UNIVERSITY OF ALBERTA

An Adaptive Approach for Acquiring Missing Knowledge

BY

Alan D. Sharpe

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1993

ISBN 0-315-88101-1

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Alan D. Sharpe

TITLE OF THESIS: An Adaptive Approach for Acquiring Missing Knowledge

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1993

(Signed) A Sharpe . . . . . .

Alan D. Sharpe
903 - 7361 Halifax Street
Burnaby, British Columbia
V5A 4H2

Date: July 5, 1993 . . . . . .

...such thing as a stupid question?

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Adaptive Approach for Acquiring Missing Knowledge** submitted by Alan D. Sharpe in partial fulfillment of the requirements for the degree of Master of Science.

R. Goebel

Dr. Randy Goebel (Co-supervisor)

Peter van Beek

Dr. Peter van Beek (Co-supervisor)

Dr. Michael R.W. Dawson (External)

Renée Elio

Dr. Renée Elio (Examiner)

Date: July 5, 1993

To Mom and Dad who are always on my side.

Abstract

No matter how sophisticated the reasoning mechanism is in a knowledge-based system, its performance will always be limited by the quality of its store of domain dependent knowledge – its knowledge base. The acquisition of this knowledge has long been considered a "bottleneck" in the development of knowledge-based systems. Although much effort goes into eliciting and encoding the knowledge base of a system, there will almost always be some omissions and errors. It is therefore useful for a knowledge-based system to continually acquire new knowledge during its operation.

THINK is a framework to integrate interactive machine learning into a knowledge-based system which allows a system to incrementally acquire new knowledge when the current knowledge base is inadequate to solve a given problem. This knowledge acquisition method uses the current problem context and line of reasoning to hypothesize missing items of knowledge. Hypotheses are generated through a generalized abduction method and then subjected to a neural net based plausibility test prior to presentation to an expert user. The knowledge base is updated according to the response. If hypotheses are rejected then alternate ones are generated until accepted. In interactions between expert and system, learning is achieved through experience so more plausible hypotheses are presented earlier. This learning takes the form of training the neural net with previous hypothesis-response pairs.

An implementation of THINK is described along with experimental results which indicate the validity of the concept.

Acknowledgements

I would like to thank my supervisors, Randy Goebel and Peter van Beek for their guidance and support during my research. Peter ensured that I remembered that computing science is, indeed, a science; Randy kept me on my toes with respect to what not just he, but others in general, would expect to see in my work; and they both provided an excellent research environment and great freedom for my work while still keeping me focused enough so that I actually accomplished something.

I would also like to thank the other members of my examining committee, Renée Elio and Mike Dawson, who provided careful, constructive criticism which improved this document.

The rest of the AI gang deserves my thanks as well. Aditya, Pablo, Hong, Zhong, Shurjo, Stefan, and others were good friends and always made time in the lab more enjoyable. From discussions about our different work to occasional indulgence in games and other less productive activities, the group got along extraordinarily well. I particularly thank them for how they put up with me during my experiments when I would "hog" all the CPU and memory on the AI machines. Not only did they tolerate it but they made sure I knew that they understood.

To those mentioned above and countless others whom I like to call friends, I am thankful for the support and words like, "You'll do OK", at times when I needed a boost.

Finally, to the National Sciences and Engineering Research Council of Canada, the Alberta Heritage Scholarship Fund, and the University of Alberta, I gratefully acknowledge the financial support I received.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

A knowledge-based system (KBS) is a computer system which use stores of domain-specific knowledge to solve problems. The store of knowledge, called the *knowledge base* (KB), is applied to problem instances with some problem-solving method. This method is embodied in a separate part of the system called the *inference engine*. Thus, the inference engine applies the knowledge within the KB to a given problem instance to produce a solution. The term "inference engine" derives from the use of various logical inference methods to apply knowledge to a problem. An important specification of a system is the knowledge representation (KR) scheme. Although the content of the KB is domain-specific, its representation scheme is system specific to enable interpretation by the system's inference engine. A view of a KBS in its simplest form is given in figure 1.1.

One central area of KBS research is *knowledge acquisition* (KA). The problem of acquiring the domain-specific knowledge for a KBS has been recognized as a "bottleneck" in the development of practical systems since the DENDRAL days [5]. The sheer volume of knowledge to be acquired for a practical system makes it inevitable that there will be errors and omissions. Omissions can also occur when domain experts omit details they think are unimportant to the inference process in order keep

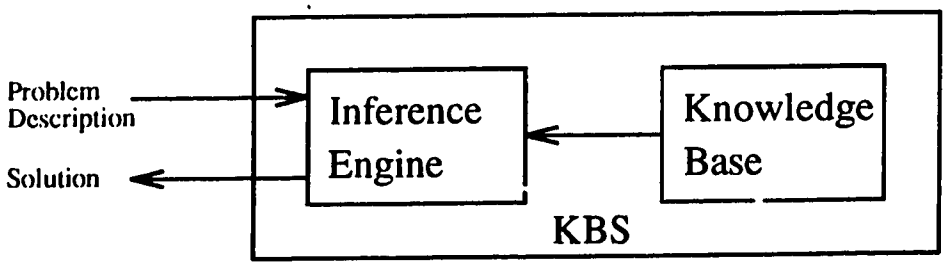

Figure 1.1: Simple KBS View

things simple for the knowledge engineer. Some details just never naturally come to the mind of the expert until they are needed in an actual problem case. The expert may overlook some details because the details are so obvious (to the expert at least). The result is that while the KB may allow adequate or even expert performance, it can always be improved.

Buchanan [6] notes that the KA problem has several dimensions and naturally goes through phases. He likens it to a chess match with an "opening," a "middle game," and an "end game." In the development of R1, McDermott [16] noticed a similar requirement to go through a "rule splitting" refinement phase. However the phases are identified or characterized, it is natural to use a specific KA approach for each. A phase might be characterized by an emphasis on filling in missing knowledge. Missing knowledge will be manifest in one of two possible ways:

1. The KBS will find no solution to a problem because of the missing knowledge.

2. The KBS will provide an incorrect solution or a less than best solution to the problem.

In the second case, the system could allow the user to reject these solutions one by one until the first case arises. For example, a system might be expected to generate A and B as possible diagnoses for some given symptoms, but only comes up with A because of missing knowledge for diagnosis B. Closer examination by the user might result in A being rejected. Then there will be no other possibilities to investigate although there should be. Consequently, it makes sense to take a close look at the first manifestation, above.

A robust system will have an alternate way of proceeding when faced with the first situation, above. Any such contingent reasoning is likely to be less effective and less reliable, and therefore less desirable, than the primary reasoning mechanism. Otherwise, it would be used primarily. It makes sense, therefore, to use the contingent reasoning not just to continue with the problem or subproblem at hand, but rather to "repair" the problem with the primary reasoning mechanism — to find the missing knowledge required for the task at hand. After the missing knowledge has been inferred, it can be verified with an expert user, to give it the same credibility as the extant system knowledge. A good, robust KBS will have a way of automatically switching, as required, into a mode of incremental KA *directed* at the missing knowledge for a given task *within* the context of that task. Not only will the task context be set up in the system for this directed KA, but will also be fresh in the user's mind.

THINK (Thoughtful Hypotheses for Inquiring on New Knowledge) is a framework to integrate a directed, and interactive KA subsystem as motivated above. Davis encourages these basic ideas of integration and expert interaction in [7].

1.2 Objective and Contribution

The objective and contribution here is to show that an adaptive (learning) KA subsystem which acquires *missing* knowledge can be integrated into a KBS which has knowledge based on associational links. Such KBSs form a significant subset of all KBSs. They include systems, such as CASNET [25] which is based on causal-associational links; and production systems, such as R1 and XCON [16] whose production rules can be viewed as associations between conditions and actions. Any KBS in this subset can derive the additional robustness and other benefits described in the previous section with the inclusion of a KA subsystem similar to the one presented here.

Previous work in KA is focused on either initial KA for a KBS (e.g., [17, 4]) or on *refinement* of knowledge bases. The work here clearly falls into the refinement category. Other work in this category is directed at refining existing knowledge items

(e.g., [22, 10, 8, 13]); and guiding an expert to general areas of the domain which need attention to *gradually* elicit new knowledge items (e.g., [7, 14]). There are no existing systems, known to the author, which infer, on their own, new[1] KB items complete and ready to be incorporated into the KB and only requiring the expert to approve or reject (or perhaps provide a certainty factor for) the new item.

It is the author's thesis that a useful KA subsystem for acquiring missing associational knowledge is possible which combines abduction and the application of Adaptive Logic Networks[2] (ALNs) to perform adaptive hypothetical inference to generate candidate missing KB items to present to an expert for verification.

To show that such a KA subsystem is possible and viable, a particular such system is implemented and tested on the following predictions:

1. The system will be capable of acquiring missing knowledge required for a given problem, putting it to an expert for approval, incorporating it into the primary KB, and allowing the primary KBS reasoning mechanism to continue to successfully solve the given problem.

2. Because of its adaptive nature, the KA subsystem will improve with increased use. That is, a more mature KA subsystem will present fewer candidate KB items which are rejected by the expert.

1.3 Overview of Chapters

The remaining chapters are organized as follows: The next chapter describes the general THINK framework along with its scope of application.

Chapter 3 provides preliminary background information on abduction and on the particular type of neural net (ALN) used in the experimental system.

Chapter 4 describes the experimental system in detail. It provides more detail on the architecture of the system, and implementation specific details and assumptions

[1] By "new" it is meant that the inferred KB items are not revisions of existing KB items.
[2] A kind of neural network presented later

for the individual components.

Chapter 5 describes related work in the area intelligent knowledge acquisition. This chapter also provides comparisons between the related work and that presented here.

Chapter 6 describes the experiments conducted on the implemented system. Results of the experiments are provided as well as a qualitative evaluation of the system in light of the results.

The final chapter provides a general appraisal of the THINK concept. Limitations and strengths are highlighted and suggestions for further work are given.

Chapter 2

The THINK Framework

2.1 Scope of Application

The THINK framework applies to KBSs which have knowledge based on self-contained associational links. Many KBSs are based on some form of associational knowledge. The designers and users of these systems have their own various meanings for these links which are usually independent of how they are used. Whether the links are "causes", or inversely "evidences" , or orthogonally "allows", or generally "explains", a common basic inference of, say, backchaining can result in a useful KBS. All that need change is the name of the link and which associations are made in the KB. Here, such associational links will be represented with a $\rightarrow$ connective.

As an example consider an "animal world" KB with the following associations:

$$\{bird(X) \rightarrow fly(X), dog(X) \rightarrow run(X), cat(X) \rightarrow leap(X)\}.$$

Let the KBS also know: $\{bird(tweety), dog(lassie), cat(morris)\}$ as observations. If the KBS is tasked to explain $fly(tweety)$, backchaining can be used by finding an association whose right hand side matches $fly(tweety)$. Since the first association matches, the system then continues by trying to explain $bird(tweety)$. This is explained simply by citing it as an observation.

Suppose such a system is unable to provide an explanation for some observation.

This would be the case in the above example if the association, $bird(X) \rightarrow fly(X)$ was missing from the KB. In this situation, the system might *suggest* to an expert user a candidate KB item which, if added to the KB, would help in the explanation. Suppose the example system is supposed to explain $fly(tweety)$ and is missing the first association. It cannot explain $fly(tweety)$. At this point the system might suggest to an expert that $bird(tweety) \rightarrow fly(tweety)$ (or some more general form of it) be added to the KB. Based on the expert's response, the KB could then be automatically updated and reasoning could continue. This sort of suggestion would only be necessary if the system is unable to find any explanation for $fly(tweety)$.

There is an obvious problem with this simple scheme, though. If the KB or set of observations is large, then the number of possible suggestions is prohibitively large. No expert user wants to be deluged with many thoughtless suggestions:

- Does $dog(lassie) \rightarrow fly(tweety)$ hold? Answer: No.
- Does $cat(morris) \rightarrow fly(tweety)$ hold? Answer: No.
- Does $bird(tweety) \rightarrow fly(tweety)$ hold? Answer: Yes.

This could be much worse if either the KB or the observation set were large.

2.1.1 Thoughtful Hypotheses

The solution to the above problem is to intelligently order the suggestions so that the right one is likely asked early on. The determination of whether the response is likely to be positive or negative is where intelligent hypothetical inference can be used. Based on a history of previous suggestion-response pairs, each possible suggestion is checked for some kind of support. Only if there is support, is a suggestion actually made to an expert.

THINK is a framework for performing this secondary inference directed at incremental KA. There is a separate adaptive knowledge-based subsystem with its own KB

for this purpose. The subsystem is meant to generate either links or literal elements[1] as needed.

2.2 KA Subsystem

When the primary reasoning system fails to find an explanation for a problem, it is then up to the KA subsystem to acquire missing items which will allow an explanation. There are two kinds of possible missing items:

Missing literals: These are usually observations which should have been given for the current problem. These could include "askables" which are common in expert systems.

Missing links: These are usually missing KB items.

The KA subsystem can generate either kind of these items as candidates.

The technique for acquiring these missing items is generate and test. Based on its KB, the primary KB, and the current line of reasoning, the KA subsystem generates candidate knowledge elements which would allow an explanation. Testing is just asking an expert if the hypothesized missing items should in fact be added to the KB or problem description.

The expert is expected to provide two responses to each presented item:

1. A categorization of good or poor of the item as a suggestion. This categorization could be based on anything the expert believes, particularly if this basis is something not naturally represented in the primary KB. For example, it could be a "poor" suggestion to ask something the expert could never know. If an expert could know an answer (after a reasonable amount of effort) then it would be a "good" item to suggest. The idea here is to develop a classification scheme which is not easily represented in the language of the main domain knowledge.

[1] A literal is just an atomic proposition or its negation. E.g., $bird(tweety)$ and $\neg dog(morris)$.

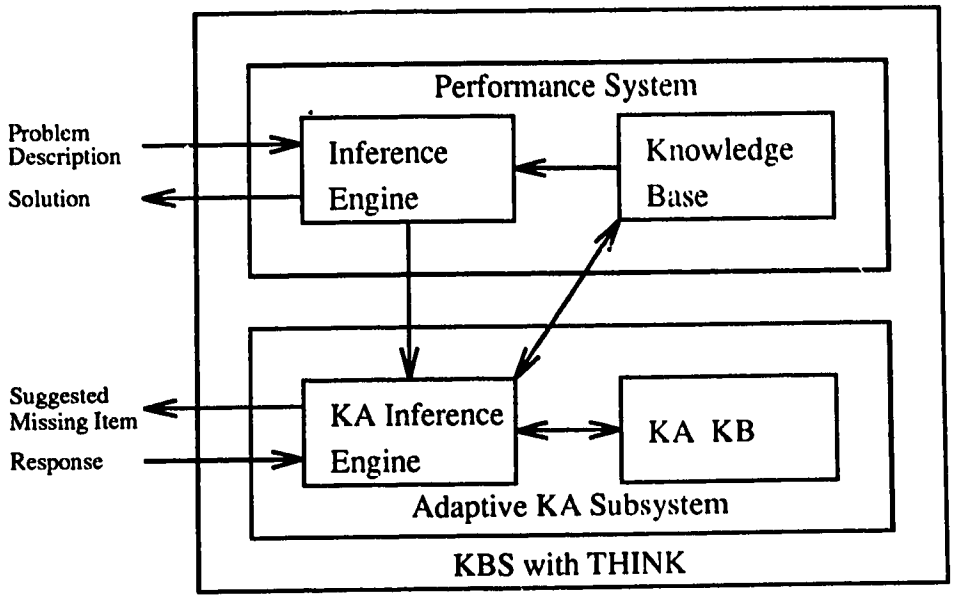

Figure 2.1: View of General KBS with THINK

2. A response as to whether the suggested item should be a new KB item, new observation, or neither. The response here may include parameters for the new item (e.g., a certainty factor).

The first response is not part of the knowledge acquisition, proper. Rather, it is used in adaptation of the KA subsystem. The order of generation of candidates is based on previous expert categorizations. It is important to have a two part response mainly to allow for "good" candidates which, nonetheless, are just not true. In this case the "good" part of the response will be used in adapting the system to tend to ask a similar question again in a different case even though the main (second part) response the first time was negative. In this way, the KA subsystem is one which *learns* from experience to ask "good" questions earlier than "poor" ones.

2.2.1 Architecture

The architecture for THINK is an extension of that for a KBS to include the KA support described above. This architecture is depicted in figure 2.1. The KA inference engine receives, from the performance inference engine, the current task or

subtask that cannot be performed. The KA inference engine then uses the main KB augmented with its own KB to generate a suggested new knowledge item to which an expert responds. Based on this response, the KA inference engine updates the main KB and through adaptation also updates its own KB.

2.3 Summary

The following points summarize the THINK framework:

- THINK applies to systems with a KB mainly consisting of associational links. Other ancillary knowledge may be present, though.
- When the performance system is unable to solve a problem, the KA subsystem is called upon to acquire missing knowledge.
- The KA subsystem uses its own hypothetical inference procedure to generate candidate new KB elements which will allow the performance system to solve its problem. This subsystem uses its own specialized KB along with the the KB of the performance system. Based on this knowledge, KB candidates are generated in an order of decreasing quality as indicated by past experience.
- The expert user responds to candidates in two parts. The first part indicates the quality of the candidate and is used to adapt the KA subsystem. The second part is an indication of how the candidate should be incorporated into the KB of the performance system.

This is a very general framework. Any kind of adaptive subsystem can be used which performs as described above. It is *within* this framework that systems can be developed which support the authors thesis: a useful KA subsystem for acquiring missing associational knowledge is possible which combines abduction and the application of ALNs to perform adaptive hypothetical inference to generate candidate missing KB items to present to an expert for verification.

Chapter 3

Background

This chapter provides introductory descriptions of abduction and ALNs. These are central to the mechanism of the adaptive KA subsystem presented here. Abduction is used to generate candidate new KB items. ALNs are used in ordering the candidates by quality.

