions are allowed in the solution trace, the system

cannot decide which one is the alternative correct solution, or which one is a negative instance. Under these circumstances, further input from the user would be required to distinguish negative solutions from alternative correct solutions.

The second problem is that there is no guarantee that negative instances will exist in a solution trace. In the data-base of the family-tree in the figure 3.1, the predicates "parent(janice,christopher)" and "parent(janice,christine)" are put in front of the predicates "parent(ringo,christopher)" and "parent(ringo,christine)". As a result, the negative instance of "parent(janice,christine), parent(janice,christopher)" is generated before the correct solution of "parent(ringo,christopher), parent(ringo,christine)". However, if those predicates involving "ringo" are put at the beginning of the data-base, then the first solution is the correct one. The system will not have a chance of generate another instance involving "janice".

The final problem with the solution trace is that it is an unstructured source of negative instances. In the current implementation, only negative *near-miss* instances which lie within the version space are useful. Consider the same example involving "christopher" and "christine". Beside the instance cl(3.14) "related(christopher,christine):- parent(ringo,christopher), parent(ringo,christine)" which is accepted as the positive one, two negative instances are generated before the instance (3.14). They are:

cl(3.18)

related(christopher,christine) :- parent(janice,christopher), parent(janice,christine).

cl(3.19)

related(christopher,christine) :- parent(ringo,christine), parent(janice,christine), parent(janice,christopher).

Of the two negative instances, only cl(3.18) lies in the current version space bounded by the rule (3.15) "related(X,Y) :- parent(Z,X), parent(Z,Y)", and the instance (3.14) "related(christopher,christine) :- parent(ringo,christopher), parent(ringo,christine)". Therefore, the negative instance (3.18) is the only one useful in narrowing down the most general boundary of (3.15) into the one "related(X,Y) :- parent(ringo,X),parent(ringo,Y)". (3.19) cannot be a useful negative instance because its structure is different from the positive instance (3.14).

Cl(3.18) is a useful negative instance not only because it lies within the version space, but also it is a *near miss* instance. It contains only one discriminant, "janice", from the positive instance.

Consider another case involving the "great grandparent" relationship. The most general boundary and most specific boundary are defined by

cl(3.20)

related(X,Y) :-parent(X,X1),parent(X1,X2),parent(X2,X3),parent(X3,Y).

cl(3.21)

related(ringo,adrian) :- parent(ringo,jane), parent(jane,mary), parent(mary,adrian).

There are two possible negative instances within the version space defined by these two boundaries. They are:

cl(3.22)

related(ringo,adrian) :- parent(ringo,jane), parent(jane,alec),

parent(alec,adrian).

cl(3.23)

related(ringo,adrian) :- parent(ringo,christopher), parent(christopher,alec), parent(alec,adrian).

Only the instance (3.22) can be used as it contains only one discriminant,"alec". The negative instance (3.23) cannot be used because there are two discriminants, "alec" and "christopher" in (3.23). The prototype cannot tell which discriminant causes the instance (3.23) to be a negative one. It may be "alec", or "christopher", or both of them. If the instance (3.22) was positive and (3.23) was negative, using both of them could point out that "christopher" was the discriminant which caused (3.23) to be negative. But if (3.22) and (3.23) are both negative instances, then the prototype cannot find out all essential discriminants. The prototype can only definitely identify "alec" as the essential negative discriminant since the cl(3.22) is a near-miss negative instance. However, the prototype cannot decide for sure that "christopher" in cl(3.23) is also a negative discriminant. It may be "alec" which also causes cl(3.23) to be a negative instance. The current implementation of the prototype only uses a single rule instead of a set of rules to represent the most general boundary of the version space. It can only handle *near miss* negative instances. The possible improvement to this limitation will be discussed in the chapter (6).

3.5.2. Generating instances from current database

The solution trace cannot guarantee to have useful negative instances. Even if it has, there may be insufficient negative instances to specialize the upper boundary of the target rule. As a result, further instances are still

required. In the case of the sibling example, the upper boundary has been narrowed down to "related(X,Y) :- parent(ringo,X),parent(ringo,Y)." as a result of the negative instance (3.18). There is a possibility that the variables X and Y are also constant. The system has to find and confirm this. The last part of the prototype is to generate instances, asking the user to classify the instances as being either positive or negative. Based on the user's classification, the system tries to determine which variable in the upper boundary can be turned into a constant. The following paragraphs describe why the generation of instances from the current database *must be guided.*

One of the simple ways of generating instances is to use the upper boundary as the rule to generate an instance. By causing the rule to backtrack continuously, the prototype can eventually generate all the possible instances in the current domain. While this approach is simple to implement, it generates a lot of redundant and useless instances. For example, if a rule has been specialized to the form "related(X,Y):- parent(ringo,X), parent(ringo,Y)" from a previous instance of "related(jessica,christopher) :- parent(ringo,jessica), parent(ringo, christopher)", another instance of "related(jessica,christopher) :- parent(janice,jessica), parent(janice, christopher)" is redundant because it does not contribute to any further generalization or specialization. An instance with one discriminant is useful because its classification as positive or negative can uniquely identify whether the discriminant can be turned into a variable. An instance of two discriminants may not be very useful unless one of them has been identified previously. To avoid generating redundant and useless instances, the process of generating instances must be guided.

The prototype uses two criteria to generate instances. The criteria are similar to the ones used in the previous step of extracting negative instances from the solution trace. Whether they are positive or negative, only instances within the current version space are useful. Therefore, the instance "related(jessica,pat):- parent(janice,jessica), parent(janice,pat)" is not useful as it lies outside the current most general boundary of "related(X,Y) :- parent(ringo,X),parent(ringo,Y)". In order to ensure that only instances within the current version space are generated, the most general boundary "related(X,Y) :- parent(ringo,X), parent(ringo,Y)" is used as the rule to generate instances.

As the user only gives a simple answer "yes" or "no" to the instances, the system has to generate those instances with only one discriminant. Therefore, the instance of "related(jessica,christine) :- parent(ringo,jessica), parent(ringo,christine)" is a useful instance, whether it is a positive or negative, as there is only one discriminant, "jessica". The instance of "related(jessica,pat) :-parent(ringo,jessica), parent(ringo,pat)" is a useful positive instance, but not a useful a negative one. If the user answer "yes" to this one, then the system can justify that both the constants "jessica" and "pat" are variables "X" and "Y". But if the answer is "no", then the system runs into the same problem of deciding whether one or both of them are the negative discriminants. Consequently, this current prototype is restricted to generating instances with only one discriminant.

