INFORMATION FILTERING:

SELECTION MECHANISMS IN LEARNING SYSTEMS

by

Shaul Markovitch

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer and Communication Sciences)
in The University of Michigan
1989

Doctoral Committee:

Assistant Professor John Laird, Co-chairman
Professor Paul D. Scott, The University of Essex,UK, Co-chairman
Professor Yuri Gurevich
Professor Keki Irani
Associate Professor Robert Lindsay

ACKNOWLEDGMENTS

I would like to thank my advisor, Paul Scott, for the constant help he has given me during my stay at Michigan. I would like to thank John Laird for the careful reading and helpful comments. I would also like to thank the other members of my committee, Yuri Gurevich, Keki Irani, and Bob Lindsay for their guidance and support. Thanks to the management and staff of the Center for Machine Intelligence for being supportive and sympathetic. Finally, thanks to my family and my friends for supporting me all along the way.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The most important outcome of AI research during the 70s was the general acceptance of the major role of knowledge in intelligent systems (Buchanan & Feigenbaum, 1982). Lenat and Feigenbaum (1989) call this belief the *knowledge as power* hypothesis and assert it as:

> "**The knowledge principle (KP)** A system exhibits intelligent understanding and action at a high level of competence primarily because of the *specific* knowledge that it can bring to bear: the concepts, facts, representations, methods, models, metaphors, and heuristics about its domain of endeavor."

Or as Buchanan and Feigenbaum (Buchanan & Feigenbaum, 1982) put it, "the power of an intelligent program to perform its task well depends primarily on the quantity and quality of knowledge it has about that task."

Thus, it is not surprising that the general attitude toward knowledge was a greedy one - grab as much knowledge as you can. The general belief has been that the more knowledge you have, the better you are. This approach will be called "the monotonicity of knowledge value" to indicate that the approach views the performance of a system as a monotonic function of the amount of knowledge it possesses. A good illustrative example of this approach is a diagram drawn in (Lenat & Feigenbaum, 1989) which shows a monotonic improvement of performance as a function of the amount of knowledge.

Soon it was realized that often it is undesirable or impossible for humans to supply the needed knowledge. The field of machine learning then emerged to build programs that can fully or partially automate the

process of acquiring the desired knowledge. The shift of the burden of knowledge acquisition from human to machines brought two important consequences: it became possible to acquire much larger amounts of knowledge; and the knowledge acquired was of worse quality, because less intelligence was involved in acquiring it. As a result, several problems have been observed in learning systems in recent years, which have made researchers reconsider the indiscriminate approach to knowledge acquisition:

1. **Excessive/harmful potential experience**: The space of experiences that the learning system faces is too large to be handled by a resource-limited learner. Alternatively, there is a great variation within the experience space with regard to how useful experiences are for generating beneficial knowledge. This has been the case for almost all learning systems. Most systems overcome this problem by using a human teacher (Winston, 1975) who supplies the experiences to the system, thus making the selection for it. However, systems that generate their own experience (Carbonell & Gil, 1987; Lenat, 1983; Mitchell, Utgoff, & Banerji, 1983; Scott & Markovitch, 1989a) still face this problem. The same is true for systems that interact with complex environments.
2. **Unfocused attention**: The particular experiences that the learner faces are too complex to be handled by a resource-bounded learner. There are too many features or events in a particular experience to be considered.
3. **Excessive knowledge**: The amount of knowledge that the learning system generates is too large to be stored in the knowledge base. This problem was more significant in the earlier stages of AI research (Samuel, 1963) when machines had very limited storage capacity;

however, even today storage is limited, and Samuel's program would still be able to store only a tiny fraction of the set of possible checkers configurations.

4. **Non-monotonic acquisition**: Adding knowledge to the knowledge base occasionally leads to deterioration in the performance of the problem solver (Markovitch & Scott, 1988a; Markovitch & Scott, 1988b; Minton, 1988a; Tambe & Newell, 1988; Tambe & Rosenbloom, 1989).
5. **Non-optimized knowledge base**: There is a subset of the knowledge base with which the problem solver performs better than with the whole knowledge base (Markovitch & Scott, 1988b; Minton, 1988a; Wilkins, 1987).

These problems all involve either an excess of information or harmful information in parts of the learning system. These problems are going to be more significant as more and more realistic (and complex) domains are tackled by learning systems. There have been attempts to solve some of the problems. For example, Carbonell & Gill (1987), Mitchell, Keller & Kedar-Cabelli (1983), Ruff & Dietrich (1989), and Scott & Markovitch (1989a; 1989b) all contributed to the study of excessive potential experience. Gennari (1989), Iba (1989), and Minton (1988a) contributed to the study of unfocused attention. Samuel (1963) contributed to the study of excessive knowledge, Etzioni (1988), Iba (1989) and Minton (1988a) contributed to the study of non-monotonic acquisition, and Keller (1987), Markovitch & Scott (1988a; 1988b), Minton (1985; 1988a) and Tambe (1988; 1989) contributed to the study of non-optimized knowledge base.

A limitation of most of these works is that they concentrated on one or two of the given problems, without attempting to view all these problems as instances of a more general framework. A notable exception is the work

done by Minton (1985; 1988a; 1988b) to study the problem of utility. Minton's work on utility was a good start for a unifying framework for the above problems, but it too has some limitations. First, it is too specific, fitting only learners of search control rules. Second, although it offers solutions for some of the problems, it makes no attempt to define a general framework for solving the whole class of problems associated with harmful information. Third, its scope is too narrow - it does not include excessive or harmful experience as part of the problem.

The problem of harmful knowledge has received increased attention recently. In IJCAI 1989, a whole session was devoted to works dealing with the utility problem (Greiner & Likuski, 1989; Markovitch & Scott, 1989d; Mooney, 1989; Tambe & Rosenbloom, 1989).

This thesis studies the problems of harmful and excessive information in learning systems in a general way and has the following three goals:

1. To analyze and understand the problems associated with harmful and excessive information .
2. To set up a framework for eliminating/reducing harmful and excessive information in learning systems.
3. To test the framework in the context of a realistic and useful learning system.

To accomplish these goals we first have to define a scope for the discussion. I basically view learning systems as information processing systems where the information flows from the space of experiences through some attention mechanism through the acquisition program, via the knowledge base to the problem solver (see Figure 1).

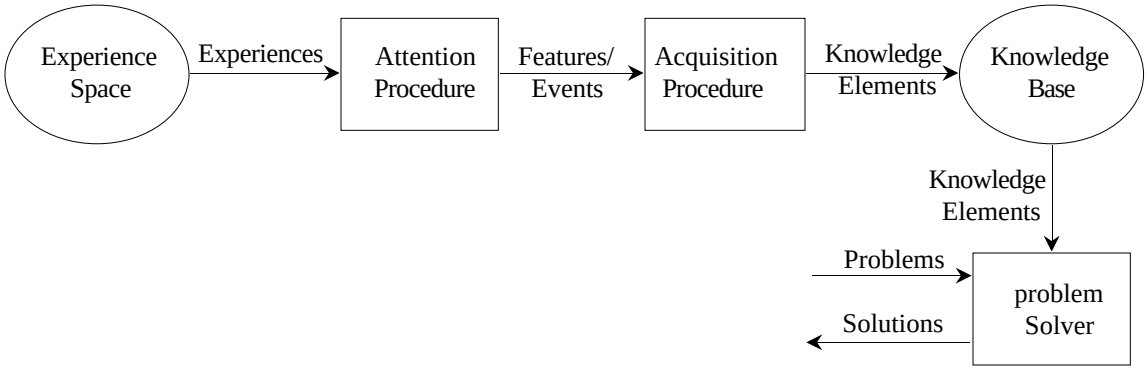

Figure 1. The information flow in a learning system

Problems 1-5 all involve excessive or harmful information in different parts of the information flow. Solving any of the problems requires some selection procedure. Basically, there are two types of selection operations: relative and absolute. Relative selection chooses between two items. Absolute selection decides whether to select an item or not. Both types of selection need some way of evaluating knowledge elements or knowledge sets.

Assuming a learning system that is trying to improve the performance of a problem solver with respect to a given evaluation criterion and a set of tasks, the value of a knowledge item will be defined as the performance with the given knowledge item minus the performance without it (measured by the given criterion). That definition leads to the definition of harmful knowledge - knowledge with negative value.

Given the definition for value of knowledge, it is possible to talk meaningfully about selection processes which try to filter out harmful knowledge, or to select higher valued knowledge out of a set of knowledge elements. The information flow model of learning systems is used to identify five types of selection processes according to their location within the information flow: selective experience, selective attention, selective

acquisition, selective retention and selective utilization. These filters can be fixed, or can use information learned by the main learning procedures, or can employ their own learning procedures. Such learning procedures are called secondary learners, to differentiate them from the main learning procedures (the primary learners).

To achieve the third goal, i.e. testing the framework within the context of a realistic and useful learning system, a learning system called LASSY has been built. LASSY's goal is to improve the efficiency of a Prolog interpreter by exploring the logic database that represents the domain. There were several reasons for the choice of a Prolog interpreter as the problem solver that the learning system tries to improve. First, the semantics of Prolog is clear and well understood (Lloyd, 1984), unlike those of other problem solvers. Second, there is a real need for improvement in the efficiency of Prolog. Prolog has recently been considered a good language for implementing intelligent databases (Brodie & Mattias, 1986; Dahl, 1986; Gallaire & Minker, 1978; Parker, et al., 1986; Sciore & Warren, 1986; Sciore & Warren, 1988; Zaniolo, 1986). However, everybody agrees that the major obstacle to such an integration is the inefficient way in which Prolog processes queries (Kowalski, 1979; Kowalski, 1985; Lloyd, 1984). Third, being a deductive problem solver, Prolog allows an implementation of a deductive learning procedure which utilizes theorems (lemmas) encountered during learning for future problem solving. Deductive learning is of a special interest to this study since deductive learning systems can potentially learn an unbounded number of theorems, making the role of selection more crucial.

LASSY employs two primary learning mechanisms. One is deductive - learning lemmas, and the second is inductive - learning

averages of costs and numbers of solutions, to be used by a procedure that automatically orders the subgoals in a rule's body. LASSY also employs various filters during its learning. The system performs learning by experimentation, i.e. it learns by solving self-generated problems. An experience selector biases the experiences towards those similar to tasks given to the system in the past by external agents. Lemmas are acquired selectively, and removed if proved to be of no value. A utilization filter allows the problem solver to use lemmas only for subgoals that have high enough probability of success, reducing the probability of using lemmas harmfully within failure branches of the tree. Many experiments were performed with LASSY. The results were encouraging, showing a significant improvement in performance.

The rest of this thesis has the following structure: Chapter 2 defines the value of knowledge and discusses types of knowledge that are harmful, and considers how different features of knowledge effect its value. Chapter 3 describes the framework of information filtering. The chapter also evaluates the merit of the framework by trying to apply it to existing learning systems. Chapter 4 describes LASSY architecture using the framework defined in Chapter 3. Chapter 5 describes LASSY's deductive component and the selection processes it employs. Chapter 6 describes the inductive component and the selection processes that it employs. Chapter 7 discusses future work and Chapter 8 concludes. The appendix contains a detailed account of POST-Prolog, the extension of Prolog that was created to allow the implementation of LASSY.

2. KNOWLEDGE EVALUATION BY LEARNING SYSTEMS

How can knowledge be harmful? To understand this question we first need to agree on how knowledge should be evaluated. This chapter will start by defining the value of knowledge. The need for such a definition in the context of the information filtering framework is obvious: if there is a need to select between alternatives, there must be an agreed-upon way of deciding what alternatives are better than others. If all the selection processes were relative, then we could be satisfied with a weaker definition of partial order over elements of knowledge (so that knowledge elements can be compared). However, absolute selection needs the stronger definition - for evaluating one element of knowledge and for deciding whether it is harmful.