3.1 Abduction

This section describes the inference method of abduction. First, it is presented as Peirce [21] originally did. Then, the contemporary AI view is presented. Although the modern view seems less general than Peirce's, it is argued that it need not be so. Finally, the mechanism of abduction is related to the common AI theme of search. This relationship is very evident in THINK.

3.1.1 Peirce

The notion of abductive inference was first put forth by the philosopher, Charles S. Peirce [21]. He describes abduction as the initial *formation* and consideration of a hypothesis. Peirce distinguishes this from *verifying* a hypothesis which is an *inductive* process. As Peirce puts it, a hypothesis is any "supposed truth from which would result such facts as have been observed". In short, a hypothesis is a supposition

which explains something. Thus, abduction is the *generation* of any supposition which explains something – quite a general kind of inference.

Strictly speaking, there are no restrictions on hypotheses (and therefore abduction) except that it must explain something. Peirce does, however, provide some guidelines and caveats which should help in performing "good" abduction. Some of these are outlined below.

1. "... abduction commits us to nothing," says Peirce. The point is that no conclusion is drawn by abduction. It merely generates a hypothesis to be further investigated. Abduction is sometimes introduced by others as "an unsound rule of inference" (eg: [11]). This is a rather unfair sounding introduction. Abduction is a best introduced as a method to generate hypotheses. Although a "method" can be though of as a rule and a hypothesis is a kind of inference, the connotation is lost in the "rule of inference" description with which people easily associate the idea of faulty reasoning. Abduction should be used freely without fear of generating a faulty line of reasoning, but with the understanding of what abduction produces.

2. Any hypothesis which explains something is a valid product of abduction. For example, to hypothesize that it had been raining recently would be an abduction to explain some wet ground. This is not very useful, however, if one is trying to find an explanation for a broken window. Here, the point is that abduction should be directed toward some end reasoning goal; "that ought to be done which is conducive to a certain end." In short, the application of abduction should be goal-directed.

3. A hypothesis need not explain just some observed phenomenon; it may be more generally applicable. Indeed, a more general hypothesis may be much more useful. For example, upon observing the relationship between the pressure and volume of a sample of helium gas, one might hypothesize that the Boyle's Law relationship will always hold for this sample; or one may hypothesize that it

holds for any helium sample, or any gas sample, or even any matter sample. Even the last hypothesis is useful if only to incite investigation towards its refutation.

4. Prior likelihood of the truth value of a hypothesis should not enter into abduction. Consider the popular Sherlock Holmes quote: (roughly) "When the impossible has been eliminated, whatever remains, however unlikely, must be the truth." Clearly, it is important for abduction to produce unlikely hypotheses as well as likely ones lest there sometimes be nothing remaining after elimination of the impossible.

5. Hypotheses definitely should be verifiable. It is of no use beyond mere pondering to suppose an explanation which cannot be checked out. It is not necessary to be able to verify a hypothesis directly, though. It is acceptable to make hypotheses for which there will be indirect evidence for their confirmation. It is good abduction to suppose earlier rain explains wet ground. Although one cannot go back in time to observe the rain directly he can check weather reports and ask around if it had rained. On the other hand it is not good abduction to suppose that the ground just magically became wet. This is not because it seems unlikely but because there is no way of checking out the theory.

6. Abduction can include a *preference* for one hypothesis over another. This preference, though, must not be based on any knowledge bearing on the possible truth of the hypothesis. This is in keeping with the first and fourth points above. Consideration of the truth or likelihood of a hypothesis is done *after* abducing it – in the *verification* of it. Some possible bases for preference have been mentioned above. A more general hypothesis could be preferred over a more special one or vice versa; or a more easily verifiable hypothesis could be preferred over one more difficult to check out. Even aesthetics may provide grounds for preferring one hypothesis over another.

3.1.2 Contemporary AI Abduction

Many AI tasks may be viewed as finding explanations. For example, diagnosis is finding an explanation for observed symptoms; recognition or classification is finding an explanation for observed characteristics; plan recognition is finding an explanation of observed actions; and natural language understanding is finding an explanation (meaning) for a sequence of words. AI practitioners often use abduction as one of the methods in the search for explanations. Toward the end of automating abduction, it has been formalized as a clean, simple inference rule [11, 15]:

$$\frac{\Phi \rightarrow \Psi,\ \Psi}{\Phi} \tag{3.1}$$

In knowledge-based systems, the $\rightarrow$ connective is not taken as material implication but as some kind of explanatory connection such as "causes", "may cause", or "allows". Thus, Φ is inferred (hypothetically, not conclusively) as an hypothetical explanation for Ψ.

At first this may seem to be a restricted version of Peirce's abduction, for it does not provide for the hypothesis of a link like $\Phi \rightarrow \Psi$. For example, $bird(tweety)$ may be hypothesized from $fly(tweety)$, $bird(X) \rightarrow fly(X)$, it does not seem possible to infer hypothetically $bird(X) \rightarrow fly(X)$ from $bird(tweety)$, $fly(tweety)$. Actually, though, such an inference is valid with this formalized abduction. Most reasoning systems have this rule implicit: $(\Phi,\ \Phi \rightarrow \Psi) \rightarrow \Psi$. Substituting into the abductive rule 3.1 gives

$$\frac{(\Phi,\ \Phi \rightarrow \Psi) \rightarrow \Psi,\ \Psi}{\Phi,\ \Phi \rightarrow \Psi} \tag{3.2}$$

Adding Φ to the antecedent and removing it from the consequent, as well as removing the implicit rule from the antecedent gives

$$\frac{\Phi,\ \Psi}{\Phi \rightarrow \Psi} \tag{3.3}$$

Abduction need not be restricted to producing but one hypothesis at a time. Just removing the implicit rule from 3.2 gives

$$\frac{\Psi}{\Phi,\ \Phi \rightarrow \Psi} \tag{3.4}$$

Also, Φ itself could be a conjunction of, say, $\Phi_1, \ldots, \Phi_n$. If the first l of these are given, known, or supposed, then they can be removed from the consequent of 3.1, and 3.4 giving

$$\frac{\Phi_1, \ldots, \Phi_l,\ (\Phi_1, \ldots, \Phi_n) \rightarrow \Psi,\ \Psi}{\Phi_{l+1}, \ldots, \Phi_n} \tag{3.5}$$

and

$$\frac{\Phi_1, \ldots, \Phi_l,\ \Psi}{\Phi_{l+1}, \ldots, \Phi_n,\ (\Phi_1, \ldots, \Phi_n) \rightarrow \Psi} \tag{3.6}$$

respectively. Abduction can be chained by using the results of one abduction as hypothesized premises to others. For example 3.4 might be chained with itself for n applications. Let Φ_i and Ψ_i be Φ and Ψ from the ith application. Feeding Φ_i into the $(i+1)$st application as Ψ_{i+1} and removing redundant hypotheses (Φ_is) gives

$$\frac{\Psi}{\Phi_n,\ \Phi_n \rightarrow \Phi_{n-1} \rightarrow \cdots \rightarrow \Phi_1 \rightarrow \Psi} \tag{3.7}$$

It should now be clear that by freely combining the above inference rules, formalized abduction is as general as Peirce's abduction. That is, any explanatory hypothesis can be generated through formalized abduction.

3.1.3 Abduction and Search

Abduction and search are closely related. Consider a search state space with state nodes, N_i, which each expand via allowable state transitions to a set of nodes, $N_i^1, \ldots N_i^{n_i}$. A search specifies a start node, N_s and also specifies (perhaps indirectly) a set of goal nodes N_G. The aim of a search is to find a state transition path from N_s to some goal node, $N_g \in N_G$. Let P_i denote the proposition, "N_i is on a path from N_s to a goal node," and let P_i^j denote the same for N_i^j. It is then true that $P_i^j \rightarrow P_i$ for $j \in [1..n_i]$. Now, either P_s is known true, or it is assumed true to begin the search. N_s gets expanded and $N_s^1, \ldots N_s^{n_s}$ as new search frontier nodes. This also identifies the rules, $P_s^1 \rightarrow P_s, \ldots P_s^{n_s} \rightarrow P_s$. It is then possible to apply abduction:

$$\frac{P_s, P_s^j \rightarrow P_s}{P_s^j} \qquad (j \in [1..n_s]) \tag{3.8}$$

One application of this abduction is exactly the process of selecting the next node to expand. Thus the whole search process is a series of forward abductive chaining steps. It ends when a rule, $P_i^j \rightarrow P_i$ is identified where $N_i^j \in N_G$. At this point P_s can be shown deductively and the chain $P_g \rightarrow \cdots \rightarrow P_s$ corresponds to a path in the search space. Thus, search is abduction.

If search is abduction, then is the reverse true? Is abduction search? The answer is yes. A state in the state space is specified by the pair, $\langle F, T \rangle$ where F is the set of known facts and T is a set of currently considered hypotheses. F is static so is the same for all states. A state transition is the application of any abductive inference rule resulting in an addition to T and so a new state. Although not mentioned by Peirce in [21], an important caveat is that any addition to F (abductive inference) should not result in any inconsistent set of hypotheses. That is, $F \cup T$ should remain consistent[1]. What remains is to show how start and goal states are specified for a search. Recall one of the principles of abduction from section 3.1.1 is that abduction should be directed to some end - that there should be something specific to be explained. Let that item be denoted, G. If G is already known to be true (just an explanation is desired) then the start state, is $\langle F, \phi \rangle$. Otherwise, G is assumed and the start state is $\langle F, \{G\} \rangle$. An end state is one where there is satisfactory evidence for the non-redundant items in T.

3.2 Adaptive Logic Networks

This section describes Adaptive Logic Networks (ALNs) which are a particular kind of neural network. A brief definition of general neural nets is first given, followed by the distinguishing characteristics of ALNs. Included, is a brief motivation for the use of ALNs.

[1]This may seem very much like reasoning in the Theorist framework [23]. The important distinction is that in Theorist, additions to T are drawn from a predefined set of allowable hypotheses; here additions to T are drawn from abductive inference. Thus, Theorist is very much a default reasoning framework and not so much and abductive one.

3.2.1 Neural Nets

Pertinent elements of the neural net definition from [12] are summarized below:

1. A neural net is a directed network of processing elements.

2. Connections in the network are unidirectional and pass the output of one processing element as the input of another. Some connections serve as inputs to the overall net and pass net input input values as input to processing elements. Some connections serve as outputs from the net and pass output from processing elements as net output.

3. Each processing element may have any number of input connections.

4. Each processing element may have any number of output connections but they each must carry the same output value. That is, a processing element computes only one function and this value "fans out" over the element outputs.

5. The output of a processing element is a function of only its inputs and the local state of the processing element.

6. The local state of a processing element may be changed as a function of the current state and of the inputs to the processing element.

The above defines a neural net as it operates. Neural nets can also be adapted (trained) automatically by an adaptation process. The function from the last two points (called a *transfer function*) above may be parameterized and the parameters my be adjusted during the adaptation process. This process (which partly distinguishes a neural net) typically consists of presenting the network with inputs as well as known correct outputs. This information is propagated throughout the network according to the process resulting in modification of the transfer function parameters. This process is repeated over many input-output examples (a *training set*) until the net performs acceptably. More details of adaptation may be found in Chapter Three of [12].

3.2.2 ALN Characteristics

ALNs were primarily developed at the University of Alberta by William Armstrong and are well described in [2]. The overall characteristic of ALNs is simplicity. This can be seen in the characteristics below which relate respectively to the general definition points in section 3.2.1.

1. An ALN is a binary tree of processing elements.
2. Connections are directed UP the tree (towards the root). Inputs to the network are passed to the leaves of the tree and there is a single output at the root.
3. Each processing element has exactly two input connections.
4. Each processing element has exactly one output connection.
5. Inputs to each processing element are boolean values (0 or 1). The output of a processing element is one of four possible boolean transfer functions: AND, OR, LEFT, or RIGHT. There is no local state information as there is only one state of the processing element.
6. The local state does not change during operation.

Because the four possible transfer functions are each monotonic[2] and monotonicity is preserved through composition, ALNs are monotonic. This may or may not be a desirable feature. Any function can be realized, though, by providing the complements of the arguments as well as the arguments themselves as ALN input. With AND and OR functions available and the availability of input complements, it is clear that any boolean function can be realized with a large enough ALN.

In order to realize some functions it is necessary to replicate some of the inputs to more than one leaf processing element. In fact, a multiplicity of a dozen or more

[2]A boolean function is monotonic if a single change of any argument from 0 to 1 will not cause the output to change from 1 to 0 and vice versa.

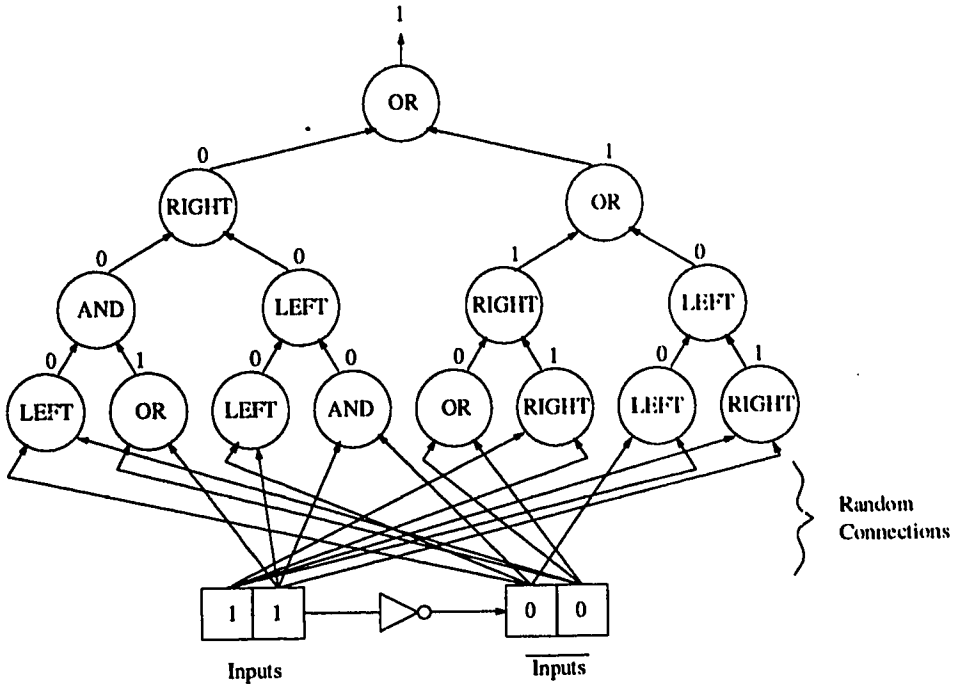

Figure 3.1: Nonmonotonic ALN with 8 leaves and 2 inputs

is sometimes needed in order to have a large enough ALN structure. Although input connections and processing element transfer functions can be defined manually, it is common (as it is in general for neural nets) for the transfer functions to be automatically defined through an adaptation mechanism. Initial connections from inputs to leaf elements are random. If the adaptation procedure is unsuccessful (the network does not perform acceptably after a reasonable training period), another random input connection scheme can be tried or the ALN can be enlarged increasing the multiplicity of inputs. An example ALN is depicted in figure 3.1.

3.2.3 Insensitivity of ALNs

Bochmann and Armstrong have shown that ALNs are insensitive to small changes in input [3]. That is, a small perturbation in the input bit pattern will result in a small probability of a change in the ALN output. This property of ALNs makes them good for pattern recognition. After training on a small set of exemplar bit input patterns, and then presenting the ALN with a new pattern which is close (in Hamming distance)

to an exemplar, it is likely that the ALN will classify the new pattern in the same way as the similar exemplar. This brings to light an important principle in the use of ALNs for pattern recognition: Patterns should be encoded into ALN inputs in such a way that "similar" patterns have encodings which are close in Hamming distance. This allows a more natural adaptation of the ALN to recognition[3]. For example, to use ALNs for case retrieval in a case-based reasoning system, case indices should have encodings such that cases that are similar with respect to case solutions will have index encodings which are close in Hamming distance.

3.2.4 ALN Adaptation

The ALN adaptation procedure is described in detail in [1] and [2]. It is summarized here. Adaptation is fairly simple because it is locally isolated at each processing element. A training set consists of training samples (exemplars) along with the known correct output. These are presented to the ALN in random order. For each presentation, the ALN adapts in response to the presentation. There are two main issues in adaptation: how to adapt an element and when. The how issue is about choosing an appropriate transfer function and the when issue is about deciding when an element is "important" in a computation and consequently if it should be adapted for the current exemplar. Each of these will be discussed in order.

A processing element has four transfer functions to choose from during adaptation. During each exemplar presentation, the network is evaluated on the exemplar so that each element has available its inputs. Also, the correct *network* output[4] is made available to each "important" element. "Important" elements are then adjusted according to its inputs and the desired network output. Adjustment is not done in a way that snaps the element to produce the correct output, but rather, it is edged

[3]However, ALNs are capable of learning to determine when it is important for certain subsets of the input to be close to those in exemplars. This determination is based on other subsets of the input.

[4]Because of ALN monotonicity, this serves as the desired element output for "important" elements.

towards that state. This gradual adaptation is done for the following reasons:

- It is not necessary for every "important" element to actually produce the desired output. It is possible for other parts of the network to "take up the slack" sometimes.

- The initial choice of which elements are "important" is a heuristic one and so could be wrong. Naturally, it is undesirable to snap an element if the basis for change could be wrong.

- There is always a choice of functions to which to move, given inputs and desired output. It is inappropriate to just select one and change to it. Rather, just reinforce the appropriate choices and inhibit the inappropriate ones. After a number of exemplars have been processed, the most appropriate choice should emerge.