Whether the version space can be successfully narrowed down to the target rule depends on the number of instances which can be generated from the database. If there are five undecided variables before this step, then at

least five instances, each one with only one discriminant, have to be produced. The database in the figure (3.1) can generate sufficient instances for the prototype to learn the sibling and grandparent rules, but not the "great grandparent" rule. For the "great grandparent" rule, only four different instances can be generated in total from the database of figure (3.1). They are:

cl(3.24)

related(ringo,adrian) :-parent(ringo,christopher),parent(christopher,mary), parent(mary,adrian).

cl(3.25)

related(janice,adrian):-

parent(janice,christopher),parent(christopher,mary), parent(mary,adrian).

cl(3.26)

related(ringo,ringo):- parent(ringo,george), parent(george,mathew), parent(mathew,ringo).

cl(3.27)

related(janice,ringo):- parent(janice,george), parent(george,mathew), parent(mathew,ringo).

In this example, the first instance (3.24) is taken to be the initial positive instance. The rest of the instances are generated by this step. The second instance (3.25) can be used to decide whether the constant "ringo" comes from a variable in the target rule. The instance (3.26) has more than one discriminant. If the user classifies (3.26) as being positive, then the system can justify turning all the remaining constants in the instance into variables. The rule then becomes:

cl(3.28)

related(X,Y):-parent(X,L),parent(L,M),parent(M,Y).

But if the answer is "no", then the prototype cannot infer any further information from the instance (3.26). The last instance cl(3.27) is not useful as it does not contain any extra information.

3.6. Summary

This chapter has described two types of rules for certain domains: basic and specific rules. The solution for a problem is initially found using the basic rules of a domain. Then the prototype infers a specific rule from the solution to solve a similar class of problems.

In the first stage of analytical induction, the prototype infers any common shared variables or constants in the target rule. In the second stage of empirical induction, the prototype first extracts any useful negative near-miss instances to specialize the upper bound of the target rule. Then it further generates instances from the existing database of the domain and requests the user's classification on these instances. Based on the user's classification, the prototype refines both the upper and lower bound of the target rule. The next chapter describes the implementation of the prototype.

CHAPTER 4

Implementation of the prototype

This chapter discusses the current implementation of the prototype. The target rule can be imagined as contained in a hypothesis space of many possible target rules which are bounded by its most general form and the most specific form. A common terminology for this space is called the version space. Section 4.1 describes the first stage, analytical induction which generalizes a positive instance into the most general form of its version space. The second stage, empirical induction, has two sub-stages. Section 4.2 describes the first substage of specializing the most general form of the version space by using any near-miss negative instances extracted from the solution trace. Section 4.3 describes the second sub-stage of generating instances from the database, and using these instances to narrow down the version space bounded by the most general and most specific forms. Section 4.4 describes the current status of the prototype, and section 4.5 summarizes this chapter.

4.1. Analytical Induction

In order to carry out analytical induction, the prototype has to know the form of the positive instance. The instance is actually the solution to a particular problem generated by an inference engine. The prototype also has to know how the solution is generated by the inference engine.

At present, the inference engine of the C-Prolog system just returns an answer "yes" or "no" to the query such as "related(christopher,christine)". In the case when "christopher" and "christine" are related, the C-Prolog system

just indicates an answer "yes", but gives no indication of how these two are related. Also there is no information as to what basic rules have been used in finding the solution.

Another requirement is that the prototype must be able to access any of the solutions which are rejected by the user. The storage of these solutions forms the source of negative instances for the next stage of empirical induction. The existing C-Prolog system, however, does not store these instances once a query is finished.

All the above requirements of the prototype seems to indicate the need for extracting extra information from the current C-Prolog interpreter. One way to capture this extra information is to build some functions within the existing C-Prolog interpreter. However, these functions may interfere with the performance of the interpreter and affect other users of the system. To prevent interference with other users, the work for this thesis involved building a separate interpreter to simulate the actions of the current C-Prolog interpreter.

The current implementation involves a meta-interpreter running on top of the existing C-Prolog interpreter. Besides simulating the action of the existing interpreter, the meta-interpreter captures the form of the solution such as "parent(ringo,christopher),parent(ringo,christine)" as well as any solution rejected by the user. The rejected solutions are asserted into the database with the special tag "frecord" so that they can be retrieved even after an query is finished.

There are several analytical generalization techniques for different applications such as story understanding, heuristics and so on. In LEAP and

LEX where heuristics are learned from instances produced by the basic operators, constraint back-propagation technique is used. This involves two separate stages: capturing the sequence of the basic operators during the problem solving stage; and generalizing using the same sequence in the induction stage. To use this technique, the prototype has to supply extra information on how those basic operators can be back-propagated. This extra information may not exist or be defined for all basic operators. Other limitations have been cited for this technique [Utgoff 1986]. The prototype uses a different technique from the constraint back-propagation technique. Instead of keeping a record of how the basic rules are used, the meta-interpreter produces a generalized and/or tree to represent their sequence of application.

In Prolog, the process of finding a solution can be captured in an and/or tree. The top node of the tree represents the query or the question, and the bottom nodes of the tree represent the solution to the query. The internal nodes of the tree represent the intermediate steps taken by the interpreter. In essence, the and/or tree of the solution represents the basic rules and the binding of the variables in the rules with constant values. To infer how the constants are shared in the tree, it is necessary to have a corresponding generalized and/or tree which captures the basic rules but without any binding of variables. A corresponding mapping between these two trees can reveal whether two constants in the solution tree are indeed shared together. An example is shown in figure 4.1.

The meta-interpreter builds these two trees at the same time. When the query "related(christopher,christine)" is fed into the meta-interpreter, the

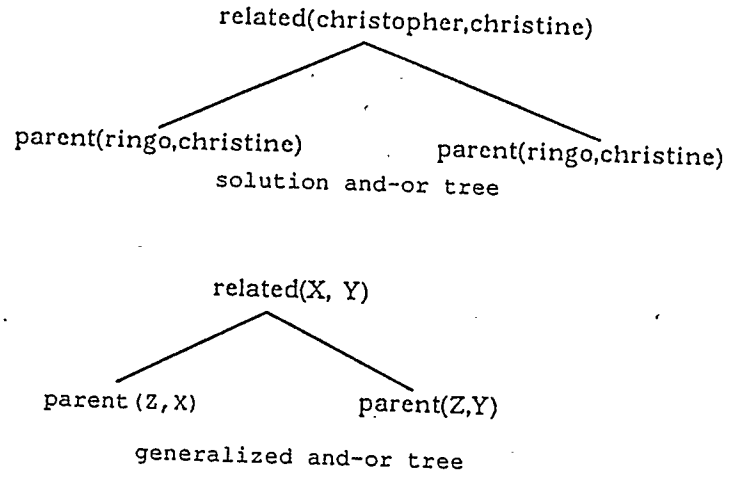


Figure 4.1: A solution tree and its generalized form.

generalized form of the query "related(X,Y)" is also fed to the same interpreter. When the solution "parent(ringo,christine), parent(ringo,christopher)" is found, the generalized rule "parent(Z,X), parent(Z,Y)" is also generated from the generalized and/or tree.