Since the common approach to learning has mostly been non-selective, there have been little need to evaluate knowledge. The only type of knowledge that has been recognized as undesirable is incorrect knowledge which is well defined in the context of logic based representations (Genesereth & Nilsson, 1988). The few works that considered correct knowledge as potentially harmful and removed it, needed some evaluation criteria to decide which knowledge elements to remove. Most of these systems used ad hoc heuristics for the evaluation, but failed to provide a definition for the actual value that the heuristics try to estimate.

For example, Samuel (1963) assigned *age*[1] to each of the board configurations memorized by his rote learner. Whenever a board configuration was used, its age was multiplied by 0.5. At fixed intervals of time, the ages of all board configurations was incremented by 1. When space was needed for new configurations, old board configurations (i.e. those with large age) were removed from the system's memory. This heuristic for evaluating board configurations considers boards that are used more often and more recently as more valuable.

Another example of a heuristic used for evaluating knowledge is the *strength* of rules in Holland's classifier system (Holland, 1986). The strength of a classifier is increased if it caused the firing of another rule. The increase in strength is proportional to the strength of the fired rule. Rules may also increase their strength if they receive payoff directly from the environment. Thus, rules are considered more valuable if they ultimately cause the firing of rules that attain goals that directly lead to payoff by the environment. Whenever the system needs more space for newly generated classifiers, those with the lowest strength are removed.

MetaLEX's (Keller, 1987) tries to eliminate (replace by true or false) subexpressions in the USEFUL concept that are harmful (the system performance would improve according to its given criteria by removing them). To do so its must estimate the value of subexpressions. The value is computed by a heuristic function that combines the estimated benefits (the time spent on evaluating the subexpression) and the estimated costs (the frequency of true evaluations for falsifying and the frequency of false evaluations for truifying).

---

1 Cf paging algorithms.

During the last two years there have been three attempts to give a more general definition for the value of knowledge. A definition of *computational irrelevance* was given by Subramanian and Genesereth (1987) as part of their work on defining irrelevancy:

> *f* is computationally irrelevant to *g* if we can prove *g* in theory *M* without the use of *f*, and also show that the proof has better complexity characteristics (it might be shorter or be easier to find)

Although the definition was given with a different goal in mind (than defining the value of knowledge) it can be easily converted to such. Assuming those complexity characteristics are measurable, the value of *f* can be defined as the complexity of the proof using f minus the complexity of the proof without using f. The definitions that will be given in this chapter are indeed similar to that implicit definition of the value of knowledge.

Another definition for the value of knowledge is a part of the work done by Minton (Minton, 1988a) and is given as a definition for the utility of a control rule:

Utility=Average-search-time-without-rule - Average-search-time-with-rule

The problem with Minton's definition is that it is not precise enough and not general enough. Specifically, average is not well defined, and search time is too a restricted criterion. Frequently other evaluation criteria are needed. Also, the definition does not take into account what problem or problems are being solved and how the utility of sets of rules should be evaluated.

Minton's definition is similar to the one given in (Markovitch & Scott, 1988b). There, the value of an item of knowledge is defined as the expectation value for the difference between the cost of solving a problem

with the item of knowledge and solving it without that item. In the next section a more precise version of that definition will be given.

2.1. Defining the value of knowledge

The definition of harmful knowledge will be brought up in several stages. for the sake of simplicity, the first definition applies to the case of a single knowledge element and a single problem. The definitions that follow apply to cases of sets of knowledge elements and sets of problems.

Most learning systems generate or modify a body of knowledge as a result of their learning activity (Michalski, 1986; Scott, 1983). The ultimate goal of such a learning program is to increase the value of the knowledge base with respect to some criterion. Such a criterion is often implicit and embedded within the system architecture. Most learning programs attempt to improve the performance of a problem solver. Such systems evaluate the learned knowledge by evaluating the performance of the problem solver.

The definition of value of knowledge given in this section assumes the following assumptions:

1. The knowledge base is a set. The nature of the elements of the set is not important. It can be a set of rules, of axioms, of classes, etc.
2. There exists a problem solver that uses the knowledge in the knowledge base.
3. There exists an evaluation criterion for the performance of the problem solver.

Given such assumptions, the value of a knowledge element with respect to a specific problem, and a specific knowledge base can be defined

as the value of the performance with the knowledge element minus the value of the performance without it.

Definition 2.1 : Let P be a set of problems. Let S be a set of solutions. Let K be a set of knowledge elements. Let F: $P \times 2^K \rightarrow S$ be a problem solver (a function that maps a given problem and a given knowledge base to a solution). Let E: $P \times S \rightarrow$ REAL be an evaluation criterion that evaluates a solution s of a problem p. Let $p \in P$ be a problem and $k \subseteq K$ a set of knowledge elements. The value of a knowledge element $k_i \in k$ with respect to p and k is a function V: $K \times P \times 2^K \rightarrow$ REAL defined as:

$$V(k_i,p,k) = E(p,F(p,k)) - E(p,F(p,k - \{k_i\})) \qquad [2.1]$$

Many evaluation functions take into account the computation resources (memory and time) that were consumed during the problem solving. To account for such cases we can expand the definition of S to include such quantities together with the actual solution.

An important feature of definition [2.1] which is missing in other definitions is that it specifies explicitly the dependency between the value of a knowledge element and the rest of the available knowledge, and the dependency between the value of knowledge and the specific problem solved. It is interesting to note that Minton's definition (specified above) is a special case of [2.1]. The set of knowledge elements is the set of control rules and the evaluation criterion measures the search time.

Given the definition for the value of knowledge, it is easy to define harmful knowledge.

Definition 2.2 : Given a set of knowledge elements $k \subseteq K$, a knowledge element $k_i \in k$ is *harmful* with respect to a problem $p \in P$ iff

$$V(k_i,p,k) < 0 \qquad [2.2]$$

Intuitively, definition 2.2 says that a knowledge element is harmful if we would have done better without it than with it.

Definition 2.1 applies to single knowledge elements. Often there are dependencies between knowledge elements in the knowledge base. For such cases we need to be able to evaluate sets of knowledge elements.

Definition 2.3 : Let P, K, E, k and p be defined as above. The value of a set of knowledge elements $k' \subseteq k$ with respect to p and k is a function

$V: 2^K \times P \times 2^K \rightarrow REAL$ defined as:

$$V(k',p,k) = E(p,F(p,k)) - E(p,F(p,k - k')) \quad [2.3]$$

Often we need to evaluate knowledge with respect to a set of problems $P' \subseteq P$. To define the value of knowledge with respect to a set of problems we first need to assume the existence of $E': 2^P \times S \rightarrow REAL$, an extension of E, that evaluates the performance of the problem solver with respect to a set of problems. Typically E' is based on E. For example, it is common to evaluate the performance of a problem solver with respect to a set of problems by averaging its performance for every problem. On the other hand, it is also common to find systems that use different criteria, such as maximization of the minimum value of E. Given such a criterion E', definition 2.3 can be generalized to account for sets of problems.

Definition 2.4 : Let P, K, k and k' be defined as above. Let

$E': 2^{P \times S} \rightarrow REAL$ be an evaluation criterion that evaluates the set of solutions for a set of problems. The value of a set of knowledge elements $k' \subseteq k$ with respect to a set of problems $P' \subseteq P$ is a function

$V: 2^K \times 2^P \times 2^K \rightarrow REAL$ is defined as:

$$V(k',P',k) = E'\left(\left\{\langle p,F(p,k)\rangle \mid p \in P'\right\}\right) - E'\left(\left\{\langle p,F(p,k - k')\rangle \mid p \in P'\right\}\right)$$

Sometimes the domain of the problem solver is specified as a subset of P together with the associated probability distribution. If that distribution is

fixed we can define the value of a knowledge set (regardless of a specific problem) as the expected value of V for a problem p randomly drawn from the task space.

The reader should be aware that although the definition given for the value of knowledge is a functional one, it is not proposed as a useful algorithm for evaluating knowledge - the complexity of such an algorithm could be prohibitively high. The definition will let us talk meaningfully about selecting knowledge based on its value. It will also help design heuristics that estimate the value of knowledge sets.

2.2. Evaluating experiences

Learning systems do not necessarily have to be selective *after* they have generated knowledge. They can be selective about what types of experiences they learn from. To do so, a learning system should evaluate the potential experiences it may have. Assuming that the ultimate goal of the learning system is to maximize the value of the knowledge base, the value of an experience can be defined in terms of the knowledge that the learning system generates as a result of processing that experience.

Assuming that there exists an acquisition program that gets as an input an experience and produces a set of knowledge elements, the value of an experience can be defined as the value of the knowledge set that it produces.

Definition 2.5 : Let P, K, k, P' and V be defined as above. Let I be the set of experiences that the learning program faces. Let A: $I \rightarrow 2^K$ be an acquisition program. V': $I \times 2^P \times 2^K \rightarrow$ REAL is defined as:

$$V'(i, P', k) = V(A(i),P',K) \qquad [2.5]$$

Having defined the value of knowledge and the value of experience, we can talk about the value of information and harmful information.

2.3. Costs and benefits of knowledge

Section 2.1 defines harmful knowledge but does not give an account of what causes knowledge to be harmful. This section identifies some features of knowledge that can make it harmful. Each knowledge element has a certain cost associated with retaining it in the knowledge base and using it. Each knowledge element also carries a potential benefit for problem solving. Knowledge turned to be harmful if the costs associated with it are larger than the benefits.

2.3.1. Costs of retaining and using knowledge

The costs of retaining and using knowledge are expressed as part of the value formula 2.4. This section looks at three specific measurements that are typically part of the evaluation criteria for problem solvers.

Memory

The most obvious cost of retaining knowledge is the cost of the memory space used to store it. Memory space may not be a problem for a modern computer with a large virtual memory. On the other hand, the use of a large virtual memory space can cost in access time. In any case, memory is finite, thus if there is more knowledge to store than available memory, the cost associated with memory space can be high. Earlier works in machine learning had stricter memory limitations. For example, space limitations forced Samuel (1963) to remove board positions to allow the storing of new learned board positions.

Search time

When discussing search time, it is helpful to distinguish between two types of knowledge: search knowledge and control knowledge. Search knowledge is any knowledge that changes the topology of the search space, i.e., that add edges between states. Control knowledge is any knowledge that is used by the problem solver to choose among alternatives (either which operator to apply or which state to expand). There are four factors whereby additional knowledge increases cost:

- *Increase in decision time due*: Additional control knowledge can increase the complexity of the decision procedure used by the problem solver to select operators. The total cost is the product of the increase in decision time and the number of times that a decision procedure is applied, which is proportional to the number of operators that are selected by the problem solver. These costs result in the negative utility in Minton's (1988a) program, where the additional control rules increased the cost of matching during decision time.
- *Increase in search time due to different decisions:* Additional control knowledge can also change the decisions made by the problem solver. The new decisions can cost the problem solver in search time. Of course, perfect control knowledge does not carry such costs, since it always generates the right decisions; However, it is very common to have control knowledge which generates the right decision in some cases and wrong decisions in others.
- *Increase in matching time:* Additional search knowledge can increase the complexity of operators, or can increase the number of operators. In both cases the time spent in expanding a state will increase. This cost is

different than the first one (increase in decision time) since it is independent of the decision procedure. If the number of operators is increased, the problem solver has to spend more time finding out which operators are applicable (before even initiating the decision process). Most of the macro learning programs that identified harmful knowledge have considered matching cost as the main factor in causing the problem (Fikes, Hart, & Nilsson, 1972; Minton, 1988a; Tambe & Newell, 1988).

- *Increase in branching factor:* If the number of applicable operators is increased, then the branching factor of the search tree is increased. Increasing the branching factor of a search tree can lead to an exponential growth in search time. Recent works have recognized the increase in branching factor as a very significant factor in causing knowledge to be harmful due to its exponential nature (Iba, 1989; Markovitch & Scott, 1988a; Markovitch & Scott, 1988b; Shrager, 1989).