To facilitate this gradual movement towards an appropriate function, two bounded counters are maintained at each element during adaptation. The bounds are set arbitrarily with the tradeoff that smaller bounds result in "snappier" behaviour and larger bounds can result in slower adaptation. The current range is 64. One will be called the (1, 0) counter and the other the (0, 1) counter. If the inputs are (0, 0) or (1, 1) then the output is predetermined regardless of the transfer function. In this case, the element adaptation is foregone as pointless. It is when the input is either (1, 0) or (0, 1) that the transfer function is important. The counters respond to the inputs according to the desired output and they then determine the function of the element. If both counters are above the midpoint of their bounded range, then the element function is an OR. If both are below, the function is an AND. If only the (1, 0) counter is above the midpoint, the function is a LEFT and the function is a RIGHT if the (0, 1) counter is the only high one. On a (0, 1) adaptation input, the (0, 1) counter in incremented if the desired output is 1, otherwise it is decremented. This appropriately reinforces both the OR and RIGHT functions in the first case and

inhibits them in the second case. The treatment for the (1, 0) counter and input is symmetric.

The choice of which elements are "important" (should be adapted rather than just left alone) for a given exemplar is a heuristic one. Clearly, if an element is not "important" for an exemplar, then none of its children are. It is also clear that the root element is always "important". The remaining heuristics for determining which child elements of "important" elements to adapt are outlined below. Note that any discussion of left and right is symmetric:

- If one of the inputs of an "important" element is not the same as the desired output then the source of the *other* input is declared "important". The rationale for this is that if the other side is producing the desired output then the adaptation of the current element will encourage the use of the other side and so it should be encouraged by making it "important". If the other side is also not the same as the desired output, then both children will be declared "important" by this rule since at least one of them needs to come around.

- If one input is "enabled" to pass through an element then its source is "important". This occurs when one input is a 0 to an OR element or one input is a 1 to an AND element. In these cases the other input effectively just passes through. Also (clearly) the left and right inputs at LEFT and RIGHT elements respectively are "enabled" to pass.

- The right and left children are declared "important" at LEFT and RIGHT elements respectively. This may seem the most strange of all. Part of the rationale for this rule is that "it can't hurt". Later on, perhaps if the current element changes function, "it might help".

3.3 Why ALNs

ALNs are used here to filter candidate knowledge items based on previous expert responses to similar candidate items. Basically, this is an inductive learning function and many inductive techniques might be well applied here instead. For example, Quinlan's ID3 [24] method might be used as it also builds a classification scheme based on a set of exemplars. Other alternatives might include various other neural net methods. ALNs, however, are used for the following reasons:

- All that is needed is a yes-no to the question of whether a candidate new KB item should be put forth or not. The boolean output of an ALN is adequate for this while other inductive methods may have more elaborate results which are not needed.

- ALNs seem suitable for inductive reasoning within a KBS but have not yet been used as such. ALNs have been used for induction or empirical generalization in other applications including the following:

  - database mining – the discovery of functional dependencies within a relational database;
  - grading beef based on ultrasound images;
  - discriminating subatomic particles produced by a high-energy accelerator;
  - assisting in the control of sophisticated suspension systems for rough-terrain vehicles;
  - assisting in the control of walking prostheses; and
  - measuring tarsands composition from spectral data.

  The above applications are specialized. The use of ALNs here will demonstrate that they can be used in a more general, domain independent way for induction within a KBS.

- Since ALNs are subsymbolic, this will demonstrate the effectiveness and practicality of using symbolic and subsymbolic methods together in a reasoning system.

- The auxiliary KB needed can be maintained in the short term as the actual network and in the long term as a training set made up from all previous suggestion response pairs. The second form is simple, intuitive, and easy to manipulate.

Chapter 4

An Implementation of THINK

This chapter describes, in detail, a THINK implementation. The purpose is twofold:

1. To show by example how the THINK framework can be applied.

2. To describe the experimental system used to test the thesis of this research.

The description provides sufficient detail to convey all the important fundamentals of THINK. Only those details which either are directly part of THINK or are needed as glue to put things in context are described here. For example, details for the user interface, for enhancing efficiency, and KB storage-retrieval are omitted here for the sake of brevity and clarity.

First, a description of the performance system is given along with examples. Then, the KA subsystem is described. Finally, an example of THINK in operation is given with annotations highlighting the above details.

4.1 The Performance System

The KB of the performance system here is based on causal associational links. Here, the $\rightarrow$ connective means "causes". The links have the form $C_1, \ldots, C_n \rightarrow E$, where the C_is and the Es are literals. The C_is are taken as a conjunction. An example knowledge item would be $[wolf(X), full\_moon] \rightarrow howls(X)$.

As well as the causal associations above there is a class hierarchy of predicates[1]. An example would be $timberwolf \rightarrow_{isa} wolf$. Thus, $wolf(a)$ can be inferred from $timberwolf(a)$ but not *causally*. The hierarchy is not strict in the sense that a predicate name may be a direct descendant of more than one other predicate name. That is, the hierarchy forms a partially ordered set rather than a tree. An example from automotive diagnosis would be *loose_cylinder_valve*. This could come under *cylinder_problem* and *mechanical_problem*. The two classes are neither disjoint, nor does one contain the other.

After providing the system with a set of observations (usually just literals), the system can then be asked to explain a literal item. The system then uses a straightforward backchaining mechanism to establish a cause for the literal. The *immediate* cause is reported. If the next-immediate level cause is desired it can be found by having the system explain the immediate causes. The system reports immediate causes, because the semantics of the causal links, here, preclude transitivity.

There is an important notional distinction between causal links and isa links which is reflected in the operation of the system. Consider the examples above. If $timberwolf(a)$ is given as an observation and an explanation for $wolf(a)$ is requested, it may not make sense to give $timberwolf(a)$ as a causal explanation[2]. Backchaining inference proceeds along both isa and causal links, but only steps across causal links are considered as causal explanation steps. An example is when $full\_moon$ is also given as an observation and an explanation is requested for $howls(a)$. The answer given is the pair $[full\_moon, wolf(a)]$. There is no further explanation for $wolf(a)$ even though it was actually inferred through an isa link. Also, if $howls \rightarrow_{isa} noisy$ was an isa link and an explanation for $noisy(a)$ was requested, then the same answer would be given - $howls(a)$ is not part of the *cause* for $noisy(a)$.

[1]The use of a hierarchy illustrates that there can be other knowledge in the performance system other than associational. However, only associational knowledge will be the target of KA.

[2]This depends entirely on the intended meaning of "causes" within the system. In any case, there needs to be a distinction between "causes" and "is a" at least for purposes here.

4.1.1 Facts and Observations

The KB consists of *facts*, *observations*, and hierarchal links. The only distinction between facts and observations is one of permanence. Facts are permanent KB items and observations are temporary applying to only one or a few problem instances. Other than that, they have equal standing in the KB and are used without preference for one or the other. It is allowable, although probably not appropriate, to have a causal link as an observation and, likewise, it is allowable to have a literal item as a (permanent) fact. Typically, observations are used in defining a problem instance (e.g., symptoms to diagnose).

4.1.2 Restrictions on Variables

In order to ease reasoning tasks (particularly redundancy and consistency checks), all explanation requests must be for ground (containing no variables) literals. Also, to ensure the same holds for all subsequent subtasks, all links (of the form $C \rightarrow E$) must have the property that all variables which appear in C also appear in E (although the reverse need not hold). Thus, links such as $bird(X) \rightarrow fly(X)$ and $friendly(X) \rightarrow likes(Y, X)$ are allowed but a link such as $likes(Y, X) \rightarrow happy(X)$ is not allowed.

4.2 The KA Subsystem

This section describes the KA subsystem in detail. First, some definitions necessary for understanding are provided. Then, the candidate generation procedure is described. To structure this description, it is given in two parts. Each of these parts draw on the techniques described in detail in Chapter 3. First, the method of identifying candidates is described, then an adaptation of that method which orders the presentation of candidates by quality is described. Finally, the actions taken on the expert's response is described.

4.2.1 Partial Causal Links

The performance system allows links of the form $C_1, \ldots, C_n \rightarrow E$. The method for acquiring these conjunctive cause links is described in detail later. In brief, it involves incrementally building the conjunction by adding conjuncts one at a time. To support this, an intermediate kind of link is needed. This will be the partial causal link and will have the form, $C_1, \ldots, C_n \rightarrow_p E$. In short, this link means that the conjunction forms *part* of a cause for E. Formal meanings of both kinds of links together with meanings of *roughly* negated links are given below.

$\mathbf{C \rightarrow E}$: C is a minimal set of conditions to cause E. That is, C causes E and if anything were removed from C the result would not cause E.
E.G., $[full\_moon, wolf(a)] \rightarrow howls(a)$.

$\mathbf{C \rightarrow_p E}$: C is some proper subset of some C' which is a minimal cause for E. That is, all of the components of C could, together, play a part in some cause for E.
E.G., $[full\_moon] \rightarrow_p howls(a)$.

$\mathbf{not(C \rightarrow E)}$: C does not form *any* subset of some cause for E.
E.G., $not([full\_moon, grey(a)] \rightarrow howls(a)$,
and $not([full\_moon, wolf(a), grey(a)] \rightarrow howls(a)$.

$\mathbf{not(C \rightarrow_p E)}$: This has the same meaning as $not(C \rightarrow E)$.

Notice that "not" here is not the same as logical negation. However, when "not" is applied to an atomic proposition, its meaning is negation. Recall that causal links are not considered transitive; they can be thought of as meaning "directly causes" instead of just "causes". Hierarchal links, however, *are* transitive.

4.2.2 Abducing Candidate Missing KB Items

This section describes the raw (thoughtless) generation of candidate KB items. Later, an adaptation is described to "add thoughtfulness" by ordering generation. To generate candidate KB items, an abduction based search is used. It is actually an extension

of the backchaining search used to search for an explanation for some literal. As such, the two searches are combined into one.

The best way to describe the combined search is as a cost bounded iterative deepening search. The nodes in the search space are literals to be explained[3]. Nodes may be expanded in the following ways with preference in the order given. Note that any hypothesis must be both consistent and non-redundant. Also, circularity is checked and pruned.

1. If the literal is explicitly in the KB, then it is branched to a success leaf. In this case there are no other branches needed.

2. For a node, G, if there is a link, $C \rightarrow G$, then it is branched to a C node.

3. For a node, G, just hypothesize that G is true. To exercise this branch would mean asking the user if G was true. The resulting node is a success leaf. This kind of branch is not applicable to the root of the search because it would not result in any explanation for the root literal.

4. For a node, G, if there is some partial link, $C1 \rightarrow_p G$, and there is some other explicitly known literal, C, then hypothesize that $C' \wedge C1 \rightarrow G$ is true where C' is the result of arbitrarily filling in variables of C with arguments of G[4]. The resulting node is $C' \wedge C1$. This way partial links are pursued and built upon.

5. For a node, G, if there is some partial link, $C1 \rightarrow_p G$, and there is some other literal, C, appearing in any link or partial link, then hypothesize $C' \wedge C1 \rightarrow G$ where C' is the result of the same kind of variable filling as above. The resulting new node is $C' \wedge C1$. This also builds on current partial links but in a more "desperate" manner.

6. For a node, G, if there is some other explicitly known literal, C, then hypothesize that $C' \rightarrow G$ is true where C' is the result of variable filling as above. The

[3]Some nodes can be conjunctions which are expanded as AND nodes.
[4]Each different way of filling in variables results in a separate branch.

resulting new node is C'. This can result in a new link or partial link in the KB.

7. For a node, G, if there is some other literal, $C1$, appearing anywhere in any link or partial link, then hypothesize $C' \rightarrow G$ where C' is the result of the same kind of variable filling as above. The resulting new node is C'.

To "exercise" a hypothesis-based branch means to ask the user if the hypothesis is true. The cost of a branch for the first 3 types above is 0. The cost for the other, hypothesis-based, branches is 1. The cost of a node, n, is

$$\min_{n_s \in successors(n)} cost(n_s) + branch\_cost(n, n_s).$$

The cost of a success leaf is 0. The cost of an AND node is the corresponding max function. Note that branches from an AND node have no cost. Given a KB of $\{A \rightarrow B, B \rightarrow C, D\}$ and a goal of explaining C, the search space expanded for costs bounds of 0, 1, and 2 are shown in figures 4.1, 4.2, and 4.3, respectively. Success leaves (explicit KB lookups) are in boxes; interior nodes are shown as large letters; zero cost branches are shown as solid lines; and cost nodes are shown as dashed lines labeled with the corresponding hypotheses.The preferences from the above ordering are reflected in the left to right ordering of the branches in the figures; a left to right pre-order traversal represents the actual search.

Notice that a 0 cost bounded search just corresponds to using the KB as it is (no KA at all) to find an explanation. This allows a simple yet close integration of KA into the reasoning mechanism. During KA, the search is "forward looking". That is, before actually exercising a hypothesis branch (asking the user), the space below that branch is checked to ensure that a success is possible with the current bound. For example, with a bound of 1 in figure 4.2, the branch labeled with $A \rightarrow C?$ would not be exercised for this reason, but it can be exercised with a bound of 2 (figure 4.3). After asking about a hypothesis, if the response is positive, then the cost for the associated branch is discounted (it is not hypothesis-based anymore) and the space below it is expanded accordingly. For example, in figure 4.3, if a positive response for

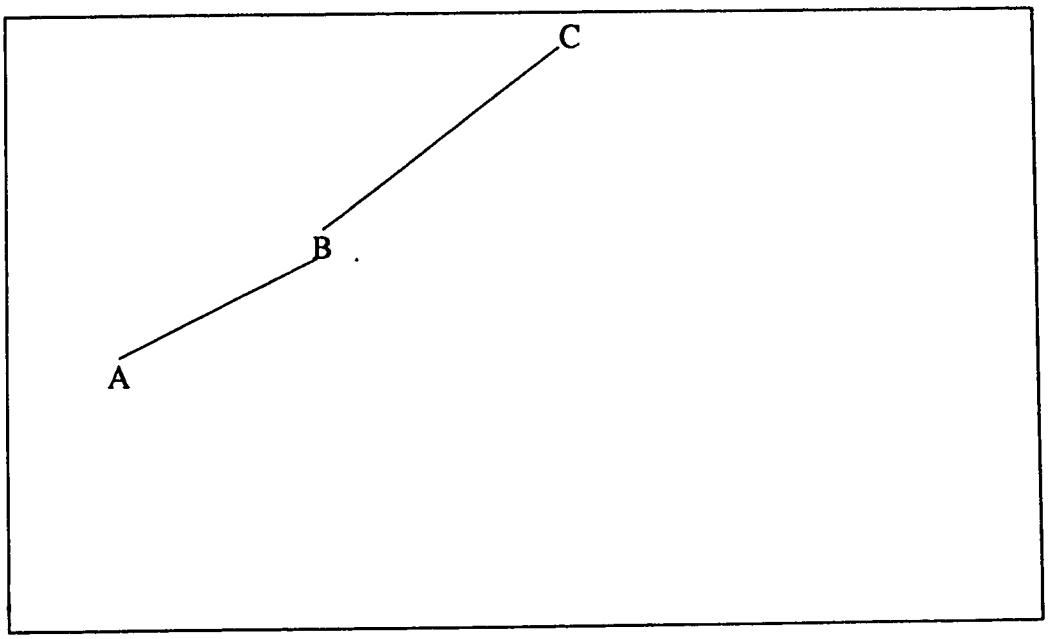

Figure 4.1: A Depth 0 Search Space

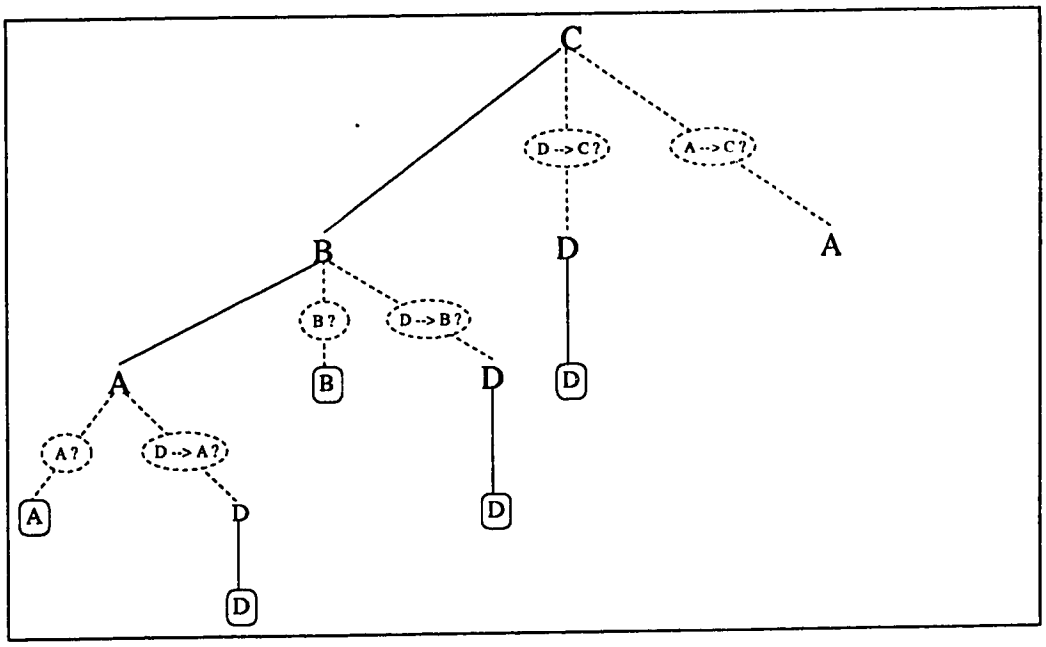

Figure 4.2: A Depth 1 Search Space

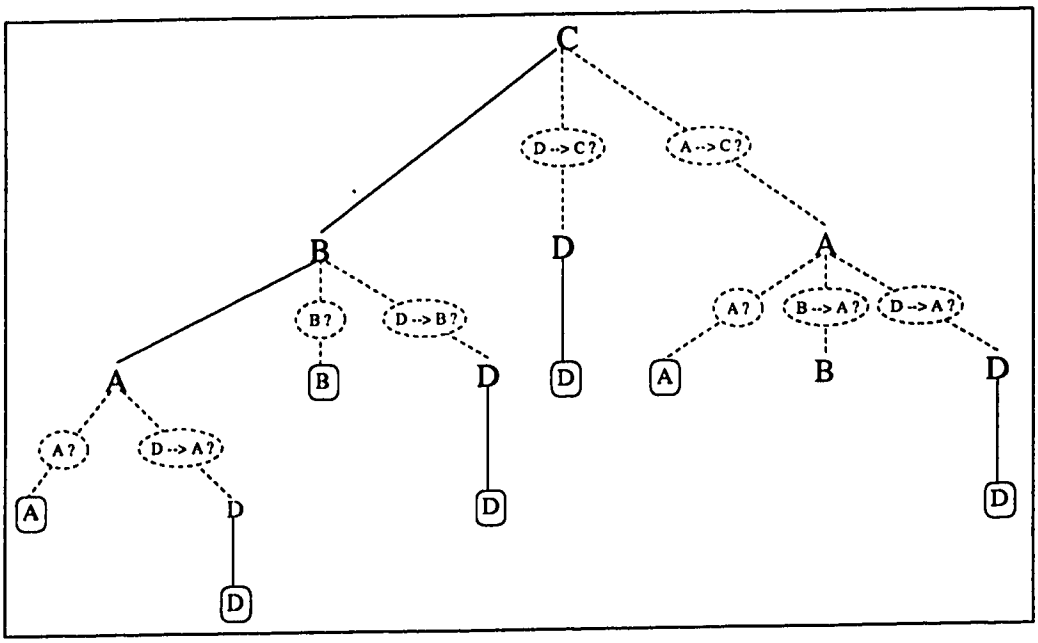

Figure 4.3: A Depth 2 Search Space

the C—A link is received, then the B node at the end of the C—A—B path can be expanded. Nodes are automatically pruned when they become inconsistent.