It is essential that the two trees correspond to each other. Two major problems can arise from trying to produce a corresponding generalized tree. Part of the first version of the meta-interpreter is shown in figure 4.2. Two simple examples are shown in figure 4.3 and 4.4 to illustrate the problems. A modified version is shown in figure 4.5.

```
Call1(Goal, Result, Head, Rule) :-
        clause(Goal, Body1),
        clause(Head, Body2),
        call1(Body1, Result, Body2, Rule).

where Goal--the query
        Result--the bottom nodes of the solution tree
        Body1--subgoals of the query
        Head--generalized query
        Rule--the bottom nodes of the generalized tree
        Body2--subgoals of the generalized query
```

Figure 4.2: A portion of the meta-interpreter (Version I).

The first problem is that the generalized query may instantiate with some clauses which the solution query will avoid. For example, in figure 4.3, the meta-interpreter has to instantiate both the actual query of "test(2,Y)", and the generalized query of "test(L,M)". The query "test(2,Y)" can only bind with the (2) clause of "test", but not (1). "test(L,M)" can bind with either (1) or (2). Because of the sequential evaluation, "test(L,M)" in this case will bind with (1) first. Therefore a discrepancy occurs between these two trees. The solution tree binds with the second clause (2) of test, while the generalized tree binds with the first clause (1) of test. To prevent this mismatch from happening, the instantiations in both trees must be tested for equality at each step of instantiation. In this example, the generalized tree has to give up its first instantiation and try the second one which matches with the instantiation of the solution tree.

```
subgoal(1).
subgoal(2).
subgoal(yes).
subgoal(no).

1) test(1,Y) :- subgoal(yes), subgoal(Y).
2) test(2,Y) :- subgoal(no), subgoal(Y).

QUERY: call1(test(2,Y), R, test(L,M), Rule)
ANSWER:
Y = 1
R = subgoal(no),subgoal(1)
L = 1
M = _7
Rule = subgoal(yes),subgoal(_7)
```

Figure 4.3: The problem of different instantiations.

The second problem comes from different backtracking by the two different trees. In figure 4.4, the interpreter has tried 1), 2) and 3) of the clause "rel(X,Y)" in both trees and fails. The interpreter then first backtracks the solution tree and tries the final clause 4) where it succeeds. However, the generalized tree still remains at the previous clause 3). Consequently the instantiations in both trees are different. To prevent this problem, the interpreter must backtrack both trees to the same place at the same time.

The improved version (II) is shown in figure 4.5 which takes care of the two problems. However the code of the improved version (II) is less easy to understand than the first version.

```
parent(ringo,john).
parent(ringo,mary).

1) rel(X, Y) :- parent(X,Y).
2) rel(X, Y) :- parent(Y,X).
3) rel(X, Y) :- parent(X, Z), rel(Z, Y).
4) rel(X, Y) :- parent(Z, X), rel(Z, Y).

QUERY: call1(rel(john,mary), R, rel(X,Y), Rule).
ANSWER:
R = parent(ringo,john),parent(ringo,mary)
X = _5
Y = _6
Rule = parent(_5,_24),parent(_24,_6)
```

Figure 4.4: The problem of different backtracking.

```
Call1(Goal, Result, Head, Rule) :-
        clause(Head, Body1),
        copy((Head:-Body1),(Head2:-Body2)),
        Goal = Head2,
        call1(Body2, Result, Body1, Rule).
```

Figure 4.5: A portion of the meta-interpreter (Version II).

The first stage can be summarized by the following procedure:

--input the query and its generalized form to the meta-interpreter
--While the acceptable solution is not found loop
    --search for another solution and its corresponding generalized rule simultaneously
    --output the solution to the user for its feedback
    --if the answer is "acceptable" then

```
        --output the solution and its generalized rule
        --stop the loop
--else
        --assert the solution into a database with a tag "frecord"
        --continue the loop
```

4.2. Extracting negative instances

The first sub-stage of the second stage "empirical induction" is to specialize the most general form of the rule found in the previous stage. The specialization is done by using any near-miss negative instances extracted from the solution trace. There are three steps in this stage. The first step is to extract negative instances which are marked with the tag "frecord" from the database. The second step is to rearrange the negative instances and makes use of those near-miss instances for specialization. The final step is to specialize the general form of the rule so that the version space is smaller.

After the first stage of analytical induction, the version space which contains the target rule can be described and bounded by the most specific and most general form of the rule. The most specific form is the positive instance used in the analytical induction. The most general form is the rule induced in the analytical induction. If instances are rejected by the user while the interpreter is finding the solution, and some of the instances are within the version space, then these instances can be used further to narrow down the boundary of the version space. In other words, the version space should be narrowed further to exclude those instances by specializing the most general form of the space. The instances marked with the tag "frecord" in the database are negative instances. Some of them are within the version space, but some of them are not. The prototype has to extract those within the version space in order to specialize the most general form. To do that,

the prototype extracts those instances which can match the most general form, i.e. the rule induced during the analytical induction. Those negative instances which do not even match the most general form are outside the version space and they are not useful for any further induction.

The second step is to organize those negative instances lying inside the version space. The current prototype can only make use of the near-miss negative instances. Therefore, it is necessary to select further the near-miss instances from the ones found in the previous step. The negative instances are compared with the most specific form one by one to find out how different they are from the most specific form. Then the instances are sorted in a list according to the order of difference. Those near-miss instances with just one discriminant are put at the front of the list, followed by those with two discriminants, and so on.

The final step is to specialize the most general form. Both the most general and most specific forms are compared with each negative instance, starting with the first one in the list. Comparison of the most specific form with a given negative instance indicates what is the essential discriminant. The essential discriminant is used to locate the corresponding value in the most general form. If the corresponding value in the most general form is a variable, then the variable is specialized to the corresponding constant in the most specific form. If the corresponding value in the most general form is a constant already, then the value is kept the same. The specialization stops when all near-miss instances with one discriminant in the list are exhausted. The procedure is shown in the figure 4.4.

Most general form-----unique list [Z,X,Y]
related(X, Y) :- parent(Z, X), parent(Z, Y).