Quality of solutions

Both additional search space knowledge and additional control knowledge can make the problem solver change the order in which it performs searches. This can cost in finding solutions which are of lower quality. If, for example, the length of the solution is part of the evaluation criteria, and if a new operator causes the problem solver to take a longer path to get to the final state, then the operator costs the problem solver in quality of solution. In the extreme case, the additional knowledge can take the problem solver through an infinite (or practically infinite) branch of the search tree.

2.3.2. Benefits of knowledge

Adding new knowledge carries potential benefits in terms of the value formula [2.4]. This section takes the same three measurements and examines how these three factors can increase the benefit of a knowledge element.

Memory

Adding a new knowledge element can not directly reduce the amount of memory used. However, it is possible that a new knowledge element is a generalization of a set of existing knowledge elements. If the system detects such a relationship and deletes the instances of the general element, then we can say that adding the new knowledge element (indirectly) benefited the system in terms of memory.

Search time

The main reason for acquiring control knowledge is to reduce search time. Adding control knowledge can make the problem solver make a better decision when it selects an operator to apply or a state to expand.

Adding search knowledge can benefit the problem solver in providing faster paths to existing states, or in making accessible states that were previously inaccessible.

Quality of solutions

Additional control knowledge or search knowledge can lead the problem solver to shorter solutions which can have higher value if the length of solution is a factor in the evaluation criteria. In the extreme case, it can lead to new solutions it could not previously achieve.

2.4. Features of knowledge that can cause harmfulness

There are three types of knowledge that are typically associated with harmfulness: incorrect knowledge, redundant knowledge and irrelevant knowledge.

2.4.1. Incorrectness

Incorrect knowledge is often harmful. Search space knowledge is considered incorrect if it allows transitions between states that are not connected (i.e., there is no finite sequence of transition that leads from the first state to the other). Control knowledge is considered incorrect if it causes the problem solver to make a suboptimal decision.

In practice, incorrect knowledge can often contribute to successful performance. If, for example, the system acquires an operator which is overgeneralized, but is correct in most cases, it is quite possible that the system performance with respect to a set of problems will be improved. The same is true for control knowledge. A suboptimal decision can still lead to a great increase in performance.

More generally, an approximate model of the world is necessarily incorrect but may well be better than no model at all. Furthermore, different types of performance error, such as 'misses' and 'false alarms' in a detection task, typically have different associated costs, and an erroneous piece of knowledge which biases the system towards the cheaper type of error may be beneficial. The commonest type of beneficial erroneous knowledge is the inaccurate generalization. For example, the incorrect because over-generalized belief that all grizzly bears are man eaters in all circumstances is certainly better than no knowledge of the possibility that bears can be dangerous (Markovitch & Scott, 1988a).

2.4.2. Irrelevancy

Search space knowledge can be defined as irrelevant with respect to a problem p if there is no solution to p that has that knowledge as a part of its path. Irrelevant knowledge according to the above definition can be useful: It can help the problem solver detect faster that it is on a wrong path. A more restrictive definition will consider search space knowledge k as irrelevant with respect to a problem p (consists of an initial state i and a final state f) if k is not in the transitive closure of i. Irrelevant knowledge under the last definition is clearly harmful (assuming that retaining it costs some non-zero amount).

Control knowledge can be defined as harmful with respect to a problem p=<i,f> if it is never applicable in the transitive closure of i. If retaining such knowledge costs some non-zero amount than it is harmful.

2.4.3. Redundancy

Search space knowledge k is redundant with respect to a problem p if there is at least one solution that does not include k in its path. Unlike irrelevant knowledge, redundant knowledge can be a part of a solution path, thus it is potentially useful, depending on the alternative paths. All learning programs that use deductive processes to acquire knowledge introduce an intentional redundancy in order to improve their performance.

The definitions given above for irrelevancy and redundancy are similar to those given by Subramanian and Genesereth (1987), except that they use logic notation while here a problem solving notation was used. Subramanian and Genesereth's definition for weak irrelevancy says that a fact f is irrelevant to a fact g within theory M if there exists at least one proof of g in M that does not use f essentially. This definition is similar to the

above definition for redundancy. Subramanian and Genesereth define f to be computationally irrelevant to g if there exits a proof of g that does not use f, and has better "complexity characteristics." That definition is similar to our definition of harmful knowledge, except that the definition for harmful knowledge is more general: It defines the "complexity characteristics" as a given evaluation criteria and it generalizes for the cases of a set of problems and a set of knowledge elements (which is analogous to defining irrelevance of a set of facts F to a set of facts G in Subramanian and Genesereth's terms).

2.5. The economics of learning and problem solving

Learning and problem solving can be viewed from an economic perspective. A system which learns, invests a certain amount of resources in order to acquire and maintain some knowledge. To make this investment worthwhile, the return on it must be positive, i.e., the resource saving provided by the learned knowledge must be at least as large as the cost of acquiring and maintaining that knowledge.

The learning process can be executed on-line or off-line. On-line learning takes place during problem solving time, making the learned knowledge readily available to the problem solver. The problem with such an approach is that learning time is measured in the same currency as problem solving time. Thus the learning process must be extremely conservative in its use of time.

The alternative approach is off-line learning as a separate process, independent of the problem solving process. Being two separate processes, learning time and problem solving time are measured in different

currencies. For example, it may be perfectly acceptable to have the learning process run a whole day just to save one second of problem solving time. The disadvantage of this approach is that there is a delay between the time that knowledge is acquired and the time that it is available for use by the problem solver.

3. SELECTION PROCESSES IN LEARNING SYSTEMS

This chapter describes the information filtering model introduced by Markovitch and Scott (1989b)). There are two situations in which learning systems should be selective. The first is when an information element (knowledge or experience) is harmful. If the system considers an information element to be harmful it should eliminate it. The second situation requiring selection is when resources are limited. Limited memory or time resources will force the system to select between alternative knowledge elements or alternative experiences. Selection processes that eliminate harmful information will be called *filters,* since they allow only positive valued information to pass through.

3.1. Types of information filters

Information selectors can be classified according to their location within the information flow of the learning system (Figure 1). There are five different locations where an information selector can be placed. A *selective experience* process filters the set of experiences that the learning system faces. A *selective attention* process filters the set of events or features that the system processes out of a particular experience. A *selective acquisition* process filters knowledge after it is generated by the acquisition procedure and before it is entered into the knowledge base. If the knowledge base is connected to the input of a filter, and the output of

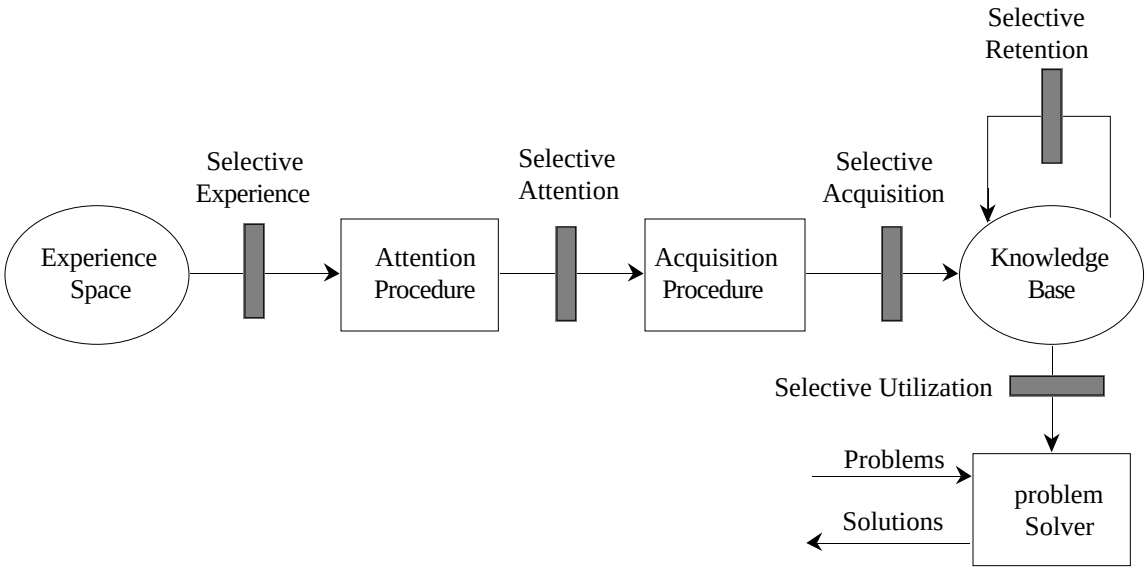

Figure 2. The five logical positions that information filters can be placed at.

the filter is connected to the same knowledge base, it is termed *selective retention* (or forgetting). If the knowledge is filtered between the knowledge base and the problem solver, the filter is called *selective utilization*. Figure 2 illustrates the five types of information filters.

Selective acquisition, selective retention and selective utilization all process knowledge elements (or subsets). Thus these three filters will be called *knowledge filters* while the other two will be called *data filters*. This distinction is closely related to those stated in (Scott & Markovitch, 1989a; Shalin, Wisniewski, Levi, & Scott, 1987) which view learning as two distinct but interrelated search processes: one in the space of possible representations (Mitchell, 1979; Simon & Lea, 1974), and one in the space of the possible experiences (Scott & Vogt, 1983). The next sections will give a detailed account of each of the five filter types.

3.2. Selective experience

The experiences that the learning program faces can be generated by some external source or by the program itself. Selective experience is a selection process that is inserted at the output of the experience generator. When the program generates its own experiences, the experience selector can be incorporated into the generator. When an external source supplies the experiences, the program can decide not to process some of the experiences, or to queue less valued experiences for later use.

One major function of an experience selector is to allocate learning resources intelligently by ordering the experiences that the learner faces in some kind of agenda such that experiences that are known or estimated to have higher value will be processed first.

When the experience selector allows only a subset of the experiences to pass through, it is called an experience filter. Experience filters are used for eliminating harmful experiences (experiences that lead to harmful knowledge), and are particularly useful for reducing the amount of irrelevant knowledge that the system acquires. Such filters will try to estimate what types of experience are likely to make the learning system acquire useful knowledge.

Since the learner evaluates the information it processes according to how it will benefit the problem solver, the closer (from the data flow perspective) the selector is to the problem solver, the more informed it is. Evaluating experience is especially difficult, because the program typically does not know in advance what knowledge will be generated out of a particular experience (otherwise no learning would be necessary).

LEX (Mitchell, et al., 1983) was one of the earlier works that employed a procedure to selectively generate its own experience. It uses two strategies for generating new problems. The first considers the current internal representation in order to generate problems that will allow refinement of existing, partially-learned heuristics. This is done by selecting a term from the maximally-specific subset of the version space, selecting a member of the maximally-general subset of the version space, and finding a sibling of a term of the specific member which is more specific than the corresponding term of the general one. This term is substituted into the specific member of the version space. The idea is to generate a problem which is different from the problems encountered in the past, so that the heuristics will be refined, yet similar enough to the most recently encountered positive instance (so that it is likely to be solvable).

The second heuristic that is used by the problem generator in LEX intersects preconditions of operators. If two operators can be applied in a certain state then the system needs heuristics to decide between the two. The problem generator therefore intersects the preconditions of the two operators and instantiates the intersection to produce a specific problem. Solving this problem will result in proposing a new heuristic to decide between the two operators.

To summarize, LEX uses its knowledge of its internal state to generate problems that are likely to improve its current representation (according to its evaluation criteria). The two strategies that LEX employs are specific to the version space algorithm used for the learning.