Result Strengthening

As pointed out in section 3.1.1, an abductive hypothesis can be more generally applicable than is necessary for the current problem. In a KB, the stronger the knowledge, the better. It is therefore useful to pursue an expert's response (positive or negative) to a candidate KB item by proposing slightly stronger ones. For example, if $canary(a) \rightarrow fly(a)$ has just been acquired by the system, then why not propose $bird(a) \rightarrow fly(a)$?

The system here uses the hierarchal information to pursue responses in this way. The ways of strengthening a new KB item depend on its form and are described below. The reader should assume the obvious intuitive hierarchy for the given examples.

P($\vec{\text{args}}$): Find a P' such that $P' \rightarrow_{isa} P$ and strengthen to $P'(\vec{args})$. E.G., $bird(a)$ strengthens to $canary(a)$.

not(P(arg̃s)): Find a P' such that $P \rightarrow_{isa} P'$ and strengthen to $not(P'(\vec{args}))$. E.G., $not(canary(a))$ strengthens to $not(bird(a))$.

C(arg̃s) → ⟨X⟩: Find a C' such that $C \rightarrow_{isa} C'$ and strengthen to $C'(\vec{args}) \rightarrow \langle X \rangle$. E.G., $canary(a) \rightarrow fly(a)$ strengthens to $bird(a) \rightarrow fly(a)$.

not(C(arg̃s)) → ⟨X⟩: Find a C' such that $C' \rightarrow_{isa} C$ and then strengthen it to $not(C'(\vec{args})) \rightarrow \langle X \rangle$. E.G., $not(fast(a)) \rightarrow late(a)$ can be strengthened to $not(very\_fast(a)) \rightarrow late(a)$.

⟨X⟩ → E(arg̃s): Find an E' such that $E' \rightarrow_{isa} E$ and strengthen to $\langle X \rangle \rightarrow E'$. E.G., $jet(a) \rightarrow fly(a)$ strengthens to $jet(a) \rightarrow fly\_high(a)$.

⟨X⟩ → not(E(arg̃s)): Find an E' such that $E \rightarrow_{isa} E'$ and then strengthen it to $\langle X \rangle \rightarrow E'$. E.G., $summer \rightarrow not(cold\_weather)$ can be strengthened to $summer \rightarrow not(nasty\_weather)$. (Cold weather is a kind of nasty weather.)

not(C(arg̃s) → ⟨X⟩): Find a C' such that $C' \rightarrow_{isa} C$ and then strengthen it to $not(C'(\vec{args}) \rightarrow \langle X \rangle)$. E.G., $not(bird(a) \rightarrow big(a))$ can be strengthened to $not(condor(a) \rightarrow big(a))$.

not(not(C(arg̃s)) → ⟨X⟩): Find a C' such that $C \rightarrow_{isa} C'$ and then strengthen it to $not(not(C'(\vec{args})) \rightarrow \langle X \rangle)$. E.G., $not(not(very\_slow(a)) \rightarrow on\_time(a))$ strengthens to $not(not(slow(a)) \rightarrow on\_time(a))$.

not(⟨X⟩ → E(arg̃s)): Find an E' such that $E \rightarrow_{isa} E'$ and then strengthen it to $not(\langle X \rangle \rightarrow E')$. E.G., $not(summer \rightarrow sunny)$ can then be strengthened to $not(summer \rightarrow pleasant)$. (Sunny weather is a kind of pleasant weather.)

not(⟨X⟩ → not(E(arg̃s))): Find an E' such that $E' \rightarrow_{isa} E$ and strengthen to $not(\langle X \rangle \rightarrow E')$. E.G., $not(fast(a) \rightarrow not(late(a)))$ can be strengthened to $not(fast(a) \rightarrow not(very\_late(a)))$.

Strengthening is done recursively until either there are no more consistent strengthening steps or a negative response is received. In the first case, recursion ends; in

the second case recursion continues on the negative response. For example, after successfully strengthening $canary(a) \rightarrow fly(a)$ to $bird(a) \rightarrow fly(a)$ and then an unsuccessful attempt to strengthen to $animal(a) \rightarrow fly(a)$, the recursion ends. A final strengthening step is to propose a universal generalization which can strengthen any form. Thus, $bird(a) \rightarrow fly(a)$ will be strengthened to $bird(X) \rightarrow fly(X)$ (for any X) and proposed to the user.

Except for the final universal generalization step, the above strengthening procedure is akin to Mitchell's version space approach [18]. All the possible ground rules along with lines of strengthening define a partial ordering much like the partial ordering of concept descriptions used by Mitchell. There are some fundamental differences in the model, though, which are described below:

1. In the version space approach, there is a distinction between concept *descriptions* and *instances* of a concept. Here, there is no distinction. This is a relaxation making things easier in this implementation.

2. In the version space approach it is assumed that examples of concept instances are provided (at random) from some external source not concerned with the generalization problem. Here, the approach is for the system to select more and more general examples based on a starting point and the lines of generalization available. This is an advantage for this system because it is always acquiring samples which are useful in identifying maximally strengthened rules (target concepts). It does not have to wait for useful samples to be generated from the environment.

3. During generalization in version spaces, it is assumed that there is a single target concept description and examples provided are positive or negative instances of that concept. Here, there are several valid rules and the KB (along with negative information) provide positive and negative examples of more than one maximally strengthened rule (concept). This serves as a hindrance to the system here with respect to taking advantage of a version space approach.

The advantage of the first two points is taken as the system freely generates strengthened rules and presents them to the user. Because there is no distinction between description and instance, the user sees each presentation as an example candidate rule. The third point above demands that the system always present generalizations to the user. As an example, consider that the system knows

$$\{canary(X) \rightarrow fly(X), bald\_eagle(X) \rightarrow fly(X)\}$$

and

$$\{canary \rightarrow_{isa} songbird, bald\_eagle \rightarrow_{isa} eagle, songbird \rightarrow_{isa} bird, eagle \rightarrow_{isa} bird\}$$

With this, the version space algorithm would immediately generalize to the rule, $bird(X) \rightarrow fly(X)$ missing at least one intermediate generalization (strengthening) step. Furthermore, it would not even have to confirm the generalization with the user because there is only one target concept description. In the system here, this would not be justified because the two rules could be independent of each other. However, the above situation does suggest that a larger step in strengthening to generate a *candidate* would make sense. Rather than first asking, say, if songbirds fly or eagles fly, just ask straight away if birds fly. If this is not accepted, the weaker (less general) versions could still be tried later. The system here does not yet take advantage of this approach but it would likely be a good enhancement. Such an enhancement might also use ALN support to justify "leaps" through the partial order. Another idea would be to use version space and ALN justification to immediately strengthen a candidate generated from the search phase and present this initially to the user.

4.2.3 Candidate Ordering with ALNs

ALNs are used to order abduced candidate items before presenting them to the user. So not to compromise completeness, this ordering does not *exclude* items but rather just provides preferences for some over others. ALNs will only apply to ground literals and ground positive links. This section will describe how information is encoded into

an ALN input and how the results of ALN computations are used to order candidate items.

Input Encoding

There are two main parts of an ALN input: an encoding of the candidate KB item, and an encoding partly representing the context. As will be seen later by example, hierarchal information is very important for generalization and this forms part of both encodings. First, the encoding of context is described, and then that for the actual candidate is described.

Context Encoding. It is difficult to come up with a reasonable representation for an arbitrary scenario which could involve any number of different objects and relations (predicates) holding between them. Consequently, a limited context description is used. Any candidate KB item will involve a limited number of objects (arguments of predicates) which will be considered to be of prime importance. The limited context description is then just a list of explicitly known literals which involve any of the "important" arguments appearing in the candidate KB item.

For any candidate item of a given form, there is a particular maximum number of "important" objects. For example, a candidate of the form, $P(X) \rightarrow Q(Y,Z)$ has a maximum of three important items. Let $\mathbf{D}$ represent a qualified wild-card object which is just any object *other* than one of the "important" ones. Using $\mathbf{D}$ and the important objects, there is a limited number of ways of filling in the arguments of any predicate. It is therefore possible to assign a particular ALN input value (bit) to each of these possibilities for each predicate in the system. For example, if there are two important objects, A and B, and there is a binary predicate, P, then there would be an input bit assigned for each of the following tuple forms of P[5]:

$$(A,A),(A,B),(A,\mathbf{D}),(B,A),(B,B),(B,\mathbf{D}),(\mathbf{D},A),(\mathbf{D},B),(\mathbf{D},\mathbf{D}).$$

To encode the context, each of these bits are set if the predicate holds for the assigned

[5]There is also the same assignment for $not P$.

tuple. Also, corresponding bits are set for all predicates above the considered predicate. For example if *tweety* is an important object and $canary(tweety)$ is explicitly known, then the bits for $bird(tweety)$ and $animal(tweety)$ are also set.

It is important to understand that the input bit assignments are done by placewise reference to the arguments appearing in the candidate KB item. For example, A, above, would mean "the the object appearing first in the KB item" and B would be the second. With a different candidate KB item of the same form (same arities but possibly different predicates), the input assignments would be the same but would be set according to the actual arguments in the candidate KB item. Thus, the context input bits are set (or not) depending on substitutions of arguments from the candidate KB item to the various predicates of the system.

Candidate KB Item Encoding. Each possible candidate will involve either one or two predicates: one if the candidate is a literal, or two if it is a link. To encode a link suggestion it is just a matter of identifying the predicates involved and which is at which end of the candidate link. To do this, two sets of bits are assigned. One set identifies one end of the link and the other identifies the other. Each set has one bit assigned for each predicate in the system. The appropriate bit is set for the involved predicate in the candidate item. Also, bits are set for all predicates above the involve the involved predicate. For example to encode $bird(a) \rightarrow fly(a)$ as a candidate, there is one set to encode *bird* and one set to encode fly. For the first, the bits associated with *bird*, and *animal* are set; for the second, the bits associated with fly and *move* are set.

There is also a set of bits indicating pairwise identity of arguments in the candidate item. This is important to distinguish two candidates like $bird(a) \rightarrow fly(a)$ and $bird(a) \rightarrow fly(b)$. While the first might make sense, the second probably does not. In this set there is a bit for each 2-combination of arguments and it is set if the combination is a pair of identical arguments. For example, for a suggestion of the form $P(X) \rightarrow Q(Y, Z)$, there are three bits - one for each of the following identities: $X = Y$, $X = Z$, and $Y = Z$.

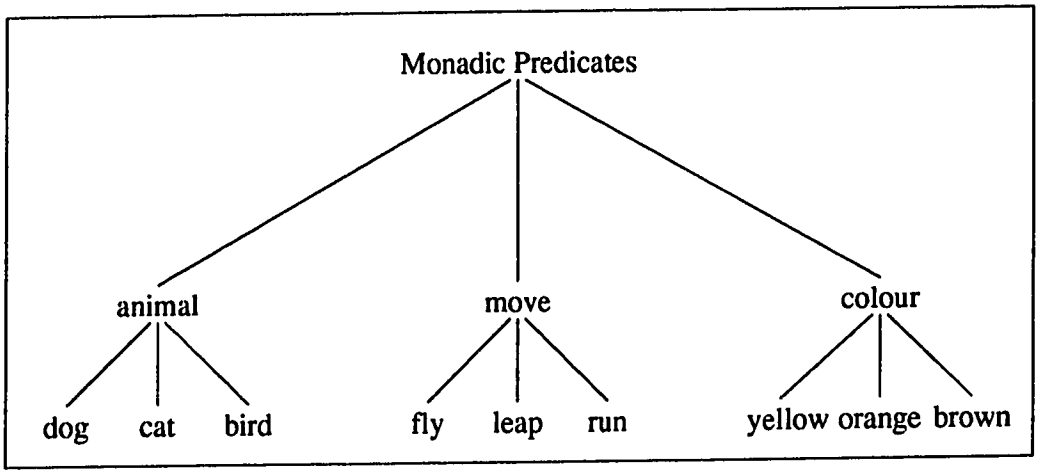

Figure 4.4: Example hierarchy

Multiple Specialized ALN Types. The form of the candidate KB item has a lot to do with the structure of the input encoding. For this reason, there are separate ALN types specialized for each form. The important aspects of the form for this are the arity of the involved predicates and whether the candidate is a literal item or a link item. The implemented system allows four arities (0 - 3). There are then 4 forms for literal candidates and $4 \times 4 = 16$ forms for links. Thus, there are 20 different specialized ALN types.

An Encoding Example. The foregoing should be made more clear with some simple examples. Consider a KB with a hierarchy as depicted in figure 4.4 and the following facts and observations:

$$\{bird(t) \rightarrow fly(t), not(yellow(t) \rightarrow fly(t)), bird(t), yellow(t), dog(r), brown(r)\}$$

Encodings for $bird(t) \rightarrow fly(t)$, $yellow(t) \rightarrow fly(t)$, $dog(r) \rightarrow run(r)$, and for $brown(r) \rightarrow run(r)$ are given in columns 1, 2, 3, and 4, respectively in table 4.1.

Now, suppose that the first two encodings are from past experience and were classified as "good" and "poor", respectively. The Hamming distance between the third and first columns is 12 and the distance between the third and second columns is 14. The ALN would therefore be expected to classify the third pattern the same

Encodings for form $P(X) \rightarrow Q(Y)$

Column 1	2	3	4	Bit Assignment
				Context Bits
0	0	1	1	$dog(X)$
0	0	0	0	$not(dog(X))$
0	0	1	1	$dog(Y)$
	0	0	0	$not(dog(Y))$
	1	0	0	$bird(X)$
0	0	0	0	$not(bird(X))$
1	1	0	0	$bird(Y)$
0	0	0	0	$not(bird(Y))$
1	1	1	1	$animal(X)$
0	0	0	0	$not(animal(X))$
1	1	1	1	$animal(Y)$
0	0	0	0	$not(animal(Y))$
0	0	0	0	$run(X)$
0	0	0	0	$not(run(X))$
0	0	0	0	$run(Y)$
0	0	0	0	$not(run(Y))$
0	0	0	0	$fly(X)$
0	0	0	0	$not(fly(X))$
0	0	0	0	$fly(Y)$
0	0	0	0	$not(fly(Y))$
0	0	0	0	$move(X)$
0	0	0	0	$not(move(X))$
0	0	0	0	$move(Y)$
0	0	0	0	$not(move(Y))$
0	0	1	1	$brown(X)$
0	0	0	0	$not(brown(X))$
0	0	1	1	$brown(Y)$
0	0	0	0	$not(brown(Y))$
1	1	0	0	$yellow(X)$
0	0	0	0	$not(yellow(X))$
1	1	0	0	$yellow(Y)$
0	0	0	0	$not(yellow(Y))$
1	1	1	1	$colour(X)$
0	0	0	0	$not(colour(X))$
1	1	1	1	$colour(Y)$
0	0	0	0	$not(colour(Y))$

Column 1	2	3	4	Bit Assignment
				Candidate Encoding
				P bits
0	0	1	0	*dog*
0	0	0	0	*not dog*
1	0	0	0	*bird*
0	0	0	0	*not bird*
1	0	1	0	*animal*
0	0	0	0	*not animal*
0	0	0	0	*run*
0	0	0	0	*not run*
0	0	0	0	*fly*
0	0	0	0	*not fly*
0	0	0	0	*move*
0	0	0	0	*not move*
0	0	0	1	*brown*
0	0	0	0	*not brown*
0	1	0	0	*yellow*
0	0	0	0	*not yellow*
0	1	0	1	*colour*
0	0	0	0	*not colour*
				Q bits
0	0	0	0	*dog*
0	0	0	0	*not dog*
0	0	0	0	*bird*
0	0	0	0	*not bird*
0	0	0	0	*animal*
0	0	0	0	*not animal*
0	0	1	1	*run*
0	0	0	0	*not run*
1	1	0	0	*fly*
0	0	0	0	*not fly*
1	1	1	1	*move*
0	0	0	0	*not move*
0	0	0	0	*brown*
0	0	0	0	*not brown*
0	0	0	0	*yellow*
0	0	0	0	*not yellow*
0	0	0	0	*colour*
0	0	0	0	*not colour*
				identity bit
1	1	1	1	$X = Y$

Table 4.1: Example Encodings

as the first rather than the second. Likewise, because the Hamming distances from the fourth column to the first and second columns are 14 and 12, respectively, the ALN would be expected to classify the fourth input pattern like the second. Thus, (based on the two supposed exemplars) $dog(r) \rightarrow run(r)$ will likely be classified as positive (good) and $brown(r) \rightarrow run(r)$ will likely be classified as negative (poor) and so preference will be given to $dog(r) \rightarrow run(r)$. This makes sense to people. What is captured in the example exemplars can be expressed as: "kinds of animal" attributes can cause "kinds of move" attributes; and "kinds of colour" attributes do not likely cause "kinds of move" attributes.