Most specific form-----unique list [ringo,christopher,christine]
related(christopher,christine):-parent(ringo,christopher),parent(ringo,christine)

Negative instance-----unique list [janice,christopher,christine]
related(christopher,christine):-parent(janice,christopher),parent(janice,christine)

The new most general form-----unique list [ringo,X,Y]
related(X,Y) :- parent(ringo, X), parent(ringo, Y).

Most specific form-----unique list [ringo,christopher,christine]
related(christopher,christine):-parent(ringo,christopher),parent(ringo,christine)

Figure 4.6: Specialization using negative instance.

Instead of storing and comparing each instance as a whole entity, the current prototype abstracts a unique list of variables and constants for each instance. For example, for the instance of

related(christopher,christine):-

parent(ringo,christopher),parent(ringo,christine)

the unique list to represent that is

[ringo,christopher,christine]

This is due to the fact that the prototype just manipulates constants and variables in the brackets. The functors of the predicates such as "parent" are not manipulated. Therefore, in the third step of specialization, each instance or a rule can be uniquely represented by a list of its constants and variables without essential loss of information. The corresponding unique list for each instance is also shown in the figure 4.6. Consequently, only unique lists are manipulated instead of the whole instance or the whole rule. Manipulation of the unique lists is both more time efficient and more space efficient than manipulating the whole rule or instance. At the end of the third step, the unique list is used to produce the rule back in its original form.

The second step can be summarized by the following procedure:

```
--extract negative instances using the most general form
--sort the negative instances in a list according to the number
of discriminants
--while near-misses still exist in the list do:
        --compare each negative instance with the most specific form
        to locate the discriminant
        --find the value in the most general form corresponding to this
        discriminant
        --if the value is a variable, turn it into the corresponding
                constant value in the most specific form
                ----else keep the value as it is.
```

4.3. Generating Instances

The next sub-stage of the "empirical induction" is to generate instances from the current database. Based on the classification of the generated instances from the user, the prototype generalizes the most specific form or specializes the most general form. The generation of instances must be guided so that redundant instances are avoided. The prototype first attempts to generate instances with only one discriminant. After all these instances from the database are exhausted, the prototype tries to generate instances with

more than one discriminant.

The current prototype relies on a list of variable/constant pairs to guide its generation of instances. This list is constructed by extracting unique variables from the most general form and their corresponding constants in the most specific form. An example is given in figure 4.7.

The prototype then uses the most general form to try to produce another instance from the database such that X is instantiated to a value other than "christopher", while keeping the variable Y instantiated to the same value of "christine". If this instance can be generated, it is prompted for the user classification. If that instance cannot be generated, the prototype attempts to generate another instance with the variable Y binding to a different value other than "christine". If this instance still cannot be generated, then the database does not contain sufficient instances to allow the target rule to be induced. In this case, the most general and most specific form will be returned instead of a single target rule.

---

the new most general form after the second stage is:
related(X,Y):- parent(ringo,X), parent(ringo,Y).

the most specific form remains as:
related(christopher,christine):- parent(ringo,christopher), parent(ringo,christine).

the list of variable/constant pairs is:
[[X,christopher], [Y,christine]]

Figure 4.7: An example of a list of variable/constant pairs.

The final stage can be summarized in the following procedure:

```
--create a list of variable/constant pairs from
the most general and most specific forms
--with each pair in the list, do:
        --generates an instance such that the variable in this pair
        has a different value from the constant in the pair.
        --if an instance can be generated then
                --prompt the user for classification of the instance
                --if "yes", the variable is maintained.
                --if "no", the constant is maintained.
        --else
                --put this pair into another list of "undefined"
                --repeat the loop with the next pair
```

4.4. Current status of the prototype

Each component, analytical induction, extracting negative instances, and generating instances, has been implemented and tested separately as an individual module. However, there are difference between the interfaces for the different modules. For example, the analytical induction module outputs the rule and instances as clauses, but the next two modules accept the rule and instances in the form of a list. The user has to modify the format of output slightly from one module to another format for input to the next module. The interface problem is presently being worked on and will be resolved in the future.

To ensure that the prototype does not just solve the family-tree problem, it has been tested with other problems such as inducing specific rules for parsing context free grammar. Given a set of basic grammar and its database [Clocksin and Mellish 1981], as shown in figure 4.8, the prototype induces a specific rule for parsing a certain group of sentences. For example, the prototype induces a specific rule of [determiner,noun,verb,determiner,noun] from the sentence of [the,man,eats,the,apple].

```
sentence(S0, S) :-noun_phrase(S0, S1), verb_phrase(S1, S).

noun_phrase(S0, S) :- determiner(S0, S1), noun(S1, S).

verb_phrase(S0, S) :- verb(S0, S).
verb_phrase(S0, S) :-verb(S0, S1), noun_phrase(S1, S).

determiner([the|S], S).
noun([man|S],S).
noun([apple|S],S).
verb([eats|S],S).
verb([sings|S],S).

query :-sentence([the,man,eats,the,apple],[]).

the solution is :
        determiner([the,man,eats,the,apple],[man,eats,the,apple]),
                noun([man,eats,the,apple],[eats,the,apple]),
                verb([eats,the,apple],[the,apple]),
                determiner([the,apple],[apple]),noun([apple],[])

the induced rule is:
        determiner(S1,S2),noun(S2,S3),
                verb(S3,S4),determiner(S4,S5),noun(S5,[])
```

Figure 4.8: Inducing a rule for context free grammar.

4.5. Summary

In the first stage of analytical induction, the prototype finds the most general form of the version space from the generalized and/or tree. The most specific form of the version space is the solution to a problem found by the interpreter. The most general form is used in the next stage of empirical induction as the basis for finding negative instances and generating instances from the database. The most specific and most general form are refined at

the stage of empirical induction. The next chapter will discuss the limitations of the prototype, both in terms of its design and its implementation, and suggests further improvements. In addition, it also discusses several problems encountered in this project.

CHAPTER 5

Evaluation

This chapter evaluates the performance of the prototype, in both its design and implementation. Section 5.1 describes the limitations of its implementation. Section 5.2 examines the performance of each part of the prototype. Section 5.3 looks at the prototype as a whole. It describes where it is useful, and the assumptions that make it work. Section 5.4 suggests several issues for further investigation.

5.1. Implementation bottleneck

The prototype has been tested with a family-tree program, a sentence-parsing program, and several list-manipulation programs to gain some estimate of its generality.