Another program that selectively generates its own experiences is DIDO (Scott & Markovitch, 1989a). DIDO is an inductive learning program whose goal is to reach a state in which it can predict the outcomes of

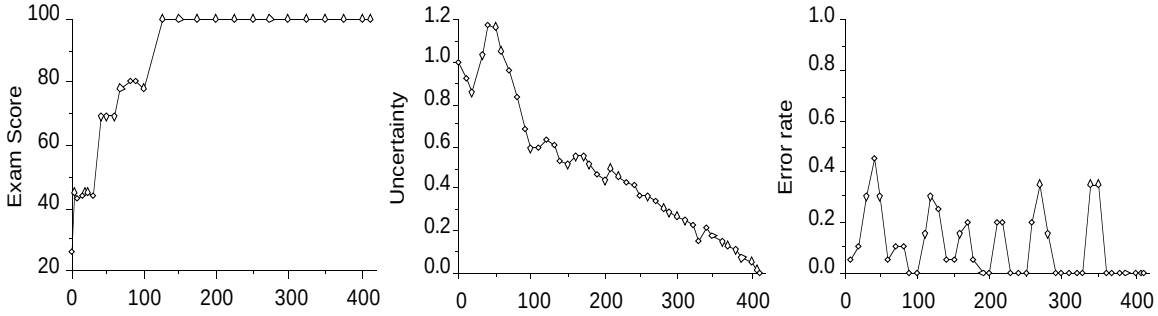

Figure 3. The learning curve of DIDO as determined by exam scores, the average uncertainty in the system during learning, the error rate during learning.

applying operations to all entities encountered in its domain. DIDO selects its own experiences by deciding which operation to apply on what object. The strategy used by DIDO is based on the uncertainty function by Shannon (1949) which is defined as:

$$H = -\sum_{i=1}^{n}(p_i \times \log_2(p_i))$$

where the sum is taken over all alternative outcomes. At any stage of the learning process, there are operations and objects for which DIDO is not sure what the outcome of applying an operation on an object will be. DIDO associates a probability with each alternative outcome and the Shannon formula is used to measure the system's uncertainty regarding the outcome of applying the operation on objects belonging to the specified class.

Figure 3 shows the results of a typical experiment with DIDO. The left hand graph shows how well DIDO performed in solving the "exams" given to it. An exam is a set of 100 randomly selected pairs of object-operation for which DIDO is required to predict the outcome of applying the operation to the object. Since the exam graph reaches a score of 100 after 120 training examples, one might argue that the other examples are useless (they do not improve system performance), and the selection

algorithm is not efficient. However, to evaluate whether an experience is useful or not, the evaluation criteria for the problem solver needs to be taken into account (see Section 3.5). DIDO's real goal is not only to be able to predict correctly, but to do so with as high certainty as possible. The middle graph shows that according to that evaluation criterion, the system kept on improving with more experiences, so the selector is indeed effective. The right hand graph shows the proportion of the selected experiences that DIDO predicts incorrectly. The error rate does not decline until the last stage of the learning, proving that the selection procedure indeed selects examples that are informative.

Another recent system that uses selective experience is ID5R (Utgoff, 1989). The ID5R tree induction algorithm is an incremental version of ID3 (Quinlan, 1986). ID5R employs an experience filter which accepts only instances which would be misclassified by the current decision tree. The algorithm produces smaller trees (than those produced by ID3 on the same data set), but the reduction does not seem to be significant. A more interesting comparison between the two would be between the number of learning instances that were used to produce the tree, or a different measurement of the learning resources used, but unfortunately such comparisons were not given.

Two works in the 1989 Workshop on Machine Learning specifically address the problem of selective experience. One work, by Scott and Markovitch (1989b), argues that a learning system has an advantage over an external teacher in selecting learning instances because its ability to directly access its own representations (which an external teacher can not). The other work, by Ruff and Dietrich (1989), studies 5 strategies for selecting experiments for a classification program. The work confirms the

hypothesis that selective experience is more effective than non-selective experience. The best strategy was one that selects experiments that most nearly split the hypothesis space in half. An almost as good, and much cheaper strategy was the one that selects any experiment guaranteeing the elimination of at least one hypothesis.

3.3. Selective attention

When the learning program receives an experience, an attention filter can filter out features of the particular experience that are likely to lead to harmful knowledge. The distinction between selective experience and selective attention is not always clear. An experience is the basic information unit as defined by the experience generator. Typically, the acquisition program prefers different information units; thus it incorporates a mechanism to transform the experience into a set of features or events that it can process. The attention selector filters or orders these features or events.

An example of selective attention is the OBSERVER module of the PRODIGY system (Minton, 1988a). The experience that the system receives is a trace of the problem solver. The OBSERVER scans the tree and identifies any instances of the target concepts. This is the system's attention mechanism: transforming the experience into units that are needed by the acquisition program. When an example is identified, PRODIGY filters out uninteresting examples by using example-selection heuristics. The heuristics are based on the expected utility of the rules that would be produced as a result of processing the example. The heuristics are not specified, but one example that is given is a heuristic that considers

an example of the target concept OPERATOR-SUCCEED interesting only if the problem solver first tried another operator and failed at the given node.

Another system that employs selective attention is MACLEARN (Iba, 1989). The experiences that the system faces are again traces of the search tree. Instead of looking at all possible macros (sequences of states), the system attends only to sequences that are between states whose values are peaks in the evaluation function relative to a path in the search tree.

Yet another example of selective attention is the program CLASSIT-2 by Gennari (1989). CLASSIT-2 stores the relative "salience" of each attribute, which is defined as the score each attribute receives from the evaluation function. The program ignores attributes with smaller values of salience.

3.4. Selective acquisition

The output of the acquisition program is knowledge that is stored in the knowledge base to be used later by the problem solver. A knowledge filter can be inserted between the output of the acquisition program and the knowledge base in order to filter out knowledge elements that are estimated to be harmful.

Selective acquisition is the first type of selector that processes knowledge elements, i.e., information in the form that will be used by the problem solver. Thus, acquisition selectors are more informed than experience or attention selectors. However, it is still a difficult task to assess the value of a knowledge element before it is used during problem solving. Very often the usefulness of a knowledge element can be estimated

only *after* the system accumulates some experience by using the knowledge.

Another major problem with selective acquisition is that it is basically a hill-climbing approach. If an element g is to be added to knowledge base K, an acquisition filter will consider only two alternatives: K and $K \cup \{g\}$. An acquisition filter can not account for the case where there is $K' \subset K$ such that $\forall Y \subseteq (K \cup \{g\})[V(K' \cup \{g\}) > V(Y)]$. This is the case where a subset of the existing knowledge base should be removed after the acquisition of the new knowledge element in order to make the knowledge base optimal. The same argument can be applied to the case where the system acquire more than one element at a time.

An example of a system which employs selective acquisition is PRODIGY which estimates the utility of the control rules immediately after their generation and filters out rules with negative estimated utility. The utility estimate is based on a single empirical measurement of the rule's cost and benefit. The saving that the rule carries is calculated by measuring the size of the tree that would have been eliminated by the new generated control rule. This is similar to the estimate that is used by the LASSY system and will be discussed in Chapter 5.

Another example for selective acquisition is MACLEARN (Iba, 1989), which employs a *static filter* to discard macros before they are stored in the system's knowledge base. The filter eliminates macros which have large expanded length (the claim is that macros with larger expanded length tend to have more preconditions and are thus less often applicable).

Yamada and Saburo (1989) developed an interesting heuristic for assessing the value of a macro in order to decide whether to acquire it. The method is called *perfect causality*. A sequence of operator satisfies perfect

causality if it causes applicability of an operator that would not be otherwise applicable (would not be applicable in the initial state).

Aha and Kibler (1989) developed the NTGrowth algorithm which performs instance based learning. The algorithm employs selective acquisition: it acquires only instances that were classified wrongly. In the next sections other filters employed in NTGrowth will be described.

A different type of acquisition filter is the hypothesis filtering described in (Etzioni, 1988). This example is radically different from the others given here. The filter is much more sophisticated and guarantees that all knowledge elements that pass through it have value above a given threshold. The filter is a general purpose procedure which can be inserted at the output of any concept learning procedure. The filter performs a statistical test and releases only hypotheses that pass the test. These hypotheses are guaranteed to be accurate and reliable to an arbitrary degree, i.e. it turns the learner into a PAC (Probably Approximately Correct) learner (Valiant, 1984).

3.5. Selective retention

Selective retention (also referred to as *forgetting*) is a filtering process in which the system selects a subset of its knowledge base and replaces the knowledge base with this subset. Basically, a retention filter is looking for $K' \subseteq K$ such that $\forall Y \subseteq K[V(K') \geq V(Y)]$. Since the number of subsets that can be candidates for deletion is exponentially large, heuristics must be employed. Most filters that implement selective retention assume the independence of single knowledge items even though such assumptions are likely to be wrong. Several programs has implemented some variation of

selective retention, and all assumed such independence (Holland, 1986; Iba, 1989; Keller, 1987; Markovitch & Scott, 1988b; Minton, 1988a).

Typically, retention filters base their evaluation of knowledge elements on their contribution during performance. Thus, every knowledge element must be allowed to stay in the knowledge base for a certain period to give it a chance to contribute. This may create a problem when harmful knowledge elements reside in the system's knowledge base, reducing the system's performance.

An important advantage of selective retention over selective acquisition is that it allows the learner to take more risks while acquiring knowledge, knowing that decisions are not final. Instead of deciding whether to acquire a knowledge item without knowing about it sufficiently, the learner acquires it and waits until enough is known about it to decide whether to retain it or not.

A good example of a program that uses selective retention for this purpose is DIDO (Scott & Markovitch, 1989a). DIDO generates hypotheses based on relatively small amount of evidence, based on the assumption that a selective retention filter will remove hypotheses that prove to be incorrect or useless. These hypotheses are called conjectures. Conjectures are retracted if they are incorrect or if they are useless. A conjecture is considered useless if it does not add any information to that stored in its superclasses.

Some of the aspects of selective retention were studied in (Markovitch & Scott, 1988a; Markovitch & Scott, 1988b). A simple macro learner, FUNES was built in order to explore the effects of forgetting on learning. Experiments performed with FUNES brought some interesting results. First, it was shown that the performance of the problem solver can be

improved after some of the learned knowledge is removed. Second, it was shown that even random removal of knowledge can bring improvement in performance. This result looks paradoxical at first, but a further analysis makes the result more rational. After learning many new macros, the system has many alternative paths between points in the search graph. The degree of redundancy introduced by the learner is very high. Thus removing any subset of macros will not harm the system's performance (because alternatives exist) but will reduce the branching factor, thus making the search more efficient.

Several other programs incorporate selective retention. Samuel's checker player (Samuel, 1963) removes board positions that are not used frequently. Holland's genetic algorithm (Holland, 1986) removes classifiers that have low strength. Iba's macro learner (Iba, 1989) employs a dynamic filter which removes macros that are not used by the problem solver. PRODIGY (Minton, 1988a) removes control rules with low utility. NTGrowth (Aha & Kibler, 1989) removes instances that appear to be noisy.

An interesting example of a program that performs selective retention is MetaLEX (Keller, 1987). MetaLEX, unlike the other programs mentioned above, does not acquire the knowledge that it forgets, but receives it as a part of the task specification. The knowledge set is harmful to begin with, and MetaLEX tries to remove elements (subexpressions) that are estimated to be harmful in order to make the knowledge set useful (in MetaLEX terms: tries to make the USEFUL concept operational).

One circumstance where retention filters are not effective is when a knowledge element is very useful in the context of one problem, but very harmful in the context of another problem. In such a case its value will be

close to zero and it will probably get deleted. The only type of filter that can handle such a situation are utilization filters.

3.6. Selective utilization

Selective utilization is a selection process inserted between the problem solver and the knowledge base. The main advantage of utilization filter over the other types of filters is its proximity (from data flow point of view) to the problem solver, which allows it to base its selection on the current problem being solved. The main disadvantage of utilization filters is that they cost in problem solving time rather than learning time, and hence the cost of running the filter must be less than the cost it saves for there to be a net advantage.