This example shows the importance of the hierarchal knowledge to enable the ALNs to generalize on the exemplar (training) information. Another principle which arises is that the hierarchy must be very well considered to enable this. Classes of predicates should be grouped so that generalizations as in the above example can emerge. That is, classes should contain predicates which tend to cause predicates in other particular classes (given appropriate identities among the arguments).

The ALN Role in Search

Section 4.2.2 describes an iterative deepening search method for abducing candidate KB items. The examples in the previous section illustrate how ALNs can favour candidates which are "similar" to positive training samples and disfavour candidates which are "similar" to negative training samples. It is a simple matter to use ALN results to adjust the cost associated with each hypothesis branch in the search space. For example the cost of a hypothesis branch which the ALN classifies as positive can be 1, and the cost of a hypothesis branch which the ALN classifies as negative can be 2. The cost of non-hypothesis branches can remain as 0. The iterative deepening search can then be conducted as described except with these branch costs.

This simple adaptation is very close to what is implemented here, but there are a couple of enhancements worth noting. Rather than using just one ALN to com-

pute the added cost for a branch, this implementation uses more[6]. Because of the randomness in the initial structure of the ALNs and the random order of presenting exemplars during adaptation, it is likely that a number of them will not always agree on classifications. It is reasonable to place more confidence in a classification when there is greater agreement. Clearly, a finer granularity of preference can be gained by using multiple ALNs: Prefer most those candidates which all ALNs classify as positive; next, prefer those which all but one classify positive; and so on. To apply this to the cost bounded search, the cost of a hypothesis branch becomes $1 + num\_neg$ where *num_neg* is the number of ALNs which classify the branch as negative (poor).

Informal experiments have indicated that it is best to complete searching at a bound defined by the *number* of hypotheses branches before allowing a greater number of these branches in a path. Therefore a nested iteration is used. The outer loop varies the number of allowable hypothesis branches and the inner loop varies the total cost from 1 to the maximum possible given the inner loop bound. The depth bound is therefore a two-part one: The number of hypothesis branches must be below the maximum; AND the total cost must also be in bound.

When a hypothesis branch is actually exercised and receives a positive user response, then that branch becomes a non-hypothesis branch and has a cost of 0. The search space below then automatically expands to the current bounds and the search continues down.

The ALN Role in Strengthening

Result strengthening, as described in section 4.2.2, is not part of the search, proper. It is an aside – a tangent. It makes no sense, therefore, to apply the cost bound in force at the time during strengthening. Rather a static bound is used: A strengthened hypothesis is put forth if one third or more of the ALNs classify the the strengthened hypothesis as positive.

[6]Specifically, 7 are used here.

4.3 Acting on Expert Responses

The previous sections described how candidates are generated. This section describes the allowed expert responses, the intended interpretation of those responses, and the action taken by the system. Recall that expert responses are two-part. The first part is used for adapting the KA subsystem; in this implementation it is used to add training exemplars for the ALNs. The second part is used for integrating some form of the candidate into the primary KB. Each part is meant to be treated separately and they will be described separately below.

4.3.1 Acquisition of Training Samples

Every time a hypothesis branch is exercised, it is an opportunity to gain a new training sample for the ALNs. The expert responds with either "good" or "poor" as his classification of the quality of the candidate. An encoding for the context and candidate will have already been constructed.

Acquisition During Strengthening

ALNs seem to have a hard time learning k-of-n functions (or even 1-of-n functions). This thwarts learning during strengthening. As an example, consider the above idea that *kinds* of animal attributes will cause kinds of move attributes, but the animal attribute by itself is not enough for a particular mobile abilities. During strengthening, the strengthened candidate, $animal(a) \rightarrow move_i(x)$[7] will be asked and be responded to negatively and as a "poor" candidate. Thus, every time a positive example is acquired, a very similar negative example is also acquired which actually goes *against* the desired concept. In this example, a negative sample is acquired where the *animal* attribute is set in the P bits; and a particular $move_i$ and *move* attributes are set in the Q bits. But, it is the *animal* attribute that is actually positively important in the concept. The desired concept is that particular *kinds* of animals cause cause

[7] $move_i$ is some kind of *move* attribute

certain kinds of mobile abilities (i.e., 1-of-n *animals* causes 1-of-m *moves*). Rather than learning this function (when there are few training samples), the ALN will tend to learn that the *animal* attribute (in general) is poor for this because of the negative examples received during strengthening. This is definitely NOT what is wanted.

A simple solution is to not use negative ("poor") training samples from strengthening phase. It would still be possible to get undesirable negative training samples, as above, from normal search (unstrengthened) hypotheses. Then, these would still be included. Hopefully, this would not happen too often and it certainly would not happen *as* often. This solution still leaves some deficiency in the system.

A better solution would be based on capturing the desired 1-of-n function components explicitly in some assigned input bits. There could be a particular bit assigned for each attribute in each part of the input (each set of input bits) indicating if there are more discriminating bits set for this attribute in the same set. This bit will be a kind of helper bit for 1-of-n functions. It would be set whenever the corresponding 1-of-n condition holds. EG: for $dog(x)$, the *dog*, *animal*, and 1-of-n-(animal) bits would be set, but for $animal(x)$, only the *animal* bit would be set. This solution would require about double the number of ALN inputs.

The implementation here only has the first, simple, solution to this problem incorporated. The second solution is certainly worth further investigation.

4.3.2 Incorporating the Candidate Item

Here, a formal specification of the allowed second-part responses is given. The interpretations and consequent actions for responses to the three main candidate forms ($C \rightarrow E$, $not(C \rightarrow E)$, and literal, P) are given in tables 4.2, 4.3, and 4.4.

---

[8]The "unknowns" list has the same permanence as observations.

[9]The justification for this comes out of the procedure for generating conjunctive candidate causes (discussed in section 4.2.2). $[C_1, \ldots, C_n] \rightarrow E$ is only generated from an existing $[C_2, \ldots, C_n] \rightarrow_p E$ in the KB.

RESPONSE	INTERPRETATION	ACTION
unknown	The expert does not know.	$C \rightarrow E$ is added to a list of unknowns[8] so that it is not asked again. No changes are made to the KB, proper.
never	Neither $C \rightarrow E$ nor $C \rightarrow_p E$ is ever true.	$not(C \rightarrow E)$ and $not(C \rightarrow_p E)$ are both added to the facts. If C is a conjunction with first conjunct, C_1, then $not(C_1 \rightarrow E)$ is also added to the facts[9].
no	Neither $C \rightarrow E$ nor $C \rightarrow_p E$ is true for the current problem instance but may be true in others.	$not(C \rightarrow E)$ and $not(C \rightarrow_p E)$ are both added to the observations. If C is a conjunction with first conjunct, C_1, then $not(C_1 \rightarrow E)$ is also added to the observations.
always	$C \rightarrow E$ is always true.	$C \rightarrow E$ is added to the facts.
yes	$C \rightarrow E$ is not always true but is true for the current problem instance.	$C \rightarrow E$ is added to the observations.
always(partly)	$C \rightarrow_p E$ is always true.	$C \rightarrow_p E$ is added to the facts.
partly	$C \rightarrow_p E$ is true for the current problem instance, but is not always true.	$C \rightarrow_p E$ is added to the observations.

Table 4.2: Interpretations of responses to candidates of the form, $C \rightarrow E$

RESPONSE	INTERPRETATION	ACTION
unknown	The expert does not know.	$not(C \rightarrow E)$ is added to the list of unknowns.
never	$C \rightarrow E$ is always true.	$C \rightarrow E$ is added to the facts.
no	$C \rightarrow E$ is true for the current problem instance but is not always true.	$C \rightarrow E$ is added to the observations.
always	Neither $C \rightarrow E$ nor $C \rightarrow_p E$ is ever true.	$not(C \rightarrow E)$ and $not(C \rightarrow_p E)$ are both added to the facts. If C is a conjunction with first conjunct, C_1, then $not(C_1 \rightarrow E)$ is also added to the facts.
yes	Neither $C \rightarrow E$ nor $C \rightarrow_p E$ is true for the current problem instance but may be true in others.	$not(C \rightarrow E)$ and $not(C \rightarrow_p E)$ are both added to the observations. If C is a conjunction with first conjunct, C_1, then $not(C_1 \rightarrow E)$ is also added to the observations.

Table 4.3: Interpretations of responses to candidates of the form, $not(C \rightarrow E)$

RESPONSE	INTERPRETATION	ACTION
unknown	The expert does not know.	P is added to the list of unknowns.
never	P is never true.	$not(P)$ is added to the facts.
no	P is not true for the current problem instance but may be true for others.	$not(P)$ is added to the observations.
always	P is always true.	P is added to the facts.
yes	P is not always true but is true for the current problem instance.	P is added to the observations.

Table 4.4: Interpretations of responses to candidates of the form, P, for literal, P.

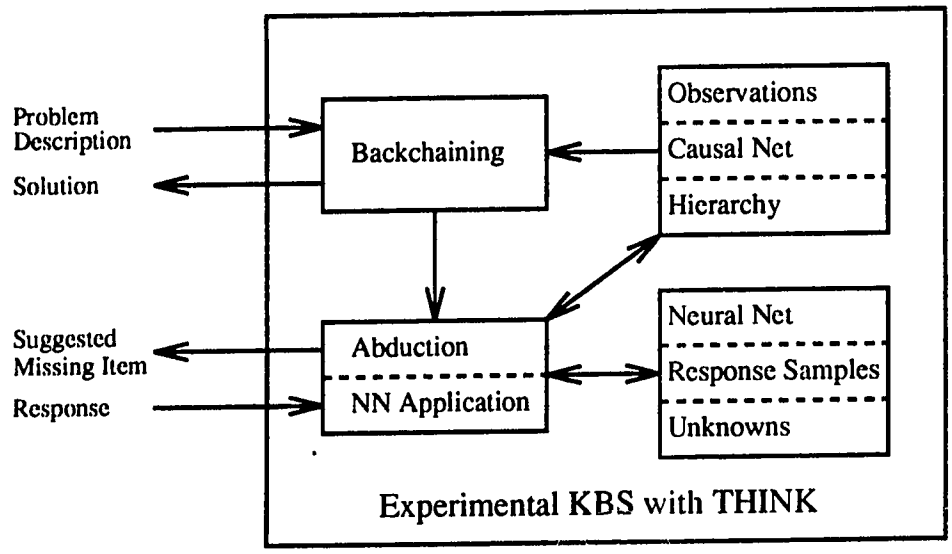

Figure 4.5: View of Experimental KBS with THINK

4.4 Summary

This implementation and how it fits into the THINK framework is summarized in figure 4.5. This diagram parallels the general figure 2.1. The only caveat with the diagram is that the distinction between the backchaining of the performance system inference engine and the abduction of the KA subsystem inference engine is an abstract one. In this actual implementation, the two are combined in a single iterative deepening search. When the cost bound is 0 (initial), then basic backchaining is performed. If this does not work, then the cost bound increases and abduction is performed.

4.5 THINK in Operation

Figure 4.6 is output from three actual THINK sessions. Some unimportant output has been removed and annotations have been added at the right. User input and responses follow the "| :" prompt. Between sessions the ALNs were adapted to incorporate newly acquired training samples. Before the first session, the KB was completely empty except for hierarchal information. The hierarchy is the same as that in figure

4.4 with the addition of *collie*, *tabby*, and *canary* predicates in expected places. The first session has no benefit of any other previous knowledge or training samples but still manages to acquire knowledge items to solve its problem. The first user response on line 5 says that the candidate, $yellow(tweety) \rightarrow fly(tweety)$, is "poor" (nonsensical), and that it is "never" true ($not(yellow(tweety) \rightarrow fly(tweety))$ should be added to the KB facts). The second response on line 7 says that the strengthened candidate, $not(yellow(tweety) \rightarrow fly(tweety))$ is "good" (is sensible) and is "always" true (should be added to the KB facts).Here, there is a high correlation between positive first and second parts of the response and between negative first and second parts of the response, but this need not always be the case. Of note in the first session is the amount of strengthening from the second search candidate. Lines 11, and 13 are strengthenings from the candidate presented at line 9. Line 15 is a strengthening of the negative response to line 13. Lines 17 and 19 are, in turn, strengthenings of the negative response to line 15. Line 21 is a strengthening based on universal generalization of the negative response to line 15. Finally, line 23 shows a similar strengthening for the positive response to line 11.

The second session shows some benefits from the experience of the first session. Here, the ALNs seem to have learned that the *canary* predicate is "good" to have on the left hand side when a $move_i$ predicate is on the right. The ALNs have seen no samples indicating the importance of the identity bit, though so the candidate at line 38 is put forth. The ALNs have learned that *yellow* is not a "good" predicate on the left here but line 41 shows that they have not learned that $coloured_i$ is also not generally "good". A line of strengthening analogous to that which followed line 9 was not repeated after line 46 because the strengthened candidates did not meet the one-third ALN agreement threshold required to put them forth. This saved on several unneeded questions but a useful strengthening ($dog(lassie) \rightarrow run(lassie)$) was missed.

The third session displays even more experience. The needed KB item was quickly found. The lack of any literals involving *tweety* or *lassie* shows that the importance

```
 1. |: fact(canary(tweety)).                          { Observations entered. }
 2. |: fact(yellow(tweety)).                          { ---------------------- }
 3. |: explain(fly(tweety)).                          { Query }
 4. Maybe:  yellow(tweety)causes fly(tweety)          { First candidate }
 5. |: poor, never.                                   { Response }
 6. Maybe:  not(yellow(tweety)causes move(tweety))    { Strengtening of response
 7. |: good, always.
 8.
 9. Maybe:  canary(tweety)causes fly(tweety)          { Next search candidate }
10. |: good, always.
11. Maybe:  bird(tweety)causes fly(tweety)            {-- ------------}
12. |: good, always.                                 {                }
13. Maybe:  animal(tweety)causes fly(tweety)         {                }
14. |: poor, never.                                  {                }
15. Maybe:  not(animal(tweety)causes move(tweety))   {                }
16. |: poor, never.                                  { Strengthenings }
17. Maybe:  animal(tweety)causes leap(tweety)        { and responses  }
18. |: poor, never.                                  {                }
19. Maybe:  animal(tweety)causes run(tweety)         {                }
20. |: poor, never.                                  {                }
21. Maybe:  animal(X)causes move(X)                  {                }
22. |: good, always.                                 {                }
23. Maybe:  bird(X)causes fly(X)                     {                }
24. |: good, always.                                 {- ------------}
25.
26. Explanation for fly(tweety):  bird(tweety)       { Solution }
27. ------------------------------------------------------------------ ------------
28.                    .
29.                    .                             { ALNs adapted }
30.                    .
31. ------------------------------------------------------------------------------
32. |: fact(collie(lassie)).                          {-- ------------------ -----}
33. |: fact(brown(lassie)).                          { Observations entered }
34. |: fact(canary(tweety)).                         {                      }
35. |: fact(yellow(tweety)).                         {- --------------------}
36. |: explain(run(lassie)).                         { Query }
37.
38. Maybe:  canary(tweety)causes run(lassie)         { Search candidate }
39. |: poor, never.
40.
41. Maybe:  brown(lassie)causes run(lassie)          { Search candidate }
42. |: poor, never.
43. Maybe:  not(brown(lassie)causes move(lassie))    { Strengthened candidate }
44. |: good, always.
45.
46. Maybe:  collie(lassie)causes run(lassie)         { Search candidate }
47. |: good, always.
48. Maybe:  collie(X)causes run(X)                   { Strengthened candidate }
49. |: good, always.
50.
51. Explanation for run(lassie):  collie(lassie)     { Solution }
52. ------------------------------------------------------------------------------
53.                    .
54.                    .                             { ALNs adapted }
55.                    .
56. ------------------------------------------------------------------------------
57. |: fact(tabby(morris)).                          {----------------------}
58. |: fact(orange(morris)).                         {                      }
59. |: fact(collie(lassie)).                         { Observations entered }
60. |: fact(brown(lassie)).                          {                      }
61. |: fact(canary(tweety)).                         {                      }
62. |: fact(yellow(tweety)).                         {----------------------}
63. |: explain(leap(morris)).                        { Query }
64.
65. Maybe:  animal(morris)causes leap(morris)        { Search candidate }
66. poor, never.
67.
68. Maybe:  tabby(morris)causes leap(morris)         { Search candidate }
69. |: good, always.
70. Maybe:  cat(morris)causes leap(morris)           {-------------------------}
71. good, always.                                    { Strengthened candidates }
72. Maybe:  cat(X)causes leap(X)                     {                         }
73. good, always.                                    {-------------------------}
74.
75. Explanation for leap(morris):  cat(morris)       { Solution }
76.
```

Figure 4.6: A THINK Session

of the identity of arguments is important. Also, the useful strengthening missed in the second session was not missed here. Line 65 demonstrates the problem with the simple solution to the problem described in section 4.3.1. Here, the more general candidate is suggested first. This candidate is *just* too general but since it was generated in the search phase (rather than during strengthening), a negative training sample is generated from it. Unfortunately, this is undesirable as it closely matches the desired positive pattern of $animal \rightarrow move$.

These example sessions show the ability of this implementation to learn to generate better KB candidates before poorer ones. It worked well here likely because of the clean hierarchy with classes closely related to the causal links as prescribed near the end of section 4.2.3. Things are not always as clean as this, though, and so the learning performance is not always as impressive. More realistic results from a more realistic KBS are given in Chapter 6.

Chapter 5

Related Work

This chapter reviews some of the other research on KB refinement and on similar methods of machine learning. This other work is compared with THINK so to highlight the relative advantages and disadvantages of THINK.