The current prototype was slow in running the test programs. For example, parsing a sentence of "[the,man,eats,the,apple]" required only 0.016 cpu second running directly on the C-Prolog interpreter, but required 1.68 cpu second on the prototype running under similar loading conditions. The major inefficiency is due to the meta-interpreter built in the prototype. The meta-interpreter is required to extract extra information for induction.

There are reports on the inefficiency of using a meta-interpreter in the Prolog system [Sterling and Lee 1986], since a large fixed overhead exists. The meta-interpreter sets up the target program and then runs the program on the actual C-Prolog interpreter. This large fixed overhead may account for the inefficiency, especially when running small programs.

Another major inefficiency is in building a generalized and-or tree for the analytical induction. In order to carry out the analytical induction, it is necessary to record those basic clauses which are used in deriving a solution. However, getting those basic clauses in their original forms is not an easy task since the prototype has no direct information on how the C-Prolog interpreter operates. The C-Prolog built-in predicate "clause(Head,Body)" does not totally solve the problem. For example, given the head of "related(christopher,christine)", the predicate "clause" returns the body of "parent(Z,christopher), parent(Z,christine)". However it is the rule of "related(X,Y) :- parent(Z,X), parent(Z,Y)" which the prototype requires for analytical induction. In order to record the body of the rule "related(X,Y) :- parent(Z,X), parent(Z,Y)", the prototype has to build a separate generalized and-or tree. As discussed in section 4.1, the generalized and-or tree has to correspond with the solution tree. To prevent mismatch, the instantiations in both trees must be tested for equality at each step of instantiation. If there are similar clauses, testing equality may be time-consuming. A major improvement beyond the current prototype will be achieved by having some means of getting the basic rules used in the problem solving stage directly from the C-Prolog interpreter. In that case, programs can be run directly on the C-Prolog interpreter without the extra overhead of the meta-interpreter.

The current prototype can handle Prolog clauses involving "and" goals, "or" goals, and "not". It cannot handle "cut". It also cannot handle goals which require their variables to have specific constant values at the time of their instantiation. For example, any goal involving the system predicate of write(X) will fail because X must be instantiated to a constant value at the time of its call. In the solution tree, X would have a specific value. However,

in the generalized and-or tree, X would remain a variable. This would cause the system to fail.

5.2. Evaluation of each component

The first stage of analytical generalization is able to identify any constants and shared variables of such complicated clauses as:

cl(5.1)

related(R,J) :- parent(R1,J),parent(M,R1),parent(G,M),parent(R,G).

from the instance

cl(5.2)

related(ringo,john) :- parent(ringo,john), parent(mathew,ringo), parent(george,mathew), parent(ringo,george).

A user may have difficulty in deciding that there is a shared variable "R1", and that "R1" is different from another shared variable "R". To do this, he has to trace through the solution step by step. In that respect, analytical induction saves the user from the tedious effort of tracing through the solution manually.

Although the first stage of the prototype can identify any shared variables, some interpretation is still required by the user to determine the types of the variables. In Prolog, there are no types for variables. A variable can assume an integer value, a symbol constant or a list. For example, the prototype returns a grammar rule of:

cl(5.3)

sentence(X,[]) :- determiner(X1,X2), noun(X2,X3), verb(X3,X4), determiner(X4,X5), noun(X5,[]).

from the instance of

cl(5.4)

sentence([the,man,eats,the,apple]) :-
determiner([the,man,eats,the,apple],[man,eats,the,apple]),
noun([man,eats,the,apple],[eats,the,apple]),
verb([eats,the,apple],[the,apple]), determiner([the,apple],[apple]),
noun([apple],[])

The induced rule can only work when all variables in the rule have the input in the form of a list. However, the induced rule does not indicate that extra requirement. The user has to infer this himself by observing that all constants in the corresponding positive instance cl(5.4) are in the form of a list. The interpretation can be tedious if there is a mixture of different types in a rule.

In the first step of the second stage of the empirical induction, it is found that the past rejected solutions do not always exist as a source of negative instances. The family-tree program may generate some rejected solutions but the parsing program seldom generates any. Even for the family-tree problem, the rejected solutions are few and insufficient for the system to arrive at target rule. The original design idea is that if any rejected solution is generated during the search for right solution, then it is saved as negative instances. It does not have to be regenerated again for empirical induction. However, the result does not indicate any significant advantage to using this idea. In addition, this step assumes that all solutions in the solution trace are rejected ones rather than alternative right solutions. This restricts the user in interacting with the system.

The generation of instances in the second step depends on the database. A large database does not necessarily mean sufficient instances can be provided for all the different varieties of specific rules. The database may contain many similar instances which are useful for inferring some types of specific rules, but not others. To induce a variety of rules, the database must have a variety of instances. The database for the family-tree and parsing problems are not very large. Consequently, the prototype could generate sufficient instances for some simple specific rules but not for some of the more complicated rules.

In both the first and second step of the empirical induction, the prototype is restricted to using negative "near miss" instances for its specialization. That may restrict the prototype from inducing a target rule. A better strategy is to consider other negative instances besides those near miss instances. The major problem is to identify the essential discriminants which cause the instances to be negative. Some strategies such as the depth-first search or the Version Space method can handle this problem. However, the implementation is more complicated than the current prototype. The computation is also likely to be more expensive than the current one because it is necessary to process other negative instances.

5.3. Performance of the prototype

The following subsections describe where the prototype may be useful and the essential features behind the working of this prototype.

5.3.1. Where it can be useful

The purpose of the prototype is to induce specific rules to improve the system's problem solving efficiency. The choice of a domain can affect the usefulness of the prototype. One criterion for using the prototype is to choose domains where useful specific rules can be found. It is a subjective criterion because there is no precise definition for the term "usefulness". Perhaps the following examples may indicate some ideas of this criterion. In the family-tree problem, the "sibling", "grandparent" and "great grandparent" relationship are useful specific rules for solving typical cases of family tree relationship. Similarly, specific rules for defining certain sentence structure are useful for the parsing problem. The prototype has been tested with some list-manipulation programs such as append, and it does not infer any useful specific rules. For example, the prototype can infer a specific rule on how to append a single element to a list of three elements. However, that rule may be too restrictive because it can only be used for cases with a list of three elements.

The prototype uses analytical induction to narrow down the search space. Analytical induction requires the existence of background knowledge for its induction. In this prototype, the background knowledge consists of the basic rules and how these rules are to be used in deriving a positive instance. Therefore, this prototype is targeted at domains where basic rules exist. Also, the prototype must be able to record these rules when an instance is derived from them.