Until recently, selective utilization was not recognized as a method for reducing harmfulness of learned knowledge. In the last IJCAI conference (1989) there were three works that proposed using selective utilization. One work is by Mooney (1989). Mooney addresses the utility problem in explanation based learning in the context of his system EGGS, and reaches the same conclusion as this thesis did: that selective use of knowledge can be effective in reducing the harmfulness of learned knowledge. EGGS includes a crude utilization filter: it uses only rules which completely solve the problems. It does not use learned rules to solve subgoals. Surprisingly, this rather simple filter proved to be quite effective in reducing the harmfulness of the learned macros.

The second work presented was by the author (Markovitch & Scott, 1989d) and is part of this thesis. The utilization filter employed by LASSY is described in Chapter 5. LASSY uses a more complex filter than EGGS,

which either uses a macro (lemma) to solve the whole problem or does not use it at all. LASSY uses lemmas in parts of its search tree where lemma usage is less likely to be harmful.

The third work is NTGrowth by Aha and Kibler (Aha & Kibler, 1989). NTGrowth keeps track of the classification performance of its instances. Only *accepted* instances are used for classifying new instances. Instances are acceptable if their classification accuracy is statistically significantly greater than their class' observed frequency.

Every problem solver selects between alternatives and utilizes only part of its knowledge when solving a particular problem. Such a selection is not an example of selective utilization. A selector is a utilization filter if the knowledge that it filters out is not accessible to the problem solver.

One problem that utilization filters may be able to solve and explain is the paradoxical phenomenon whereby humans, whose knowledge base obviously contains inconsistent facts is able to perform deductive inference. This phenomenon can be explained by the existence of a utilization filter that activates consistent subsets of knowledge when solving particular problem.

3.8. Primary learning vs. secondary learning

Information filters by themselves can be viewed as problem solvers, and as such, they require knowledge. The knowledge can come from three different sources: it can be a fixed part of the filter, it can come from the learned knowledge base, or it can be acquired by a different learning process. Thus, it is quite possible that a learning system includes more

than one learning processes. To avoid confusion, we will define two new terms: primary learning and secondary learning.

Definition 3.3.1: Given a problem solver F, a learning process is called a *primary learner* if the knowledge that it generates is used by the problem solver.

Definition 3.3.2: Given a problem solver F, a learning process is called a *secondary learner* if it is not a primary learner, and the knowledge that it generates is used by a primary learner.

The first condition in the definition of a secondary learner accounts for the case when a filter uses knowledge from the main learned knowledge base. Without the first condition, the main learning process would be primary learner and secondary learner at the same time.

Thus, a learning system can be viewed as composed of the main learning process and (possibly) several secondary learning processes. The secondary learners are full learning systems, with the filters being their problem solvers. It is quite possible to have selectors within the information flow of these secondary learners. Therefore, one could talk about an Nth-level learner which generates knowledge to be used by N-1th level learner.

Most of the systems that incorporate selective retention employ a secondary learning process to accumulate statistics about the usage of the knowledge. The Bucket Brigade algorithm by Holland (1986) is a secondary learning process which updates the strength of rules. The strength is used by the primary learner to select rules for the crossover operation and for deleting weak rules from the rule set. The Hypothesis Filtering algorithm (Etzioni, 1988) employs a secondary learning process which take a sample of the population to estimate the distance between the hypothesis and the

target concepts. These estimates are used to filter out hypotheses that were generated by the main learner but proved to be non-PAC.

3.9. Summary

Chapter 3 sets up a framework for selection processes in learning systems. The framework identifies 5 types of selection processes: selective experience, selective attention, selective acquisition, selective retention and selective utilization. The chapter argues that the framework is a useful unifying framework by showing how many existing systems fit nicely into the framework. Table 1 summarizes those examples.

Table 1. Selection mechanism in existing learning systems (part 1)

System	Selection Type	Description	Evaluation	Source
DIDO	Experience (selector)	Performs experiments on classes with high uncertainty	Prefer experiences involving objects of classes with higher uncertainties	uncertainties computed using probabilities which are maintained by the primary learner
	Retention (filter)	Deletes useless hypotheses	Class is useless if it predicts the same as all its superclasses	
LEX (Mitchell, Utgoff, Banergi)	Experience (filter)	The Problem Generator constructs new practice problems	Prefer problems that will refine partially learned heuristics	
	Attention (filter)	The Critic marks positive and negative instances in the search trace	Select search steps on the lowest cost solutions as positive	
PRODIGY (Minton, Carbonell, Gil)	Experience	Experiment Generator generates experiments when discovers incomplete domain knowledge		
	Attention (filter)	The OBSERVER selects training example out of the trace tree.	Training example selection heuristics eliminate "uninteresting" examples	
	Acquisition (filter)	The MONITOR estimates the utility of newly acquired control rules and deletes rules that are highly unlikely to be useful	Eliminate rules whose cost would outweigh it saving even if it was always applicable	
	Retention (filter)	Empirical Utility validation by keeping the running total of the costs and frequency of application	Estimated accumulated savings minus accumulated match cost. If negative discards rule.	Secondary learner accumulate frequency of use and matching cost.
Holland's Classifier System	Retention (selector)	Removes classifiers with lowest strength.	Prefer rules with higher strength. Strength proportional to generality and freq of use	Strength updated by the bucket brigade algorithm. Used by primary learner to generate new rules.
MACLEARN (Iba)	Attention (filter)	The macro proposer uses a peak-to-peak heuristics	Propose only macros that are between two peaks of the heuristic function.	fixed
	Acquisition (filter)	Static filtering. Only macros estimated to be useful are acquired	Redundancy test (primitive) + limit on length + domain specific test.	Fixed.
	Retention	Dynamic filtering. Invoked manually.	Freq of use in solution.	Secondary learning - accumulate statistics
FUNES	Retention (filter)	Various heuristics to decide what macros to delete	Random, Frequency of use * length	Frequency of use accumulated in a secondary learning process

Table 1. (cont) Selection mechanism in existing learning systems (part 2)

System	Selection Type	Description	Evaluation	Source
CLASSIT-2 (Gennari)	Attention (filter)	Attributes with low salience are ignored	Salience	
ID5R (Utgoff)	Experience (filter)	Trains only on instances that would be misclassified	if misclassified then 1 else 0.	Uses primary knowledge structure (tree)
Learners (Ruff & Dietterich)	Experience (selector)	Studies 5 strategies for experiment selection	• prefer experiment that is guaranteed to eliminate at least one hypothesis. • prefer experiment that most nearly splits the hypothesis space in half	Uses primary knowledge structure (current hypothesis space)
SOAR (Laird et al)	Experience	Only considers experiences that involve impasses.	If involves impasse then 1 else 0	
	Attention (filter)	Only considers working elements used to solve the subgoal.		
Hypothesis Filtering (Etzioni)	Acquisition (filter)	Runs a test on the sample of the population. Passes only hypotheses which are PAC	For a given ε and δ, computes an upper bound on the distance between the hypothesis and the target concept	A secondary learning process takes a sample of the population to estimate the distance between the hypothesis and the target concept
Samuel's Checker Player	Retention	Forgetting mechanism discards boards position least frequently used	Discard board positions which are too old	Keeps age for each board position. Multiply by 0.5 each use. Add 1 each cycle.
EGGS (Mooney)	Utilization	learned macros are used only if they solve the problem.	If solves the problem then 1 else 0.	
PiL2 (Yamada & Tsuji)	Acquisition	Acquires only macros with perfect causality.	If perfect causality then 1 else 0	
NTGrowth (Aha & Kibler)	Acquisition	Acquires only instances that are misclassified	if correctly classified then 0 else 1	
	Retention	Removes instances that appear to be noisy	If the instance accuracy interval's highest value is less than their class frequency interval's lowest then o else 1	A secondary learning process maintains classification records for all instances
	Utilization	Uses for classification tasks only instances that are "accepted"	If the instance accuracy interval's lowest value is greater than their class frequency interval's highest then o else 1	A secondary learning process maintains classification records for all instances
MetaLEX (Keller)	Retention	Removes subexpressions that are estimated to be harmful	A weighted combination of estimated cost and estimated benefit.	Two secondary learning processes. One to accumulate the frequency of false and true during problem solving, another runs the program on the whole set of problems to get an initial estimate

4. IMPLEMENTING INFORMATION FILTERS: THE LASSY SYSTEM

The principles described in chapters 2 and 3 were studied in the context of a machine learning program called LASSY (Learning And Selection SYstem). LASSY is a program that learns domain specific knowledge by gaining experience in solving problems in the given domain. The program utilizes the learned knowledge to increase the efficiency of the Prolog interpreter when solving problems within that domain. This chapter will give an overview of the LASSY system. Chapters 5 and 6 will give a more detailed account of the two main learning components of the system.

4.1. The motivation for combining Prolog with Learning

Logic programming is a paradigm that was intended to make programming more declarative than traditional programming languages. The basic idea of logic programming (Kowalski, 1979; Kowalski, 1985; Lloyd, 1984) is that algorithms consist of two disjoint components: a logic component and a control component. Ideally, logic programming is always declarative, but unfortunately this goal can not be completely realized. Hence both types of logic programming are necessary (Kowalski, 1985).

One particular application of logic programming where the problem of having to specify the control is noticeable is for intelligent databases.

Many researchers have suggested that logic programming languages such as Prolog can be an ideal tool for implementing intelligent databases (Brodie & Mattias, 1986; Dahl, 1986; Gallaire & Minker, 1978; Parker, et al., 1986; Sciore & Warren, 1986; Sciore & Warren, 1988; Zaniolo, 1986). However, the inefficient way in which Prolog processes queries (Kowalski, 1979; Kowalski, 1985; Lloyd, 1984) is a major obstacle to this approach.

It is well known that it is possible to write efficient programs in Prolog, but usually only expert programmers can do so. To write an efficient program one has to understand exactly how Prolog works, which is exactly what logic programming was intended to avoid. The problem is apparent when we want to use Prolog as a database language. For such a use we do not expect users to be expert programmers. Typically, they will be novice users who understand logic and can specify what objects are in a domain, what relationships exists between the objects, and how one can infer new relationships.

Unfortunately, such novice users are likely to encounter a major problem. It is very likely that the queries submitted will require a substantial amount of computing resources. Although the basic idea of logic programming is that the system will supply the control, Prolog does it in a very primitive way, by conducting exhaustive depth first search in very large AND-OR search trees. It is widely recognized within the AI research community that to conduct a search in an efficient manner, a problem solver must use domain specific knowledge (Pearl, 1984). Prolog, in its original form, does not have any facilities for using such knowledge.

There are two possible approaches for supplying the interpreter with the needed control knowledge. One approach is to require the user to supply that knowledge. Several Prolog extensions were built to incorporate

facilities for allowing the user to specify control [Clark, 1979 #34; Clark, 1982 #35; Galliare, 1982 #64; Pereira, 1982 #141; Pereira, 1984 #143; Naish, 1985 #316]. The problem with such an approach is that the language becomes completely procedural, and its advantage over conventional programming languages is no longer apparent .

An alternative approach is to try and build a learning program that will acquire the needed knowledge without external help. The next section will describe a system named LASSY which takes this approach and learns domain specific knowledge by performing self generated tasks in the given domain.

4.2. The architecture of LASSY

LASSY is a system that learns domains represented by Prolog databases, and exploits the learned domain specific knowledge to accelerate the search process of the Prolog interpreter. The interpreter used is for the POST-Prolog language - an extension of Prolog that allows declarative programming (see appendix).

The system has both a deductive learning component and an inductive learning component. The design of the LASSY system assumes that the Prolog interpreter receives tasks (queries) from an external source, and that the main goal of the system is to improve the expected average speed with which the interpreter will execute future tasks. It is also assumed that the system performs off-line learning (i.e. no learning takes place while solving externally supplied problems) and that learning time is "cheaper" than performance time.