5.1 TEIRESIAS

TEIRESIAS is a KBS which made several contributions to KR and reasoning in general. With respect to KA, its prime contribution is an interactive KB debugging method using *rule models*[7].

The performance system of TEIRESIAS is similar to THINK in that it is based on backchaining along associational links. The associations are premise-action associations. The main differences are that TEIRESIAS uses certainty factors and it does not use a hierarchy.

After the system performs a task, it checks with the user (if the user is known to be an "expert") for an evaluation of the result. If the result is wrong, TEIRESIAS enters a KB debugging session. Like THINK, the session is interactive and takes good advantage of the problem context. It begins by asking what parts of the result were wrong and what the correct result should have been. Through a series of leading questions which refer to actual KB rules, the user is led back along the lines of

reasoning to find the problem in the KB. Typical leading questions are like: "Should it have been possible to conclude ⟨some premise⟩ for ⟨some rule⟩?"; (after listing rules applicable to some action) "Should there be another rule?"; and "Is ⟨some rule⟩ correct?" Eventually, the user is asked to modify or add a rule or rules.

Rule Models

Rule models are used in TEIRESIAS for two main reasons:

- During interpretation of a natural language input of a new rule, rule models allow some predictions on the rough form and content of the new rule.
- After a new rule has been interpreted, the model provides a base for evaluating or "second guessing" the new rule.

A rule model is based on a subset of the KB rules. The subsets are grouped according to classes of the action part of the rule. For example, rules that conclude about some property, $\langle X \rangle$, would form the basis for a model. The models can be more or less specific than each other as well. For example, there can be models for rules that conclude that $\langle X \rangle$ holds and models for rules that it does not hold. Both would be more specific than the model for rules *about* $\langle X \rangle$.

A rule model characterizes the members of the associated subset according to the contents of the premise and action parts of the rule. For each part, a list of commonly occurring forms is provided as well as a list of close correlations of forms in each part. Each listing is accompanied by a measure of how common or closely correlated the item is. These lists are constructed through simple statistical analysis of the member rules. An example from [7] is reproduced in part in figure 5.1. This example from an investment advice system models rules that have conclusions about investment areas. In such rules, premises about the desired return rate and investment time scale being either the same or not the same as some value are common. Also common, but less so, are premises about the market trend being the same as some value. When there is a premise about the desired return rate being the same as some value there is

```
PREMISE  (  (RETURNRATE SAME NOTSAME 3.8)
            (TIMESCALE SAME NOTSAME 3.8)
            (TREND SAME 2.8)
            ((RETURNRATE SAME) (TIMESCALE SAME) 3.8)
            ((TIMESCALE SAME) (RETURNRATE SAME) 3.8)
            ((BRACKET SAME) (FOLLOWS NOTSAME SAME)
                                        (EXPERIENCE SAME) 1.5))

ACTION   (  (INVESTMENT_AREA CONCLUDE 4.7)
            (RISK CONCLUDE 4.0)
            ((INVESTMENT_AREA CONCLUDE) (RISK CONCLUDE) 4.7))
```

Figure 5.1: A rule model

usually a premise about the timescale being the same as some value and vice versa. Also, if there is a premise about the income bracket of the investor being the same as some value, there are usually premises about how he follows the market and about his investment experience. There are similar listings for the action part of the rules. Note that there can be multiple parts to an action in TEIRESIAS.

An example use of a rule model for "second guessing" an expert's new rule would be if he enters a rule like: "If 1) the investor's income bracket is 50%; and 2) the investor follows the market closely; then the investment area should be high technology." In this case, TEIRESIAS would suggest the inclusion of a premise about the investment experience of the investor as well as the inclusion of an action about desired risk of the investment.

Evaluation

TEIRESIAS takes advantage of the problem context to (together with the user) come up with missing rules. Like THINK, it uses an auxiliary KB (rule models) to help in this KA. The TEIRESIAS rule models come from the primary KB whereas the THINK auxiliary KB (ALNs) derive more directly from the interactive KA process.

In the end, TEIRESIAS is likely to ask fewer but harder questions than THINK does. For example, TEIRESIAS will just ask the user to enter a new rule for some needed action and THINK will suggest a series of rules on its own until it hits on one

the user likes. Questions like "Should $\langle X \rangle$ have been able to be concluded?" require a bit more insight by the TEIRESIAS user into the reasoning process of the system. Also, TEIRESIAS will lead the user around the reasoning lines until the weakness is identified. THINK will, on its own, find a KB hole which has the most likelihood of being filled and which will allow the problem to be solved.

It is difficult to provide anything other than a subjective, qualitative comparison as above because it appears that there has been no formal, quantitative evaluation of TEIRESIAS to compare with the formal experimental results for THINK provided in Chapter 6. Davis, in [7], remarks speculatively on the relative utility of TEIRESIAS KA in different domains. There is no mention of any learning or improving ability of TEIRESIAS with experience. Here, THINK is shown to improve with experience.

5.2 SEEK and SEEK2

SEEK [22] and its successor SEEK2 [9, 10] are systems which perform KB refinement for a production system. In the performance system, the productions have the form: If $C_1, \ldots, C_n$, then conclude D with confidence, cf, where D is some diagnosis and the C_i are called *choice components*[1]. If each choice component of a rule is satisfied, then the rule fires. The individual choice components, C_i, have the form: $k : c_1, \ldots, c_m$, where the c_i are distinct findings (intermediate or observed). The choice component is satisfied if at least k of its m findings are true.

Empirical Analysis

SEEK and SEEK2 use a case base of problems with known solutions to perform an empirical performance analysis of the current KB. The analysis provides information on which cases were correctly and incorrectly diagnosed by the performance system as well as indications of the strong and weak areas. The analysis is broken down in

[1]Actually, the productions in SEEK are a restricted form of this but the distinction is not important here.

two main ways:

1. For each type of diagnosis, the analysis tells how many times the diagnosis was missed (false negatives) and how many times the diagnosis was incorrectly concluded (false positives).

2. For each rule, the analysis tells how many times the rule was responsible for a misdiagnosis (false positive). Also, various statistics are given on cases which had the rule's conclusion as the correct diagnosis including: when the system missed this diagnosis and the rule did not fire; when it would have made a difference[2]; how close it was to firing in these cases; what is the most frequently missed choice component in these cases.

KB Refinement

Based on the first breakdown above, the user (in SEEK) or the system (in SEEK2) selects a diagnostic area on which to try to improve performance. Then the rules responsible for concluding that diagnosis are looked at with respect to the second breakdown. There are a number of heuristics which will suggest specific refinements according to the empirical analysis. The refinements fall into two categories:

Generalization: If a rule with the correct diagnosis as its conclusion did not fire and *consequently*, the system missed the diagnosis; and this happened often, then the rule should be *generalized* (made easier to fire). Of course, if the rule is already responsible for many false positives also, then this is not so certain. One way to generalize is to take the most frequently missed choice component of the rule and reduce the k value for it by one. Another form of generalization is to take a rule which *did* fire for the correct diagnosis but which had too low a cf value. For this rule, the cf value could be raised. This second method is more drastic.

[2]It might not make a difference had such a rule fired if the cf value is still too low.

Specialization: If a rule is responsible for many false positives then it should be *specialized* (made more difficult to fire). One way to specialize a rule is to increase the k value for one of the choice components. Another form of specialization is to reduce the cf value for a responsible rule.

There are other forms of generalization and specialization with associated heuristic rules, but the above gives the general flavour.

After the system generates the set of refinements which are likely to improve performance, the SEEK user selects one to "try out". In SEEK2 the system just tries them all, one at a time. To try out a suggested refinement, it is *temporarily* incorporated into the KB and all the stored cases are run over again. Then, a new analysis is done and if the user likes the results, he can tell the system to keep the refinement permanently. In SEEK2, the system just keeps the refinement with the best improvement.

This whole process is a cycle which is repeated until the SEEK user is satisfied or, for SEEK2, there is no more possible improvement.

Evaluation

SEEK and (even more so) SEEK2 work much more on their own than TEIRESIAS. Like THINK, they infer changes to the KB rather than prompting the user for them. The SEEK work did not include any formal quantitative evaluation of its method. This is not to be confused with the fact that the empirical analysis can evaluate individual changes for the user. For example, there is no indication (experimental or otherwise) how, or even if, better refinements are suggested when there is a larger case base.

The main evaluation measure for THINK is how much user interaction is necessary. SEEK2 goes to an extreme in this respect with no user interaction at all. One question is whether this goes too far. This can be a debatable and certainly domain dependent issue.

The SEEK2 method is supported by a solid set of experiments. Naturally, perfor-

mance will improve on the case base used for the empirical analyses. These are used to generate refinement recommendations and then to select which ones to actually incorporate. To verify the SEEK2 method, a case base is divided randomly. One part is used as a test set. The performance on the test part is evaluated before and after running SEEK2 with the other part. The process is redone (from start) but with the roles of the two parts switched. In both cases the performance improves on the test part after running SEEK2 on the other. A similar principle applies in the testing of THINK. Tests are run in various sequences, but never are two similar tests in the same sequence (See Chapter 6.).

The main differences are what make comparison difficult. Both SEEK and SEEK2 take an *overall* approach to refinement; they generate several refinements which will improve the system for many cases. THINK, on the other hand takes an *incremental* approach: it generates candidate(s) which will help in bet one problem. The SEEK advantage is a likely better overall improvement of the system and the THINK advantage is an immediate improvement which can be incorporated right when needed in solving a problem. Another difference, is that SEEK and SEEK2 only refine *existing* rules while THINK is capable of generating complete rules from scratch. This does not grant an advantage either way. Both kinds of KB update are useful.

5.3 APT

APT [19] is a system which does intelligent KA for a performance system based on STRIPS like rules and a hierarchy of properties (much like the hierarchy for monadic predicates in THINK). The hierarchy is used to allow STRIPS like rules defined for general kinds objects to be applied to more specific kinds. An example rule (from [19]) is shown in figure 5.2. Now, if a goal was to have a *box* which is currently at an *assembly line* to be at the warehouse, then the rule above could be applied if *box isa object* and *assembly line isa line* were parts of the hierarchy. Before a rule is actually used, the user is asked to validate the rule choice. This just gives the user a chance

Preconditions	Location of Object is a Line
	Location of Robot is the Warehouse
Problem	MOVING
Subproblems	Robot moves to the Line
	Robot takes the Object
	Robot moves to the Warehouse
Post-conditions	Location of Object and Robot is the Warehouse

Figure 5.2: An example APT rule

to say "do not use that rule; use some other."

APT is much like THINK in that it switches into a KA phase when it cannot resolve a step in a problem. This way, APT derives all the benefit that THINK does in doing *incremental* KA in the context of an actual problem. When there are no rules that the user likes, the user is simply asked to provide his own. Naturally, the provided rule will be specific to the problem at hand. For example, if the rule above was missing and the current goal was to get a box, currently at the assembly line, to the warehouse, then the user might put forth a rule like the one above except with "Box" and "Assembly Line" instead of "Object" and "Line". Although, not as powerful as the above rule, it would do for the task at hand.

Rule Completion

Because users are not a knowledge engineers, they might not provide perfectly formed rules; or might miss some important precondition. For this reason, APT uses an explanation mechanism to check on the completeness of the preconditions. Suppose, in the above example, that the user forgets to include that the location of the robot is the warehouse as a precondition. APT would try to find an *explanation* for the first subproblem (Robot moves to the Line) and the explanation would be in the fact that the robot is *not* already at the assembly line. This would identify the need for the location of the robot in the preconditions and the current location of the robot (from the current state of the "world") would be suggested as an added precondition for the new rule. In this way the rule gets properly completed. This is a bit like the

"second guessing" mechanism described for TEIRESIAS.

Rule Generalization

After acquiring a new, problem specific, rule in this way, it attempts to generalize it using a process almost identical to the strengthening phase of THINK. By climbing the concept hierarchy, more and more general rules are suggested to the user until he rejects them. This generalization of rules is much like navigating a *version space* [18] of rule concepts. At this point, THINK and APT behave differently. THINK begins to generalize on the *negative* response while APT takes a different tack. Instead of just stopping at the most general and accepted version of the rule, APT takes one step further to the first *too*-general (rejected) version and tries to *specialize* that. This may identify versions of the rule which are siblings (with respect to generalization) to the most general and accepted version. These sibling versions are checked for acceptability as well; some of them will be accepted and some of them will not. By eliciting a new class in the hierarchy or new preconditions for the rule or both, APT specializes the first too-general version to cover all its accepted children versions in a single rule.

Evaluation

APT is closer to THINK in its methods than TEIRESIAS or SEEK. APT takes advantage of problem context; it acquires knowledge incrementally; most, but not all, of its questions to the expert are leading ones; and it uses a separate hierarchy both in its normal reasoning and in its KA process.

A significant difference between APT and THINK is in how APT starts its KA process. The initial problem specific rule is provided outright by the user when asked. There is no inference of this important item by APT. Getting started seems to be the hard part[3]. Also, in the specialization phase just described, the user takes a more active role.

[3]The author has found this to hold in writing chapters of this thesis as well.

The specialization phase of APT and THINK's generalization of *negative* responses end up having the same effect. To generalize a negative response, THINK *descends* the hierarchy to generate new candidates. This results in the same search that APT makes during its specialization phase. The effect is roughly the same: THINK ends up with a disjunction of accepted siblings while APT ends up with a single rule covering the same siblings. APT, however, requires more complex interaction with the user for specialization. The benefit of APT's method is the increased chance of acquiring a more cohesive and semantically better founded rule. This may include the acquisition of a possibly semantically important new node in the hierarchy[4].

There is some concern expressed in [19] about the possible number of rule versions proposed during generalization. It is felt that there should be some kind of preference criteria to restrict generalization so to cut down this number. THINK already does this with ALNs (see section 4.2.3).

As with TEIRESIAS, there is no reported quantitative evaluation for APT although it is reported to have been useful in a variety of applications. There is no indication that the KA process improves with experience (as is shown for THINK). Even if there is some improvement it is probably not as marked as with THINK because APT does not retain negative information which is known to be very valuable in THINK.

[4]THINK is able to acquire new hierarchy nodes too, but it is through *incidental* acquisition rather than active elicitation. If THINK encounters a predicate it has not seen before, it allows the user to navigate the hierarchy in order to place the new predicate.

Chapter 6

Experimental Results

This chapter presents the results of experiments with the current implementation of THINK. First, a description of the experimental design is given. Then, quantitative results which bear on the following predictions are provided.

1. After making a number of missing item suggestions and receiving cooperative responses, the system will eventually acquire the needed missing knowledge and solve the problem.

2. The more questions asked in previous tests in a given experiment[1], the fewer, on average, will be the number of "poor" questions asked during the current test.

An interpretation of the results raises new questions. These questions are briefly addressed with conjectured answers.

6.1 Experimental Design

To evaluate the system a well considered, nontrivial causal network (taken from [15]) is encoded into the system. Building on this, a considered hierarchy appropriate for the domain is also encoded into the system. The causal net and hierarchy are for an

[1] I.E., the more mature the system during the current test

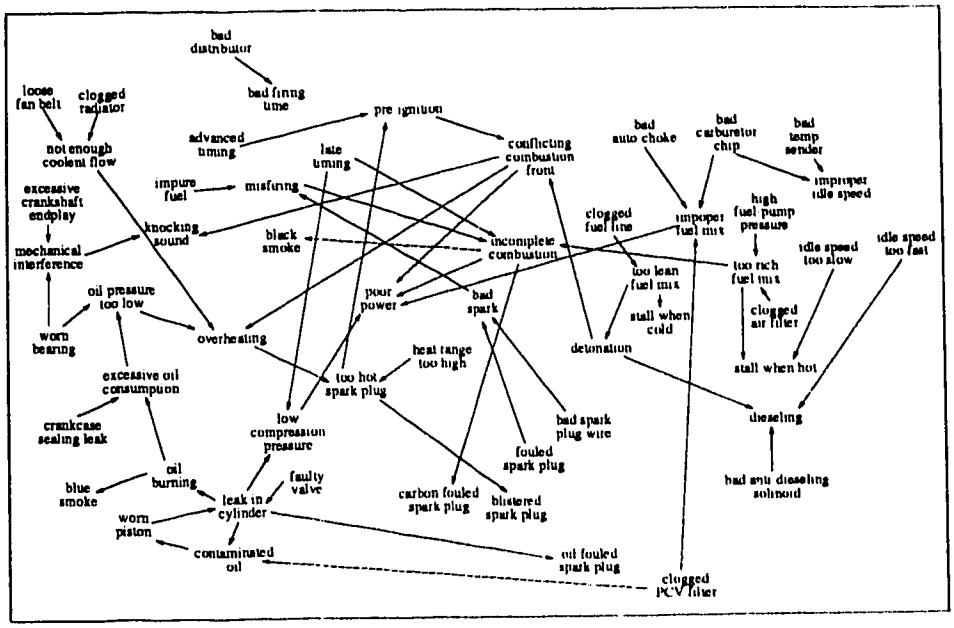

Figure 6.1: Causal Network for Automobile Diagnosis

automobile diagnosis application and are depicted in figures 6.1 and 6.2 respectively.

The aim of THINK is to be able to acquire missing knowledge items during problem solving and to get better at it with experience. It should be able to acquire literal items, links, or both. Five separate *tests* have been designed which require THINK to do this. Each test is based on the KB depicted in figures 6.1 and 6.2. This KB is a well considered one which makes real world sense. A problem scenario is chosen for which the KB has a solution and then needed items of knowledge are removed from the KB or, in the case of observations, not initially provided in the problem description. THINK is then asked to solve the problem (provide an explanation). Whenever THINK presents a candidate item, the user responds cooperatively. Statistics on the questions asked (good versus poor and strengthening versus non-strengthening) are maintained. Table 6.1 describes the actual tests used.