5.3.2. Restrictions behind the Prototype

Compared to running programs directly on the C-Prolog interpreter, the prototype is slow because it has to keep track of a lot of other information for induction. However, the prototype is able to induce the rules in a reasonable amount of time when tested with the sample programs. There are three restrictions on the prototype that make it a feasible system. These restrictions constrain the search space and prevent the prototype from facing some computationally intensive search.

The first restriction is that two logic clauses are considered equal only if they have the same subgoals arranged in the same order. This restriction limits the space of possible pairings. For example, consider two clauses A and B. With this restriction, the subgoals within the two clauses are compared with respect to their position. In other words, the first subgoal of clause A is compared with the first subgoal of clause B, the second subgoal of clause A with the second subgoal of clause B, and so on. Without this restriction, one subgoal of clause A can be paired with any one subgoal in clause B. It can be the first subgoal or the last one. To find that out, each subgoal of clause A has to be evaluated with every subgoal of clause B. That increases the computation complexity from order N to order N factorial (N!) where N is the number of subgoals in each clause.

The next restriction is that the prototype only considers the generalization of constants and variables and not functors. Consequently, the prototype has the bias of inducing rules in maximally specific form. For example, if there are two representations of the same rule such as the "great grandparent" relationship:

cl(5.5)

related(X,ZA) :- parent(X,Y), parent(Y,Z), parent(Z,ZA).

and

cl(5.6)

related(X,ZA) :- grandparent(X,Z), parent(Z,ZA).

where the goal of "grandparent(X,Z)" is represented by another rule

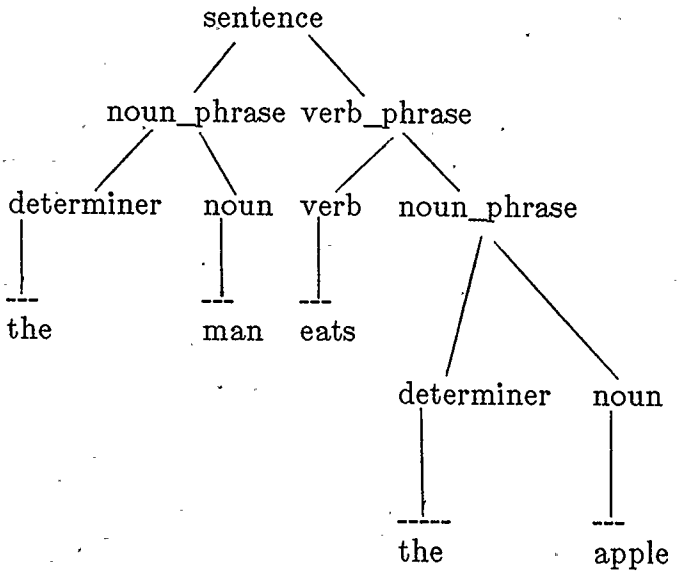
cl(5.7)

grandparent(X,Z) :- parent(X,Y), parent(Y,Z).

The two rules of (5.5) and (5.6) are interpreted to be the same since (5.6) can be converted to (5.5) by substituting its subgoal of grandparent. Under this circumstance, the prototype will induce the rule of (5.5) instead of (5.6) even though the rule (5.6) is simpler in structure.

There is an advantage of inducing a rule in a simple form such as rule (5.6). It is easier for a user to understand a rule conceptually in a simple form than in a maximally specific form. This understanding may give the users confidence in using the rules induced by the machine, especially for large and complicated domains.

However, to seek a rule in a simple form requires more computation to ensure that all the substitutions of subgoals do not create side-effects. Side-effects are due to the possibility that some subgoal may contain disjunctive clauses. Substitution of these subgoals may result in the rule being over-generalized. For example, consider a parse tree of the sentence, [the,man,eats,the,apple] in figure 5.1 [Clocksin and Mellish 1981].



1) sentence(S0, S) :-noun_phrase(S0, S1), verb_phrase(S1, S).

2) noun_phrase(S0, S) :- determiner(S0, S1), noun(S1, S).

3a) verb_phrase(S0, S) :- verb(S0, S).
3b) verb_phrase(S0, S) :-verb(S0, S1), noun_phrase(S1, S).

Figure 5.1: Parsing the sentence "the man eats the apple".

For the instance of [the,man,eats,the,apple], the induced rule in maximally specific form is

[determiner(S1,S2), noun(S2,S3), verb(S3,S4), determiner(S4,S5), noun(S5,[])].

The rule will still be correct if it is converted to the form of

[noun_phrase(S1,S2), verb(S2,S3), noun_phrase(S3,[])]

by using the clause (2) to replace the subgoals of "determiner", and "noun". However the rule is too general if it is further simplified into the form of

[noun_phrase(S1,S2), verb_phrase(S2,S3)]

by replacing the subgoals of "verb" and "noun_phrase" with the subgoal of "verb_phrase". It is because the "verb_phrase" clause (3) has another disjunctive clause (3.a): "verb_phrase(S0,S) :- verb(S0,S)". This disjunctive clause can introduce cases which the rule may effectively exclude.

To discover out these disjunctive clauses may require more computation. When the prototype generates the solution tree for the sentence [the,man,eats,the,apple], it only examines those paths leading to the solution. Consequently, the prototype does not know whether some of the intermediate clauses in the tree have disjunction. Since the prototype can only use those intermediate clauses with no disjunction to simplify the rule, the prototype has to go back and re-examine them. The re-examination of these clauses may be expensive. The prototype not only has to explore the search space of the solution, but also the search space of other alternatives.

Finally, there is an implicit restriction when generalizing constants into variables. There are only two levels in the generalization hierarchy for each variable. Either a variable is a constant value or it denotes a range of values. There is no intermediate concept between these two levels. Therefore, the prototype only has to generate two instances for each variable to decide whether it should be a variable or constant. Without this implicit restriction, the prototype would have to generate many possible instances for each variable, which would be combinatorially explosive. For example, consider an

integer variable "I" with a range of 1 to 10. With the restriction, only two values are needed to be picked from 1 to 10 to decide whether the variable "I" in a rule can remain as a variable. If an intermediate concept is allowed, the prototype has to generate all ten values to be absolutely certain that "I" can remain as a variable. It is quite possible that some intermediate concept such as "odd number between 1 and 10", or "even number between 1 and 10" can exist. The prototype would have to generate many instances to be sure and the computation would be increased.

5.4. Issues for future research

Several issues were identified during this project. They are described in the following subsections.

5.4.1. Selection and organization of specific rules

This prototype only addresses one aspect of the knowledge acquisition process: the process of inferring specific rules. Deciding which specific rules should be induced and what should be their organization in the rule base are also important parts of the knowledge acquisition process.