LASSY was designed to satisfy two major constraints:

1. Learning should not change the semantics of the given database, i.e., the set of theorems before and after learning should be identical.
2. Learning should be transparent to the user, i.e., the user should not be aware of the existence of the learning system except perhaps by noticing changes in the speed of execution.

LASSY's main learning method is "learning by doing": the system learns while doing tasks of the same type that the performance system does (Prolog queries in our case). We assume that there is no external agent providing training problems to the system; thus the system must be able to generate its own problems. The self-generated tasks must be informative, i.e. they should lead the system to acquire useful knowledge.

LASSY is best described as consisting of two logical layers: the primary learning layer and the secondary learning layer. The primary learning layer consists of the task generator, the primary acquisition procedures (inductive and deductive) and the primary learned knowledge bases (inductive and deductive). The problem solver whose performance the primary learners try to improve is a Prolog interpreter.

The information filters are shared by the first and second layers, but play different roles. From the primary learners' point of view the filters are selector processes which are inserted within the data flow of the primary learning processes. From the secondary learners' point of view, the filters are problem solvers whose performance needs to be improved. The secondary learning layer consists of the various filters, the secondary learning procedures that acquire knowledge to be used by the filters, and the secondary knowledge bases (i.e. knowledge bases acquired by the secondary learners).

The next three sections outline the structure of LASSY from the two layers perspective. Section 4.3 describes the primary learning layer. Section 4.4. describes the selection processes. Section 4.5 describes the secondary learning level.

4.3. Primary learning in LASSY

The POST-Prolog interpreter uses two mechanisms for improving its efficiency, both of which depend on domain specific knowledge. One mechanism is lemma usage, i.e., using substantiated subgoals that were proved in the past as if they were regular axioms. For such usage, no special change is needed in the interpreter. The second mechanism is the subgoal reordering procedure that is built into the problem solver.

Both mechanisms require domain specific knowledge. The lemma usage mechanism requires the lemmas. The ordering procedure requires estimates of costs and number of solutions of subgoals. Thus, there are two learning processes in LASSY, one for acquiring the deductive knowledge (lemmas) and the other for acquiring the inductive knowledge (estimates).

The experiences that both learners use as the basis for their acquisition are the traces of the search done by the Prolog interpreter while proving queries. Thus there is only one source of experiences for the two learners. The experiences are generated by the program itself. The task generator generates queries to be solved by the Prolog interpreter. The interpreter proves the queries and the trace is fed to the learners[1]. The

1 Actually, the learners observe the interpreter while proving, but conceptually it is clearer to look at it as if the trace is passed to the learner . This makes the information flow more apparent.

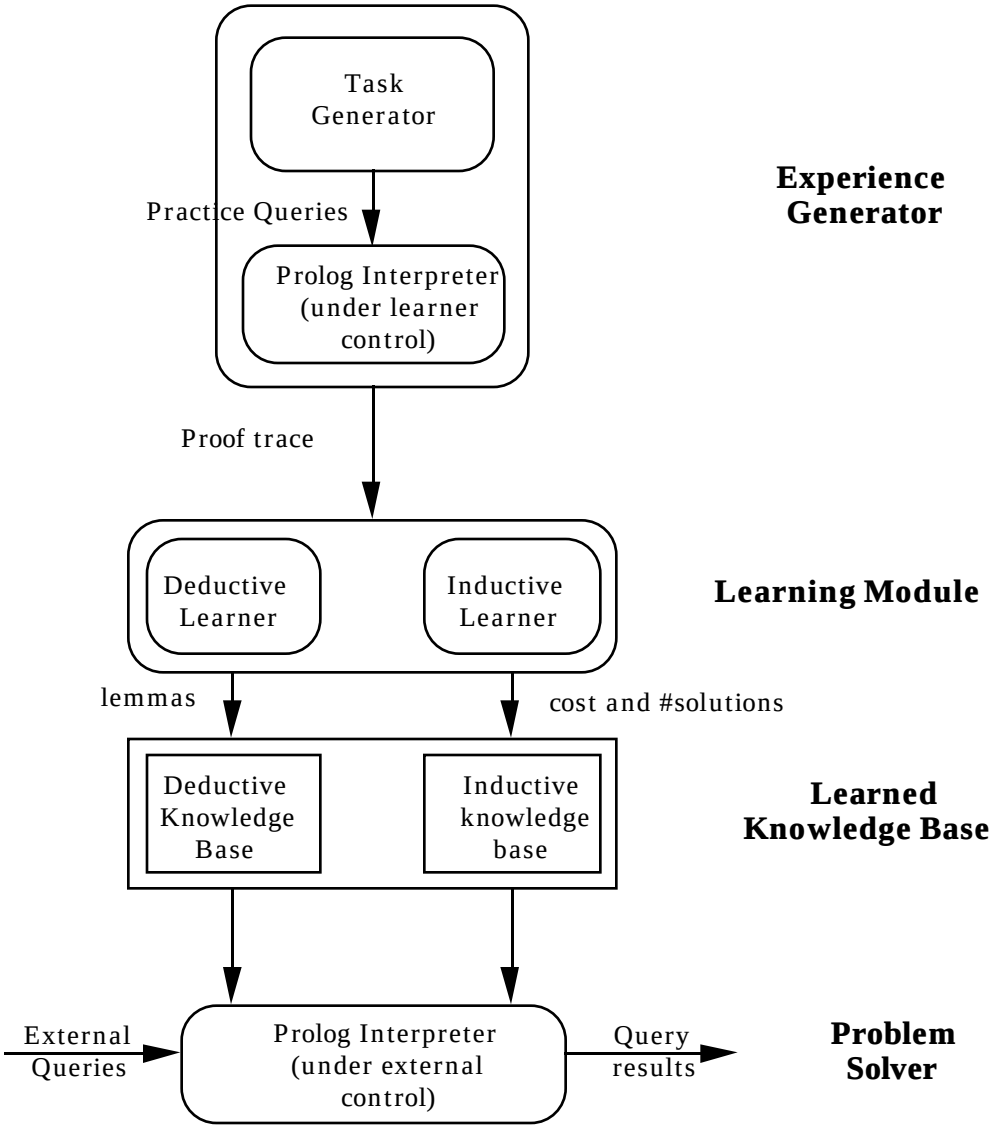

Figure 4. The primary learning layer of the LASSY system

general architecture of the primary learning system is illustrated in Figure 4.

4.3.1. The problem solver

The performance element of LASSY is a POST-Prolog interpreter (Markovitch & Scott, 1989c). POST-Prolog is a simple extension of Prolog that allows the programmer to specify which parts of programs should be executed in a particular order and which parts can be executed in any

order that the system wishes. POST-Prolog is described in appendix A. The interpreter is able to use lemmas as if they were regular axioms. A major component that was added to the POST-Prolog interpreter is a procedure for ordering subgoals which utilizes the inductive knowledge. The ordering procedure is described in Chapter 6.

4.3.2. Prolog proof generation as a search process

LASSY's goal is to make the Prolog interpreter more efficient. For the discussion of the possible ways to achieve that goal, it is beneficial to regard the proving process of Prolog as a search process. The first state space is based on the abstract interpreter for logic programs given in (Sterling & Shapiro, 1986) and quoted in appendix A. The states in that search space are sets of goals that need to be proven. A transition to another state is performed in the following way: one of the subgoals in the sequence is chosen and unified with the head of one of the clauses in the database. The matched subgoal is removed from the list of subgoals to be proved and the (possible empty) set of subgoals in the body of the matched clause is added to the list of subgoals to be proved. The substitution that was produced by the unification is applied to the list of subgoals to be proved. The resulting list defines the new state. The initial state is the list of goals in the query. The final state is the empty set. Table 2 summarizes the state space description.

Note that the procedure given above allows the execution of goals in any order and allows matching with database clauses in any order. Standard Prolog totally restricts the choice by handling all the sets shown in Table 2 as ordered sets. Thus, standard Prolog tries to unify the left most goal of the current state, and the subgoals of the clause with which the current goal was unified are appended at the beginning of the goal list as a

Table 2. The proof process of a logic program described in a search space notation

Given:

1. A logic program

$$P=\{A_1 \leftarrow A_{1_1}\ldots A_{1_{m_1}},\ldots,A_n \leftarrow A_{n_1}\ldots A_{n_{m_n}}\} \quad \text{where } m_j \geq 0$$

2. A goal G

Define SP(P,G) the search space for P and G to be <S,I,F,T> where:

$S = 2^L$ where L is the set of literals. S is the set of states.

I = {G}. I is the initial state.

F = {}. F is the final state.

$T: S \rightarrow 2^S$ defined as follows:

$$T(s) = \{ ((s - \{l\}) \cup \{A_{i_1}\ldots A_{i_{m_i}}\})\theta \mid l \in s,\ A_i \leftarrow A_{i_1}\ldots A_{i_{m_i}} \in P,\ l, A_i \text{ unifies with mgu } \theta\}$$

T is the transition function

sequence. Also, P is an ordered set; thus standard Prolog tries to match the current goal with the clauses in the order in which they appear in P.

POST-Prolog allows more flexibility. It allows reordering of the subgoals in the clause body before they are appended at the beginning of the goal list, and it allows the reordering of P before matching starts. Those orderings change the control of the search but do not change the search space itself. Of the two types of ordering, the current version of LASSY does only the first. In addition, the lemma learner learns shortcuts between states in the search space.

4.3.3. Deductive Learning

Deductive learning is the process of making explicit theorems that are implicitly available to the system. Whenever the Prolog interpreter succeeds in proving a subgoal, the subgoal under the current substitution is recorded in the positive lemma database. Using the state space notation described above, a lemma is a generalized macro which connects each state whose first subgoal matches the lemma to a state that has the same subgoals with the exception that the first subgoal is missing and the rest of

the subgoals are substituted under the matching substitution. A more complete account of the deductive learner in given in Chapter 5.

4.3.4. Inductive Learning

The inductive learners acquire information about the average costs and average number of solutions of subgoals. The averages are collected for the calling patterns (see Chapter 6). Whenever the interpreter concludes the search of a subtree, the cost of the subtree is added to the accumulated cost of the calling pattern for that subgoal, and the calling counter for that calling pattern is incremented by one. The acquired knowledge is used by an ordering procedure to order the subgoals in the body of a rule whose head has just been matched with the current goal. Using the state space notation, the ordering procedure orders the subgoals of the current matched rule before adding them to the goals of the current state in order to create the next state. A detailed description of the inductive learner is given in Chapter 6.

4.4. Selection Processes in LASSY

LASSY incorporates all five types of information filters. In the current version of the system most of the selection processes are done within the information flow of the deductive learning. The architecture of the system with its filters is illustrated in Figure 5.