In order to give each test a chance to be run with various amounts of previous experience, the 5 tests are run in 10 different sequences. Although, there are $5! = 120$ possible sequences, the 10 chosen provide good coverage. Each test is twice run in

TEST	SCENARIO	MISSING KNOWLEDGE
1	*clogged_air_filter* → *too_rich_fuel_mix* → *incomplete_combustion* → *black_smoke*	*clogged_air_filter*
2	*clogged_pcv_filter* → *contaminated_oil* → *worn_piston* → *leak_in_cylinder* → *oil_burning* → *blue_smoke*	*contaminated_oil* → *worn_piston*
3	*clogged_fuel_line* → *too_lean_fuel_mix* → *detonation* → *conflicting_combustion_front* → *knocking_sound*	*clogged_fuel_line* *detonation* → *conflicting_combustion_front*
4	*faulty_valve* → *leak_in_cylinder* → *low_compression_pressure* → *poor_power*	*faulty_valve* → *leak_in_cylinder* *low_compression_pressure* → *poor_power*
5	*worn_bearing* → *oil_pressure_too_low* → *overheating* → *too_hot_spark_plug* → *blistered_spark_plug* →$_{isa}$ *fouled_spark_plug* → *bad_spark* → *misfiring* → *incomplete_combustion* → *black_smoke*	*worn_bearing* *overheating* → *too_hot_spark_plug* *bad_spark* → *misfiring*

Table 6.1: Experimental tests

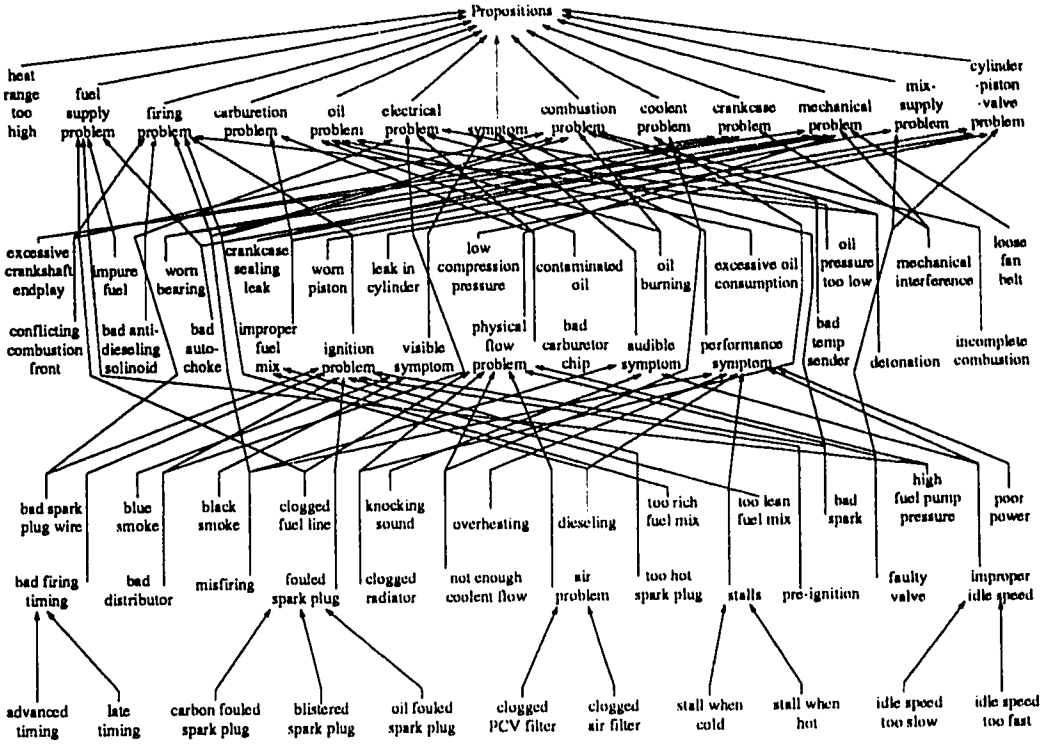

Figure 6.2: Hierarchy for Automobile Diagnosis

each possible place in a sequence and, for each test, the other tests appear before and after it an equal number of times in the various sequences. From the perspective of an individual test, the only variety gained in going to 120 sequences is in the sub-ordering of the other tests either before or after it in a sequence. Each *sequence* begins the same with a complete main KB and randomly initialized ALNs with an empty training set. Before each *test* in the sequence the appropriate "missing knowledge" is removed from the KB. Before the first test and between each of the others, the ALNs are trained with the current training set as well as all the current KB items included as positive examples. The training sequences used are listed in table 6.2

6.1.1 Testing without ALNs

To provide a control for the experiments, the same sequences were run with the ALNs decoupled from the system. In these runs, wherever an ALN result was needed, a unanimous, positive result was assumed. Actually, the control is only valid for a test

SEQUENCE NAME	TEST SEQUENCE
A	1 2 3 4 5
B	5 4 3 2 1
C	2 3 4 5 1
D	1 5 4 3 2
E	3 4 5 1 2
F	2 1 5 4 3
G	4 5 1 2 3
H	3 2 1 5 4
J	5 1 2 3 4
K	4 3 2 1 5

Table 6.2: Test sequences

when it is run first in a sequence. This is because learning still occurs because of the accumulation of negative knowledge on the candidates. It is still very useful to run the entire sequence to compare how well the system learns both with and without the ALNs. The expected result is that there will be some learning both ways but that the learning should be more pronounced with the ALNs.

6.2 The Results

To begin, an initial anecdotal report of some results will help explain the presentation style and focus of the main results. First of all, most (about two thirds) of the candidates presented are from strengthening. This is not surprising because each candidate from search can often be strengthened a couple of steps before strengthening ends. Also, most (about 90 percent) of the strengthening candidates are "good" ones. This is also not surprising because they derive from known true KB items. Because the performance with respect to strengthened candidates is thusly not very interesting and because the strengthening technique is not new[2], the focus of the presented results will be on the search candidates.

Recall the first prediction at the start of this chapter: After making a number of

[2]It is done in APT. See section 5.3.

missing item suggestions and receiving cooperative responses, the system will eventually acquire the needed missing knowledge and solve the problem. This is not a hard prediction to make considering that the iterative deepening search used is exhaustive. It came true in every test in every sequence with and without ALNs. No further address of this prediction need be made.

The second prediction was: The more questions asked in previous tests in a given experiment (sequence), the fewer, on average, will be the number of "poor" questions asked during the current test. Here, previous experience is measured in total (including strengthening questions) questions occurring previously in a sequence. Although results regarding strengthened candidates are not considered important here, the effect of all previous candidates cannot be disregarded in looking at performance in search candidate generation. The measure of performance in a single test is in the number of "poor" search candidates presented — the fewer the better. The overall, adaptive, performance is seen in the relationship between experience prior to a test and the performance during it — the more test performance improves with experience, the better.

Experimental results of the above measures are presented graphically in figures 6.3 through 6.7. Each point on a graph represents the results of a single test in some sequence. The position along the horizontal axis represents the *total* number of candidates presented in *all* other tests *before* the test in its sequence; this is the prior experience. The position along the vertical axis is the number of "poor" *search* candidates presented during the test; this is the performance measure with better performance lower on the scale. Points which lie along solid lines are from runs which use the ALNs and points along the dashed lines are from runs without ALNs. Adaptive performance is seen in how the lines descend from left to right — the more descent the better. It is expected that both lines on a chart will descend and that the ALN (solid) line will descend more than the other.

Figure 6.8 summarizes the aggregate for all five tests. The vertical scale on this chart is the average (among the five tests) number of poor search questions *as a percent*

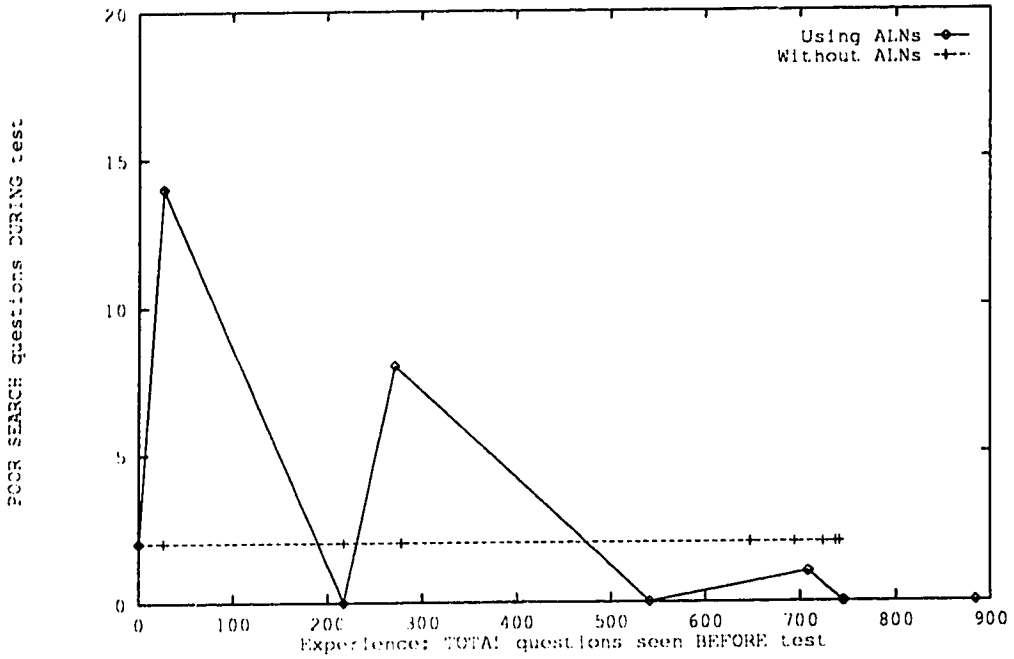

Figure 6.3: Test 1 results

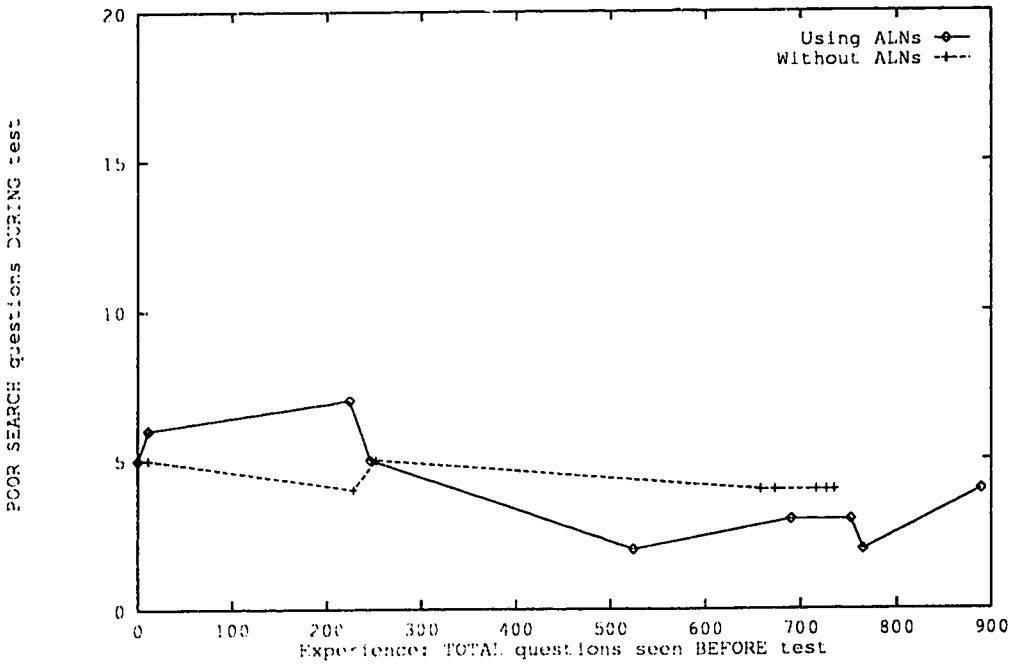

Figure 6.4: Test 2 results

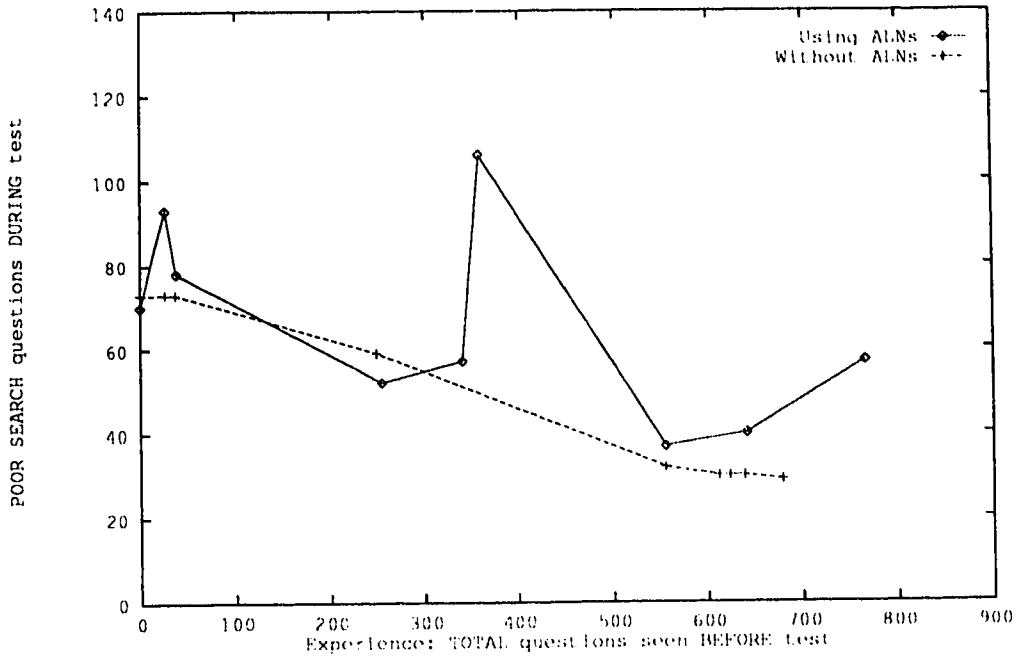

Figure 6.5: Test 3 results

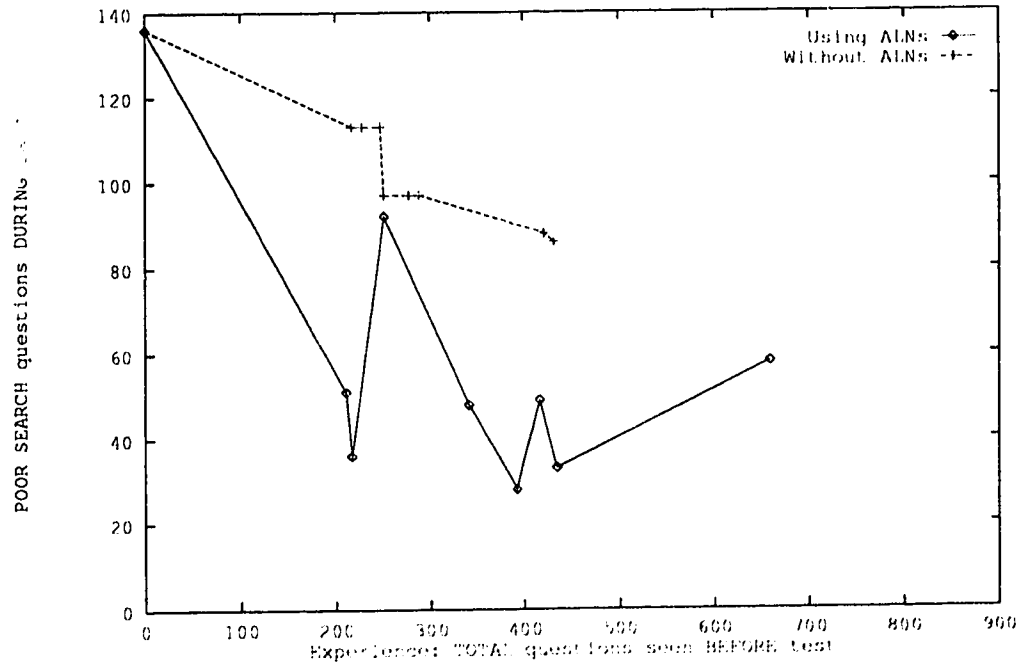

Figure 6.6: Test 4 results

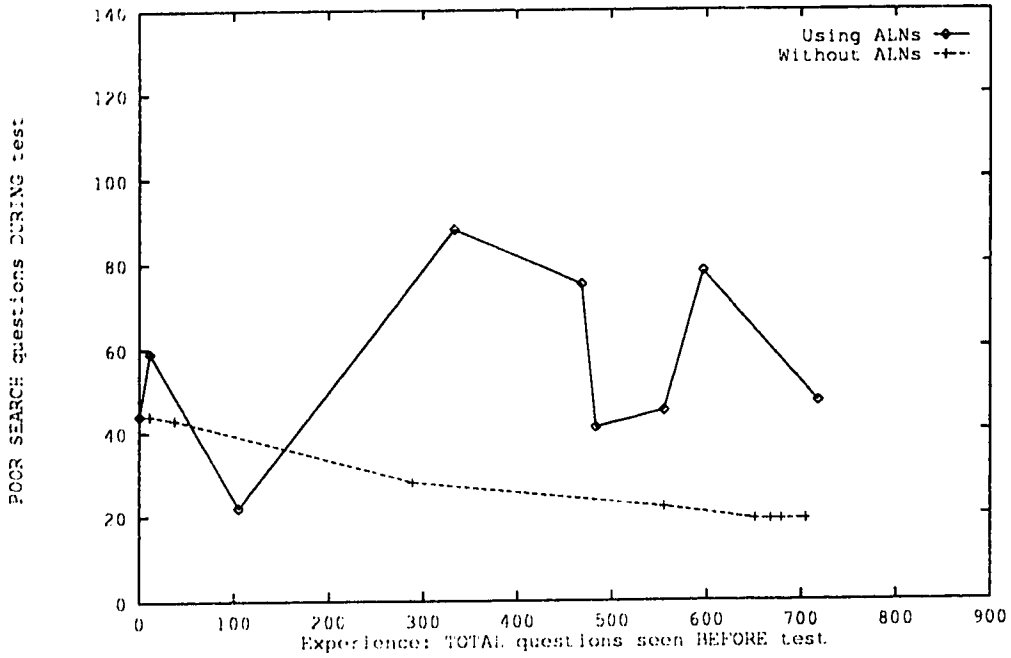

Figure 6.7: Test 5 results

of the number of poor search questions presented when there is no prior experience and without ALNs. This provides a normalization which weights equally the relative performance of each test regardless of the difficulty of the test. To illustrate how this chart is constructed, the computation of one of the points is given in table 6.3. The point from the table is that on the ALN (solid) line at position, (105, 151). Each row of the table shows the contribution from each test and this can be seen in each of the charts for the individual tests. The first column just labels the test. The second two columns show data points from the individual test which bracket the experience (prior questions) value of 105. The next column is an interpolation to 105 of the bracketing points. The next column gives the normalization divisor; it is the value for the non-ALN line at zero prior experience from the individual chart. The final column is the interpolated value as a percent of the normalization divisor. The bottom row reports the average of the last column and is what is used as the plot value at 105.