The prototype cannot induce a rule for every problem it encounters. The rule base would contain too many rules otherwise. Too many rules in the rule base would slow down the system performance because the system might spend too much time searching for appropriate rules to act upon. Therefore, the system preferably should induce those rules which solve typical cases. Unfortunately, the prototype does not know which cases are typical. It has to rely on the user's judgement in selecting those typical cases and their solutions. This issue is important for the self-learning programs such as AM.

Once the specific rules are induced, they have to be organized in the rule base. One common strategy is to arrange specific rules before basic rules. When a suitable specific rule is found for a problem, it will be used first. When all specific rules fail, then the system can use the computationally expensive basic rules to solve the problem. While this is a reasonable strategy, this arrangement may affect the induction of future rules. In particular, it may prevent induction of rules which are more general than those presently in the system. For example, if the "children of ringo" rule (5.8) is induced first, and put in front of the basic rules, then the prototype may *not* be able to induce the rule of "sibling" (5.9).

(5.8) related(X,Y) :- parent(ringo,X), parent(ringo,Y).

(5.9) related(X,Y) :- parent(Z,X), parent(Z,Y).

This is because the rule (5.9) is more general than (5.8). When the system tries to generate a solution for the "sibling" problem, the system picks the rule (5.8), as it is already in the rule base and in front of the basic rules. During induction, the prototype remembers the rule (5.8) and uses it as the most general boundary of the hypothesis space. However, this general boundary excludes the target rule of (5.9).

Putting the basic rules in front of all specific rules may prevent this problem but will destroy the usefulness of the specific rules. If the basic rules are put in front of all specific rules, the basic rules will be used on every occasion. The specific rules will be idle in the rule base.

5.4.2. Determining which techniques to use

One essential aspect of machine learning is to detect when a target rule is found. In the version space method, a target rule is found when the most specific and most general sets of the version space are equal and contain only one candidate. For analytical induction in strong domains, the rule induced from a positive instance can be confidently interpreted as the target rule because of strong background knowledge. However, if analytical induction is applied before empirical induction and the domain itself does not contain sufficient constraints, then the rule induced by the analytical process may not be the target rule. The system has to rely on the user to make the judgement. If the rule is not the target rule, then the user invokes the next stage of empirical induction to refine the rule.

The problem of deciding which techniques to use reflects one of the difficulties in machine learning. In some problems, such as verifying circuit design, and mathematical integration, there are well defined initial and final states. The problem is to find the solution path connecting the initial and final state. The knowledge of the final state can be used to judge along which solution path to proceed. However, the strategy of using the final state as the guideline does not work in machine learning. In most cases of machine learning, the final state, i.e. the target rule, is unknown. The version space method is better in the sense that it indicates the final state when the system reaches it. But until the system reaches the final state, the system only has a general bound of the final state. Consequently, it is difficult to use the final state to decide which technique is appropriate. Other criteria are required. Similarly, it is often difficult to conduct a search in machine learning because

of lack of constraints and guidance. The idea of organizing the search space and establishing criteria for conducting a search in NODDY [Andreae 1985] is the initial attempt to address the issue.

5.5. Summary

This chapter presents an evaluation of the current prototype. The prototype is targeted at domains where basic rules exist and useful specific rules can be found. Limiting the possible pairings, and allowing only conjunctive induction are used to prevent the prototype from facing combinatorial search explosion. Sample testing with the prototype indicates that it is an advantage to use analytical induction to narrow down the search space before empirical induction. However, the prototype needs improvement in future to allow other negative instances besides near miss instances in its specialization. Also the organization of specific rules in the rule base, and criteria for selecting appropriate techniques are two major issues requiring further investigation.

CHAPTER 6

Conclusion

The knowledge acquisition problem has been recognized as one of the major bottlenecks in building knowledge-based systems. One of the possible solutions to this knowledge acquisition problem is the use of machine learning techniques. This thesis describes an experimental prototype, which uses a combination of analytical and empirical machine learning techniques, to infer specific rules from solutions generated by basic rules of a domain.

Analytical induction is a knowledge-intensive approach. It makes uses of the background knowledge and the constraints of a domain, to guide its induction process. Given sufficiently strong background knowledge and constraints, it is possible to infer a rule from a single instance. Empirical induction is a data-intensive approach which relies on syntactic comparison of a number of positive and negative instances to infer a rule. It does not require any background knowledge other than the generalization hierarchy. These techniques are useful for two different types of domains. There is a need to address those domains where there is some, but insufficient, background knowledge. The use of only analytical techniques cannot support proper induction because of insufficient constraints. While empirical techniques can be used for these domains, the process of induction may involve a lot of instances, generated by the user.

This thesis explores the idea of using the analytical technique *before* the empirical technique for such domains. The analytical technique makes use of

any available knowledge and constraints to guide the induction. The empirical technique is then used to resolve those areas where available knowledge and constraints are lacking. The advantage of using the analytical technique before the empirical one is that the search space can be narrowed down by the analytical one. Consequently, it suffices for the empirical one to explore a much reduced search space.

The idea is used to build a prototype for inferring domain specific rules in a Prolog system. The domains are those which have general basic rules, but there is a requirement for inferring specific rules of arbitrary generality. A solution for a particular problem is first generated by the system using the basic rules. The prototype then infers a specific Prolog clause from the solution. The induction is characterized by a two-step process.

(1) Deciding upon any shared constants and variables in the target rule using analytical induction.

(2) Deciding upon the remaining constants and variables in the target rule by empirical induction.

In the first step, the prototype induces the existence of any shared variables or constants in a Prolog clause from both the generalized and solution and-or tree. The prototype then decides whether a constant in the solution can be turned into a variable in the target rule. The prototype stores the past rejected solutions as a source of negative instances, and generates new instances from the database of the current domain for its empirical induction. Constraints on possible pairings and a restriction to only conjunctive induction are used to prevent the prototype from facing a combinatorial search explosion.

Sample tests with the prototype indicate that it is an advantage to use the analytical techniques in the first step of induction. In the second step, the current prototype can make use of positive instances and near-miss negative instances to refine a rule. Further improvement is required for the prototype to make use of other negative instances besides near miss instances. Some problems are identified for further investigation. One problem is that the organization of specific rules in the rule base may prevent the induction of other rules. Another problem is to establish criteria to choose appropriate techniques.

Although machine learning is offered as a potential solution to the knowledge acquisition problem, much research is still needed in the area before it can be used practically. The current prototype works in a small set of domains and also imposes constraints to limit the search space. Violation of the constraints can easily make the search unmanageable.