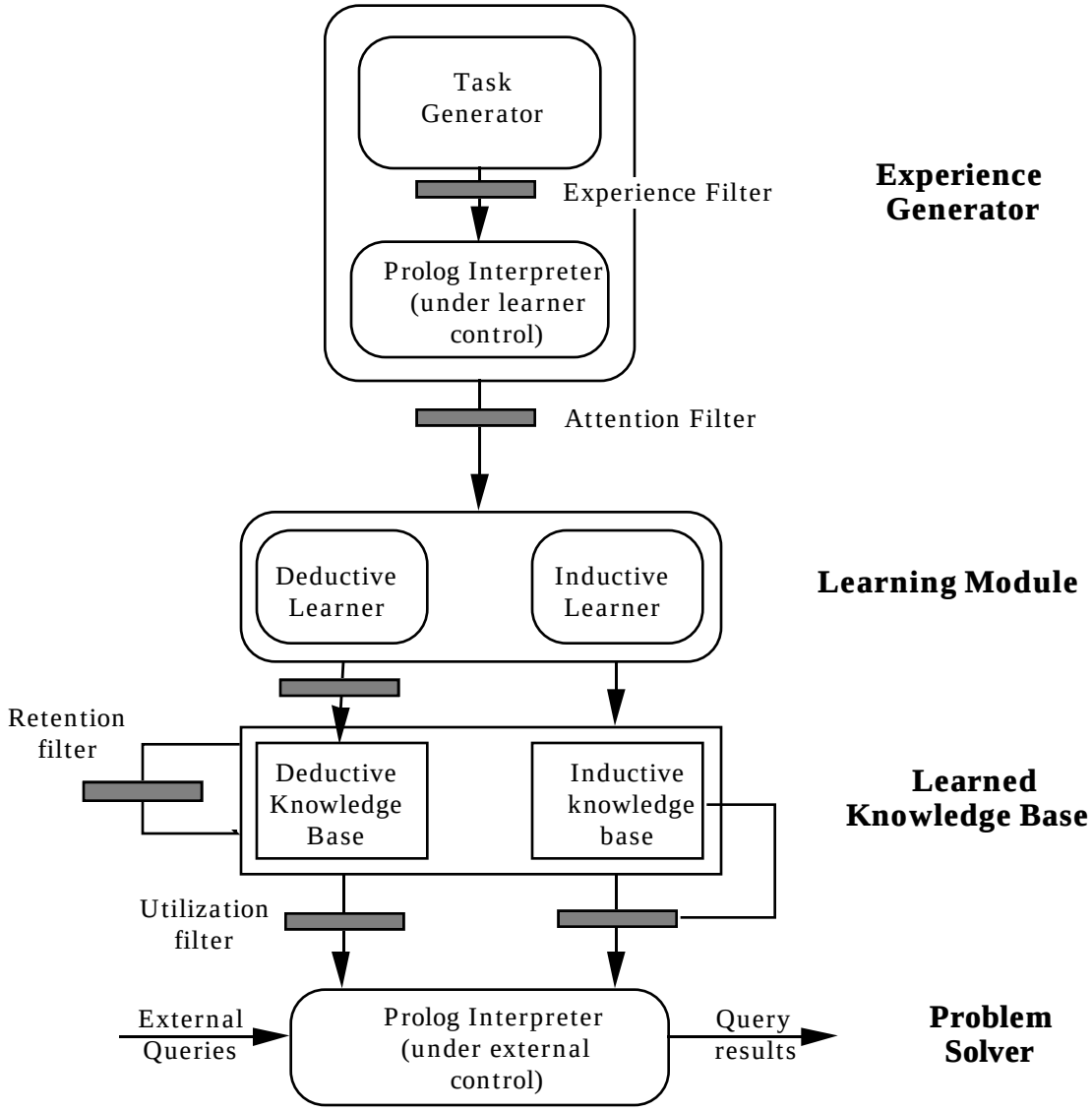

Figure 5. The primary learning layer of LASSY together with the selection processes (filters) that it employs.

4.4.1. Selective experience

LASSY employs a learning method called "learning by doing" or "learning by experimentation," i.e., it feeds the problem solver with problems to solve and acquires a variety of knowledge during the process of problem solving. If there are enough problems that were given in the past by external agents, then the system can use these problems as training

examples. If more training problems are needed , LASSY generates its own problems.

Since there is potentially a very large number of possible problems, the system may invest most of its learning resources in solving problems that will lead to the acquisition of irrelevant knowledge (which is likely to be harmful). LASSY employs an experience filter in order to decrease the likelihood that the acquired knowledge will be harmful by building the *task model* – the system's view of what problem space it faces. Currently, LASSY represents its task model by a weighted set of *calling patterns.* A calling pattern is a predicate name followed by a list of 0's and 1's that indicates which of the arguments of the predicate are bound and which are not. For example, ((ancestor 0 1) 35) is the pattern for queries that ask for ancestors of a specified person, and its weight is 35 (a problem of this type will be generated with probability of 35/total-weight). For each predicate that appears in the problem templates, the task model contains a list of the predicate domains. A predicate domain is a set of ground terms.

Whenever the task generator is required to produce another training example, it selects a random calling pattern according to its weight, and selects random constants from the domains to replace all the 1's in the pattern. Thus, the primary level acquisition programs face only a small subset of the experience space - a subset which is similar to the set of problems that were given to the system in the past, and is less likely to produce harmful (irrelevant) knowledge.

4.4.2. Selective attention

LASSY incorporates an attention filter as a part of the system architecture. Whenever the inductive learner faces a new experience (a trace of the Prolog search), it ignores all the features of a subgoal except its

calling patterns. This selection is a decision of the system builder and does not involve any secondary learning process. In future versions of the system I would like to make use of the specific bindings to classify subgoals in a more sophisticated way (see Chapter 7).

4.4.3. Selective acquisition

Selective acquisition is implemented for acquiring lemmas. Lemmas that save a very small amount of computation have a high likelihood of being harmful. Thus LASSY a collects computation value for every lemma. If the computation value is below a given threshold, the lemma is not added to the knowledge base.

4.4.4. Selective retention

Lemmas that are rarely used by the system have a high probability of being harmful. LASSY keeps track of the usage of lemmas during problem solving (during learning). The frequency of use of lemmas multiplied by their computational value gives an indication of the resources they save. Lemmas with low resource savings are deleted from the system's knowledge base.

4.4.5. Selective utilization

Chapter 5 describes a problem which occurs in any system that uses deductively learned knowledge and incorporates backtracking. If a lemma is used in a failure branch of the search tree, the usage is bound to be harmful, because the backtracking mechanism will force the interpreter to try the same binding twice: first when it is generated by the lemma, and a second time when it is generated by the rule that was used by the system to learn the lemma. Selective acquisition and selective retention can not reduce such harmfulness, since the context determines whether the usage

of a lemma is harmful or not. LASSY incorporates a utilization filter between the Prolog interpreter and the knowledge base. The filter does not permit the usage of lemmas for proving subgoals that have high probability of failure.

4.5. Secondary learning in LASSY

Most of the information filters in the LASSY system utilize knowledge that is acquired by secondary learning processes. In Section 3.3 it was argued that a secondary learning procedure together with the knowledge that it generates and the information filter using that knowledge can be perceived as an independent learning system. In LASSY all but one secondary learners use the same experiences that the primary learner use, i.e. they accumulate the information they need during problem solving of the training problems. Figure 6 shows the architecture of LASSY including the secondary learning systems which are indicated by gray regions. The following subsections give a brief account of the secondary learners in LASSY.

4.5.1. Acquiring task models

The secondary learning procedure that creates and updates the task model used by the experience filter, is the only one which uses different experiences than the primary learners. The task model builder receives as an input a set of queries that were given to the system in the past by external sources (users). For each query, the procedure checks to see whether its calling pattern is in the model. If it is in the model, the weight associated with it is incremented, otherwise it is added to the model.

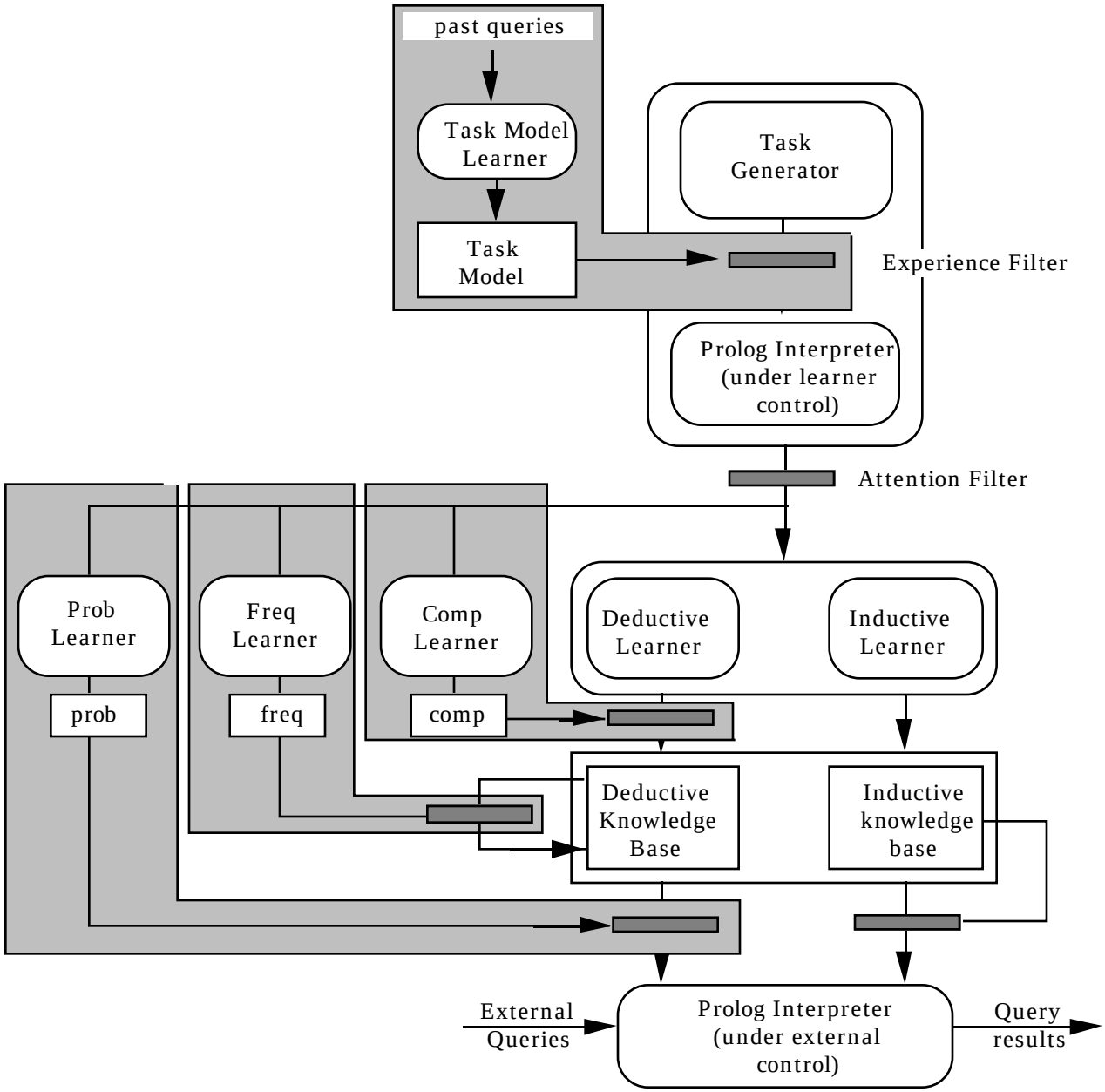

Figure 6. The secondary learning layer of LASSY, painted in gray shade.

4.5.2. Acquiring computational values

A computational value of a lemma is the number of unifications that were performed in the subtree that was searched in order to prove the lemma. Its role is to give some indication of how much computation would have been done if the lemma were not used. While solving problems during

learning, counters in all the nodes in the path from the current node to the root are incremented by 1 after every unification. Those counters are used as computational values when acquiring lemmas.

4.5.3. Acquiring frequency of use

Frequency of use of lemmas is accumulated while solving problems during learning time. The scheme used in LASSY is similar to the one used by Samuel in his checker player. The lemmas are given an initial age when acquired. Whenever a lemma is used, its age is multiplied by 0.5. Every fixed number of unifications, all the ages are incremented by one. This scheme gives more weight to uses in the recent history of the system.

4.5.4. Acquiring probability of failure

The probabilities of the failure of subgoals with specific calling pattern are also accumulated while solving training problems. There are four types of probabilities that are maintained. Further details about the probability acquisition are given in Chapter 5.

4.5.5. Information filters in the secondary learning systems

The secondary learners, being learning systems, can also employ selectors to filter information. Since all the secondary learners, except the task model learner, use the same experiences as the primary learners, their experience is filtered to be similar to past tasks. The procedure that acquires probabilities ignores all features of subgoals except their calling patterns, and thus can be viewed as a fixed attention filter. The same can be said about the task model learner. Except those (rather trivial) filters, there are no selection processes for the secondary learners.