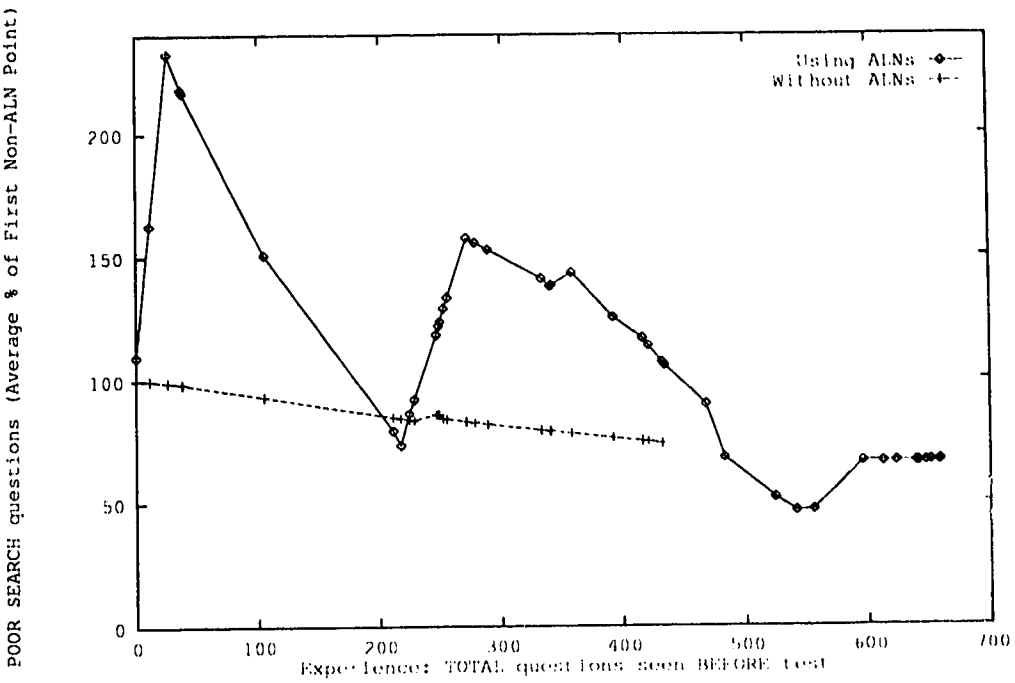

Figure 6.8: Aggregate test results

TEST	Data Points		Interp.	Norm. Div.	Percent
1	(26, 14)	(217, 0)	8.21	2	411
2	(11, 6)	(224, 7)	6.44	5	129
3	(38, 78)	(255, 52)	69.97	73	96
4	(0, 136)	(211, 51)	93.70	136	69
5	(105, 22)	(105, 22)	22.00	44	50
Average					151

Table 6.3: Sample aggregate point computation for ALNs with prior experience of 105 questions

6.3 Interpretation and Discussion

ALN Instability

One of the first things that will be noticed in the charts is the instability of the ALN lines compared to the non-ALN lines. Considering the random initializations of ALNs, this is not too surprising. Each point on a chart is from a different sequence and the ALNs are reinitialized between sequences. It makes sense that different initializations will result in different behaviour: learning rates should vary. It is even reasonable to expect the ALN tests to possibly perform worse than non-ALN tests when there is little prior experience. In any case, learning should occur and after some experience the ALN line should end up below the non-ALN line. This is best seen in the Test 1 chart.

Expected and Unexpected Results

As stated in the previous section it was expected that the non-ALN line would descend. Without the ALN instability, this descent should be fairly steady. This result was borne out for all the tests except for Test 1 which had a line which did just stayed constant. This line, however, started out at a very low value already.

The ALN line (although unstable) was expected to descend as well and to do so more than the non-ALN line. This is borne out for Test 1, Test 2, and especially Test4. Test 3 and especially Test 5 charts are not as expected, though. Some possible explanations for this are given below:

- Recall from near the end of section 4.2.3 that the hierarchy must be very well considered. Perhaps the initial design of the hierarchy is flawed. In retrospect, this seems likely when one considers the missing link from Test 5, $overheating \rightarrow too\_hot\_spark\_plug$. The hierarchy used here has the links, $overheating \rightarrow_{isa} performance\_symptom \rightarrow_{isa} symptom$. It is very likely that the ALNs would learn in general (based on the many other presented candidates) that *symptom* class propositions do *not* cause much of anything.

One way of testing this hypothesis would be to remove the *overheating* $\rightarrow_{isa}$ *performance_symptom* link from the hierarchy and run the tests again. Preliminary experimentation indicates that this only partly explains the result.

- Recall from the later part of section 4.3.1 that an inferior solution to an ALN input encoding problem was used in this THINK implementation and that this problem was manifested in one of the three sample THINK sessions of figure 4.6. It cannot be determined or further supported with the data collected from these experiments, but it could be that the same problem is manifested in Test 3 and Test 5 runs here. The obvious way of investigating this is to implement the better solution and redo the experiments. Further experimentation along this line has not yet been tried.

- One thing that Test 3 and Test 5 have in common is that they both include a missing literal item as part of the missing knowledge. When the hierarchy was designed, thought was mostly given to creating classes such that members of one class tend to have causal relations to members in other classes (as described at the end of section 4.2.3). In other words, thought was given to the ability to recognize good links and not much thought was given to recognizing good literal KB items. In this test application, a good literal item is one which is reasonably directly observable (i.e., an "askable"). Perhaps, the addition of more classes would help[3]. This is not too certain, though, because many existing classes such as *temperature_problem* and *mechanical_problem* already should help in this distinction.

The above explanations are based on minor issues of domain knowledge and knowledge representation. These are bound to arise in any first implementation of a knowledge-based system, and the KA subsystem is, after all, one of these. In view of this, in

[3]Of course the addition of two classes, *askable* and *not_askable* is an obvious idea, but this is not the point here. The idea here is to test THINK's ability to learn less well defined criteria for "good" and "poor" candidates. In a practical system, if things are easily labeled as askable or not, then that is what will be done.

TEST	AVERAGE SEARCH QUESTIONS	
	ALNS	NO ALNS
1	6.7	5.0
2	6.1	7.4
3	82.2	57.2
4	77.4	120.6
5	70.5	49.1

Table 6.4: Average search questions

view of the plausible explanations above, and in view of the positive results gained for tests 1, 2, and 4, these results bear support for the thesis of this research: a useful KA subsystem for acquiring missing associational knowledge is possible which combines abduction and the application of ALNs to perform adaptive hypothetical inference to generate candidate missing KB items to present to an expert for verification.

6.3.1 An Observation and Explanation

Another thing which stands out is that Test 1 and Test 2 runs always take many fewer candidates than the others, regardless of prior experience. The average (over all sequences) number of good and poor search questions for each test type is given in table 6.4. The question which arises is: Why is there such a jump between Test 2 and Test 3 and no corresponding jump between Test 4 and Test 5? Recall from section 4.2.3 that the iterative deepening is done in a nested loop with the outer loop limiting the number of missing items on any path in the search space. This explains the jump between Test 2 and Test 3. The search space is much larger when more missing items are allowed on a path. Why, then is there no increase between Test 4 and Test 5? The reason is that once the limit is increased to 2, the entire search space is effectively thrown wide open in this respect. This is because with a limit of 2, a link and a success leaf can always be hypothesized. This allows a link branch to by *exercised*. If it is successful, then the search space below that branch is expanded as described in section 4.2.2. This exercise-expand process can continue through the whole search space.

Chapter 7

Conclusions

7.1 Summary

This research has shown that an adaptive knowledge acquisition method can be integrated into the operation of a knowledge based system to acquire missing knowledge in the context of a problem. It has been shown that the general THINK framework depicted in figure 2.1 can be filled in as in figure figure 4.5 to yield a working system. This framework is flexible. Any kind of KA reasoning methods and auxiliary KR system which is appropriate to an application can be "plugged in". Indeed, the use of a neural network here testifies to this flexibility.

As well as the above, the specific implementation here has also shown:

- Subsymbolic and symbolic computing methods can be effectively used together in a simple unifying framework. This is not only shown in the integration of the KA subsystem with the main reasoning system but also *within* the KA subsystem where abduction and ALNs work together.

- Neural nets and ALNs in particular can perform inductive reasoning. Based on previous examples good knowledge item candidates are inductively distinguished from poorer ones. This was done with a fair degree of effectiveness in most cases.

- The use of ALNs or any other kind of uncertain method to guide the search for candidates does not have to compromise the completeness of the search which can be had by symbolic means. By adding a *bounded* cost to the search branches, the ALNs order the search space but do not prune it.

- Multiple kinds of knowledge and KR schemes can be used together. In the system implemented here there are three main kinds of knowledge: causal associational links, hierarchal links, and ALNs with their training sets. So long as the semantics of the various kinds of knowledge are held constant and uniform, the use of a single kind of knowledge can be used in multiple ways. Here, the hierarchal knowledge is used in the main backchaining reasoning process; in the strengthening process; and in the ALN input construction process.

The experimental procedure developed for this research is simple and widely applicable. Any incremental knowledge acquisition system can be evaluated by removing known needed knowledge elements from a knowledge base and then exercising the system to re-acquire the knowledge. Measurements of effectiveness are derived from interaction statistics collected during the exercise. Valuable statistics would be the number of atomic interactions, the number of negatively appraised interactions, or anything else deemed important for the user and the system.

The main limitation of any THINK-like system will lie in the knowledge and the knowledge representation of the auxiliary knowledge base. After all, the THINK knowledge acquisition subsystem is a knowledge-based system unto itself and this is a critical performance factor in any knowledge-based system. This point was evidenced in the implementation here. The poor learning result for test 5 may be an indication of the importance of getting the hierarchy right.

The use of domain specific and domain independent heuristics at the symbolic level to help guide or even prune the candidate search space was not addressed in this work. In fact, such heuristics were avoided so to better observe the performance of the raw THINK implementation here. Almost certainly, even better performance (fewer poor candidates) could be achieved with the use of more heuristics. This would

be true both with and without the use of ALNs.

7.2 Future Work

Future work on THINK should lie in exercising the flexibility of the framework as well as addressing weaknesses observed here. To begin, the further experiments suggested in section 6.3 should be tried and their results explained. Other suggestions are provided below.

7.2.1 ALN Inputs

The more relevant input there is to the ALNs, the better. In this implementation, context bits were included as well as an encoding of the candidate itself. Also the hierarchal information was very important for the generalizing ability. Other available information will probably be useful as well. One idea is to include information about existing links already associated with literals in the candidate. For example an indication of what is already known to be caused by and what is already known to cause the P and the Q in a $P \rightarrow Q$ suggestion could be useful. The original idea for including hierarchal information was that it seemed reasonable to try to find generalizations on what kinds of things cause other kinds of things. This new idea is to try and find generalizations like, "things that cause things in class A tend to also cause things in class B," and "things that are caused by things in class C tend to cause things in class D." Any other creative ideas for generalizations might also be tried.

7.2.2 Other Candidate Grading Methods

ALNs certainly are not the only way to learn to classify things. ALNs were used here for the reasons discussed in section 3.3. The integration of other inductive methods as suggested in section 3.3 should be tried and compared. Just as was necessary with

ALNs, appropriate input selection and encoding schemes will need to be developed for each.

7.2.3 Other Candidate Generation Methods

In the implementation here, abduction was used to generate candidates. Other methods should be explored. One idea is to reverse the roles of symbolic and subsymbolic computing methods and use a neural net to *construct* candidates given the current subgoal or requirement and then use symbolic heuristic methods to grade them. This would have an advantage of automatic grading which would allow more automatic generation of training data for the adaptive part of the system.

7.2.4 Acquisition of Multiple Kinds of Links

Although, as pointed out in section 7.1, multiple kinds of knowledge are used in THINK, the object of acquisition is only one kind – one kind of associational link. Pearl's use of causal and evidential links [20] shows the usefulness of multiple kinds of links in a knowledge-based system. If knowledge based systems are going to use semantically different kinds of associational links then THINK should be able to acquire multiple kinds of links in one system. Investigation into how to do this would be valuable.

7.2.5 Domain and Application Considerations

All of the above ideas were stated generally with respect to domains and applications. Certainly, some ideas, such as auxiliary KB encodings, will be better suited to some domains and applications than others. There should be some investigation into what THINK elements go best together depending on the application or domain.

Bibliography

[1] William W. Armstrong and Jan Geesei. Adaptation algorithms for binary tree networks. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-9(5):276–285, May 1979.

[2] William W. Armstrong, Jiandong Liang, Dekang Lin, and Scott Reynolds. Experiments using parsimonious adaptive logic. Tech. Rept. 90-30, University of Alberta, Edmonton, Alberta, 1990.

[3] G.V. Bochmann and W.W. Armstrong. Properites of boolean functions with a tree decomposition. *BIT*, 14:1 13, 1974.

[4] John H. Boose. Personal construct theory and the transfer of human expertise. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-84)*, pages 27–33, University of Texas at Austin, August 1984. American Association for Artificial Intellence.

[5] B.G. Buchanan, G.L. Sutherland, and E.A. Feigenbaum. Rediscovering some problems of artificial intelligence in the context of organic chemistry. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 253 280. American Elsevier, New York, 1979.

[6] Bruce G. Buchanan. Some approches to knowledge acquisition. In Tom M. Mitchell, Jaime G. Carbonell, and Ryszard S. Michalski, editors, *Machine Learning: A Guide to Current Research*, pages 19 24. Kluwer Academic, Hingam, Mass, 1986.

[7] Randall Davis. Interactive transfer of expertise: Acquisition of new inference rules. *Artificial Intelligence*, 12:121–157, 1979.

[8] Béatrice Duval and Yves Kodratoff. A tool for the management of incomplete theories: Reasoning about explanation. In Pavel B. Bradzil and Kurt Konolige, editors, *Machine Learning, Meta-Reasoning and Logics*, pages 135–158. Kluwer Academic Publishers, Norwell, Massachusetts, 1990.

[9] Allen Ginsberg, Sholom Weiss, and Peter Politakis. SEEK2: A generalized approach to automatic knowledge base refinement. In Aravind Joshi, editor, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, volume 1, pages 367–374, Los Altos, California, August 1985. Morgan Kaufmann.

[10] Allen Ginsberg, Sholom M. Weiss, and Peter Politakis. Automatic knowledge base refinement for classification systems. *Artificial Intelligence*, 35:197–226, 1988.

[11] Randy Goebel. A quick review of hypothetical reasoning based on abduct ion. In *AAAI Spring Symposium on Automated Abduction*, pages 145–9. Stanford University, March 1990.

[12] Robert Hecht-Neilsen. *Neurocomputing*. Addison-Wesley, Reading, Massachusetts, 1990.

[13] G.J. Hwang and S.S. Tseng. EMCUD: A knowledge acquisition method which captures embedded meanings under uncertainty. *International Journal of Man-Machine Studies*, 33:431–451, 1990.

[14] Gary Kahn, Steve Nowlan, and John McDermott. MORE: An intelligent knowledge acquisition tool. In Aravind Joshi, editor, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, volume 1, pages 581–584, Los Altos, California, August 1985. Morgan Kaufmann.

[15] Dekang Lin. *Obvious Abduction*. Ph.d. thesis, University of Alberta, Edmonton, Alberta, 1992.

[16] John McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88, 1982.

[17] Karen L. McGraw. *Knowledge Acquisition: Principles and Guidelines*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1989.

[18] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[19] Claire Nedellec and Karine Causse. Knowledge refinement using knowledge acquisiton and machine learning methods. In Th. Wetter, K.-D. Althoff, J. Boose, B.R. Gaines, M. Linster, and F. Schmalhofer, editors, *Current Developments in Knowledge Acquisition — EKAW '92*, pages 171 — 190, Heidelberg and Kaiserslautern, Germany, May 1992. Springer-Verlag.

[20] Judea Pearl. Embracing causality in default reasoning. *Artificial Intelligence*, 35(2):259–271, June 1988.

[21] Charles S. Peirce. The logic of abduction. In Vincent Tomas, editor, *Essays in the Philosophy of Science*, chapter XIII. The Liberal Arts Press, New York, 1957.

[22] Peter Politakis and Sholom M. Weiss. Using empirical analysis to refine expert system knowledge bases. *Artificial Intelligence*, 22:23–48, 1984.

[23] David Poole, Randy Goebel, and Roman Aleliunas. Theorist: A logical reasoning system for defaults and diagnosis. In Nick Cercone and Gordon McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 331–352. Springer-Verlag, New York, 1987.

[24] J. Ross Quinlan. Learning efficient classification procedures and their application to chess end games. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M.

Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 15. Morgan Kaufmann, Los Altos, California, 1983.

[25] Sholom M. Weiss, Casimir A. Kulikowski, and Saul Amarel. A model-based method for computer-aided medical decision-making. *Aritificial Intelligence*, 11:145–172, 1978.