References

Andreae, Peter Merrett (1985) "Justified Generalisation: Acquiring Procedures From Examples" Ph.D Dissertation, Dept. of Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Bratko,I. (1986) *Prolog Programming for Artificial Intelligence.* Addison-Wesley Publishing Co..

Bundy, A., Silver, B., and Plummer, D. (1985) "An Analytical comparison of some rule-learning programs" *Artificial Intelligence Journal, 27,* 137-181.

Carbonell, Jaime G., Michalski, Ryszard S., and Mitchell, Tom M. (1983) "Chapter 1: An Overview of Machine Learning" in *Machine Learning, An Artificial Intelligence Approach, Vol 1,* edited by Mitchell, Tom M., pp 3-24. Tioga Publisbing Co, Palo Alto, California.

Chandrasekaran, B. and Mittal, S. (1983) "Deep versus compiled knowledge approaches to diagnostic problem-solving" *International Journal Man-Machine Studies, 19,* 425-436.

Clocksin, W.F. and Mellish, C.S. (1981) *Programming in Prolog.* Springer-Verlay, Berlin, Germany.

Davis, Randall. and Lenat, Douglas B. (1982) *Knowledge-Based Systems in Artificial Intelligence.* McGraw Hill.

Dietterich, Thomas G. (1982) "Chapter XIV Learning and Inductive Inference" in *The Handbook of Artificial Intelligence, Vol 3,* edited by Feigenbaum, Edward A., pp 323-512. William Kaufmann, Inc., Los Altos, California.

Ellman, Thomas (1985) "Generalizing Logic Circuit Designs by Analyzing Proofs of Correstness" in *Proceedings of the Nineth International Joint Conference on Artificial Intelligence,* pp 643-646. August.

Feigenbaum, E.A. (1982) "Knowledge Engineering: The Applied Side" in *Intelligence Systems: the unprecedent and opportunity,* edited by Michie, Donald. Ellis Horwood Ltd, West Sussex, England.

Gordon, Mike (1985) "A Machine Oriented Formulation of Higher Order Logic" Report, Computer Laboratory, University of Cambridge, May.

Joyce, J. and Birtwistle, G. (1985) "Proving A Computer Correct in Higher Order Logic" Research Report No. 85/208/21, Dept. of Computer Science, University of Calgary, Calgary, Alberta, Canada, August.

Kowalski, Robert (1979) *Logic for Problem Solving.* Elsevier Science Publishing Co. Inc..

Langley, Pat (1985) "Learning to Search: From Weak Methods to Domain-Specific Heuristics" *Cognitive Science, No. 9,* 217-260.

Lebowitz, Michael (1986) "Integrated Learning: Controlling Explanation" *Cognitive Science, Vol 10,* 219-240.

Mahadevan, Sridhar (1985) "Verification-Based Learning: A Generalization Strategy for Inferring Problem-Reduction Methods" in *Proceedings of the Nineth International Joint Conference on Artificial Intelligence,* pp 616- 623. August.

Michalski, Ryszard S. (1986) "Chapter 1: Understanding the Nature of Learning: Issues and Research Directions" in *Machine Learning, An Artificial Intelligence Approach, Vol 2,* edited by Mitchell, Tom M., pp 3-26. Morgan Kaufman Publisbing Co, Los Altos, California.

Mitchell, Tom M. (1982) "Generalization as search" *Artificial Intelligence Journal, 18,* 203-226.

Mitchell, Tom M. (1983) "Learning and Problem Solving" in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* pp 1139-1151. August.

Mitchell, Tom M., Utgoff, Paul E., and Banerji, Ranan B. (1983) "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics" in *Machine Learning, An Artificial Intelligence Approach, Vol 1,* edited by Mitchell, Tom M.. Morgan Kaufman Publisbing Co, Los Altos, California.

Mitchell, Tom M., Steinberg, Louis I., and Shulman, Jeffrey S. (1985) "A Knowledge-Based Approach to Design" *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol PAMI-7, No.5,* September.

Mitchell, Tom M., Mahadevan, Sridhar, and Steinberg, Louis (1985) "LEAP: A Learning Apprentice for VLSI Design" in *Proceedings of the Nineth International Joint Conference on Artificial Intelligence,* pp 573-580. August.

Norman, Donald A. (1980) "Twelve Issues for Cognitive Science" *Cognitive Science, No. 4,* 1-32.

Rosenbloom, Paul S., Laird, John E., McDermott, John, Newell, John, and Orciuch, Edmund (1985) "R1-Soar: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture" *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol PAMI-7, No.5,* September.

Samuel, A L. (1959) "Some studies in machine learning using the game of checkers" *IBM J. Research and Development, 3,* 210-229.

Shapiro, Ehud Y. (1982) *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachusetts.

Simon, Herbert A. (1980) "Cognitive Science: The Newest Science of the Artificial" *Cognitive Science, No. 4,* 33-46.

Simon, Herbert A. (1983) "Why Should Machines Learn?" in *Machine Learning, An Artificial Intelligence Approach, Vol 1,* edited by Mitchell, Tom M., pp 25-38. Tioga Publisbing Co, Palo Alto, California.

Stefik, M., Bobrow, D., Bell, A., Brown, H., Conway, L., and Tong, C. (1981) "The Partitioning of Concerns in Digital System Design" (VLSI-81-3), Xeror Palo Alto Research Report, December.

Sterling, Leon and Shapiro, Ehud (1986) *The Art of Prolog: Advanced Programming Technique.* MIT Press, Cambridge, Massachusetts.

Sterling, Leon and Lee, Marucha (August 1986) "An Explanation Shell for Expert Systems" *Computational Intelligence, Vol 2, No. 3,* National Research

Council of Canada, Ottawa, Canada.

Utgoff, Paul E. (1986) "Shift of Bias for Inductive Concept Learning" in *Machine Learning, An Artificial Intelligence Approach, Vol 2,* edited by Mitchell, Tom M., pp 107-148. Morgan Kaufman Publisbing Co, Los Altos, California.

VanLehn, Kurt (1983) "Felicity conditions for human skill acquisition: validating an AI-based theory" Research Report CIS-21, Xeror PARC, Palo Alto, California.

VanLehn, Kurt (1987) "Learning One Subprocedure per Lesson" *Artificial Intelligence Journal, 31,* 1-40.

Winston, Patrick H. (1975) "Learning Structural Descriptions from Examples" in *The Psychology of Computer Vision,* edited by Winston, Patrick H., pp 157-209. McGraw-Hill, New York.

Winston, Patrick H. (1984) *Artificial Intelligence.* Addison-Wesley Publishing Co., Reading, Massachusetts.