4.6. The experimental domain

The domain used for the experiments is a Prolog database which specifies the layout of the computer network in our research lab. It contains about 40 rules and several hundred facts about the basic hardware that composes the network and about connections between the components. The database is used for checking the validity of the network as well as for diagnostic purposes. Table 3 shows a part of the database.

The Prolog program was written with a strong emphasis on declarative programming. The Prolog *cut* system predicate was not used to improve performance (it was used once implicitly by the *not*). The <,> symbols in the definition of the linked procedure are symbols of POST-Prolog. They were put around the body of the recursive procedure to stop POST-Prolog from reordering the subgoals (which would cause an infinite loop due to the left recursion).

All the experiments described in this thesis used the same scheme for generating problems. A problem domain was created in a structure similar to the structure of the task model, i.e., a set of domains and a weighted set of queries templates. The problems were generated in the same way that LASSY generates problems for itself: by selecting a problem template with probability proportional to its weight and instantiating the appropriate arguments with a randomly selected constants. The same problem domain was used for generating the example problems for the task model learner and for the creating the testing set to measure system performance.

Table 3. A portion of the database that was used for the experiments conducted with LASSY. The database describes the physical layout of a computer network.

```
check_plug(Plug,Port) :- inserted(Plug,Port), need_checking(Plug,Port).
need_checking(Plug,Port) :- not(proved_insertion(Plug,Port)).
good_plug(Plug) :- inserted(Plug,Port), proved_insertion(Plug,Port).
proved_insertion(Plug,Port) :- nine_pin_port(Dev,Port), communicate_somehow(Dev,X).
proved_insertion(Plug,Port) :-
plug(C,Plug),between(C,X,Y),communicate_somehow(X,Y).
communicate_somehow(X,Y) :- communicate(X,Y).
communicate_somehow(X,Y) :- communicate(Y,X).
between(C,X1,X2) :- cable(C,C1,C2),inserted(C1,P1),inserted(C2,P2),port(D1,P1),
        dev_con_link(X1,D1),port(D2,P2),con_dev_link(D2,X2).
dev_con_link(Dev, Con) :- pluged_in(Con,Dev).
dev_con_link(Dev, Con) :- pluged_in(ConB,Dev),linked(ConB,Con).
con_dev_link(Con,Dev) :- pluged_in(Con,Dev).
con_dev_link(Con,Dev) :- linked(Con,ConB), pluged_in(ConB,Dev).
dev_dev_link(Dev1,Dev2) :- dev_con_link(Dev1, Con), con_dev_link(Con, Dev2).
pluged_in(C2,X2),linked_somehow(C1,C2).
linked_somehow(X1,X2) :- linked(X1,X2).
linked_somehow(X1,X2) :- linked(X2,X1).
<
linked(X1,X2) :- directly_linked(X1,X2).
linked(X1,X2) :- <directly_linked(X1,X3), linked(X3,X2)>.
>
directly_linked(X1,X2) :- port(X1,P1),inserted(C1,P1),cable(C,C1,C2),
            inserted(C2,P2),port(X2,P2).
pluged_in(X1,X2) :- nine_pin_plug(X1,P1), inserted(P1,P2), nine_pin_port(X2,P2).
nine_pin_plug(X1,CP1) :- connector(X1,CP1,CP2,CP3).
plug(X1,CP1) :- cable(X1,CP1,CP2).
plug(X1,CP2) :- cable(X1,CP1,CP2).
nine_pin_port(X1,P1) :- device(X1), at_port(X1,P1).
port(X1,CP2) :- connector(X1,CP1,CP2,CP3).
port(X1,CP3) :- connector(X1,CP1,CP2,CP3).
port(X1,P1) :- extender(X1,P1,P2).
port(X1,P2):- extender(X1,P1,P2).
device(X) :- mac(X).
device(X) :- lw(X).
device(X) :- netmodem(x).
device(X) :- iw(X).
device(X) :- ibm(X).
```

4.7. Assumptions and simplifications

To make the LASSY project feasible, several simplifications and assumptions needed to be made. Some of the assumptions are necessary in

order to make LASSY work, while others could be removed by investing some more work in the system.

1. For simplification, all the experiments reported here were done with a fixed database, i.e., no modifications were made during the experiments. This simplification is not essential for subgoal reordering, since a modification will cause an adjustment of the averages after further learning activity. For lemma learning, it is only important not to remove clauses. Adding clauses will not change the truth of lemmas (unless clauses with extra-logical predicates like cut are added). Chapter 7 discusses the implementation of a truth maintenance system for the lemma database that will also allow the removal of clauses.

2. It is assumed that users' queries are drawn from a space of fixed distribution over a set of calling patterns. Fixed distribution is a very common assumption for learning systems. Most of the work in the theory of learning (Kearns, Li, Pitt, & Valiant, 1987; Valiant, 1984) assume fixed distribution over the set of input. The question whether a fixed distribution over the classes of calling patterns is reasonable is an open question. To test if the assumption is reasonable, research should be conducted which analyses queries to existing databases. Such research would be of interest to the author, but is out of the scope of this thesis. LASSY can handle a change of the fixed distribution to a new one but it requires some time: the task model learner will adjust the weights associated with the query classes, the task generator will generate queries according to the new distribution and the primary learner will learn lemmas and adjust averages of number of solutions and costs.

3. All the experiments reported here were done with a database that contain no functional terms. There is nothing in the architecture of LASSY that prevents the use of functional terms. However, a more sophisticated task model would be needed to allow a better selection in the space of experiences that would increase significantly when using functional terms. Also, the use of calling patterns should be more sophisticated.

4. All the queries submitted to the system were assumed to be single term queries. No conjunctive queries were tried. It is no problem for LASSY to handle conjunctive queries; however, the task model should be expanded to be able to handle such queries.

5. THE DEDUCTIVE COMPONENT: LEMMA LEARNING

The deductive component of the primary learning system in LASSY is the lemma learner. This chapter describes in detail the lemma learner and the filters that it employs. The chapter also contains description and analysis of the experiments that were done with the lemma learner. Sections 5.1 and 5.2 describe the basic mechanism of lemma learning. Section 5.3 shows some results of experiments done with the lemma learner. Some of the experiments had surprising results which suggest that lemmas can be harmful, and sections 5.4 and 5.5 describe attempts to reduce this harmfulness by the traditional information filters: selective acquisition and selective retention. The attempts were not very successful, and Section 5.6 explains why. It describes a general phenomenon that creates a problem for any problem solver that employs backtracking and that uses deductively learned knowledge. It also suggests some methods to solve the problem and points to basic problems with those methods. It proceeds by proposing a new method which falls under the definition of selective utilization. An implementation of that method is described. The section then describes experiments done to study the effect of selective utilization and argues that indeed utilization filters can be very effective in reducing the harmfulness of lemmas. Section 5.7 summarizes.

5.1. Lemma learning

The basic lemma learning mechanism is simple: whenever the Prolog interpreter proves a subgoal (exits successfully from an OR node), the substitution that made the subgoal successful is applied to the subgoal, and the substituted term is added to the lemma database. Whenever the Prolog interpreter tries to prove a goal (a new OR node is created), the lemma database is appended in front of the regular database, and the concatenated database is used to look for matching clauses. A lemma is not learned if a procedure with side effect was called anywhere within the subtree below the subgoal that was used to generate the lemma. No generalization of the type described in (Prieditis & Mostow, 1987) is performed in LASSY.

The lemma learning algorithm described above is similar to those suggested in (Hogger, 1984; Kowalski, 1979). LASSY's lemma learner is different from the one described in (Boyer & Moore, 1977). There, lemmas are used as hints given by the user to the system to help solve a complex theorem.

Using the search space terminology, lemma learning can be viewed as macro learning. Let $S_1 \Rightarrow S_2$ stand for "state S_2 can be derived from state S_1 using any number of transitions." Then, lemma learning can be described as:

$$[\ G_1,G_2,\ldots,G_n \Rightarrow (G_2,\ldots,G_n)\theta\ ] \rightarrow [P := P \cup \{G_1\theta\}]$$

As P becomes larger as a result of the lemma learning, more transitions are added to T. Specifically, let the learned lemma be $L=G_1\theta$. The set of transitions added to T is:

$\{<<L',H_1,\ldots,H_n>,<H_1,\ldots,H_n>\theta'> \mid L'$ matches L with substitution θ' for any $H_1,\ldots,H_n\}$

Thus, using the state space notation, a lemma adds a general macro to the transition function. In the terms that were defined in Section 2.3, lemmas constitute search space knowledge, since they change the topology of the search space, adding links that did not exist before.

5.2. Deductive learning

Lemma learning is a special case of deductive learning. A deductive problem solver is a program whose basic knowledge is a set of assertions and a set of derivation rules to derive new assertions. Thus, logic systems are deductive - the set of assertions are the axioms, and the derivation rules are the rules of logical inference. A grammar is a deductive system where the basic assertion is the start symbol, and the derivation rules are the grammar derivation rules. A state space search program is a deductive system where the set of initial states are the basic assertions and the operators are the derivation rules which allow the program to derive new states.

Any deductive problem solver can form the basis of a deductive learning program. A deductive learning program transforms knowledge from its implicit form to its explicit form. If the problem solver derives B from a set of assertions A using a sequence of applications of the program's derivation rules, the learner memorizes that B can be derived from A by adding a specialized derivation rule that specifies that fact explicitly.

There are many learning programs that are deductive by nature. All explanation based learning programs and macro learning programs use

such a scheme (Dejong & Mooney, 1986; Fikes, et al., 1972; Iba, 1989; Korf, 1985; Laird, Rosenbloom, & Newell, 1986; Markovitch & Scott, 1988b; Minton, 1985; Minton, 1988a; Mitchell, Keller, & Kedar-Cabelli, 1986).

The basic idea behind deductive learning is that by adding the explicit derivations that the system has experienced in the past, the problem solver will be able to solve problems more rapidly in the future. In later sections we will see that a problem common to deductive learners is that the added knowledge has costs in addition to its potential benefits, and if these costs exceed the benefits, then the knowledge is harmful.

One noticeable difference between the programs cited above is that some of them generalize the learned deductions while other do not. Is this difference essential? If the generalizations are deductively justified (such as in (Mitchell, et al., 1986)) then the programs that use generalization are basically equivalent to the more simple schemes, except that they learn sets of explicit derivation rules instead of one rule at a time.

Lemma learning looks very specific at first glance, but the search space analysis in the previous section reveals that a lemma is actually a general transition rule which applies to all states where the left-most goal matches the lemma. It is not as general as PROLEARN (Prieditis & Mostow, 1987), but the generality carries its cost. Segre (1987) observed that more general schemas can be more expensive to apply. Section 5.7 analyses the cost associated with using the lemmas of PROLEARN and compares it to the cost of applying the more specific version used by LASSY.

5.3. Experimenting with the lemma learner

The first experiment was run in order to find out whether the system's performance improves by acquiring lemmas. All other learning mechanisms and selection processes were turned off for the whole duration of the experiment. The experiment was conducted in the following way:

1. An arbitrary task domain T1 was defined as a set of problem templates and weights associated with each template.
2. A set of 25 problem was randomly generated from the domain. This set is called the *training set*.
3. Another set of 20 problems was randomly generated using the same domain. This set is called the *testing set*.
4. With all learning turned off, the system executed the whole testing set and the performance was recorded.
5. With learning turned on, the system executed 5 training problems.
6. With learning turned off, the system executed the testing set.
7. Steps 5-6 were repeated 5 times with different training problems.

Figure 7 shows the learning curve of LASSY with lemma learning turned on. Using the lemmas, the system performance deteriorated by a factor of 2. The results of this experiment are surprising. Lemma acquisition is a very simple learning mechanism. What can be wrong in trying to avoid repeating the execution of tedious tasks?

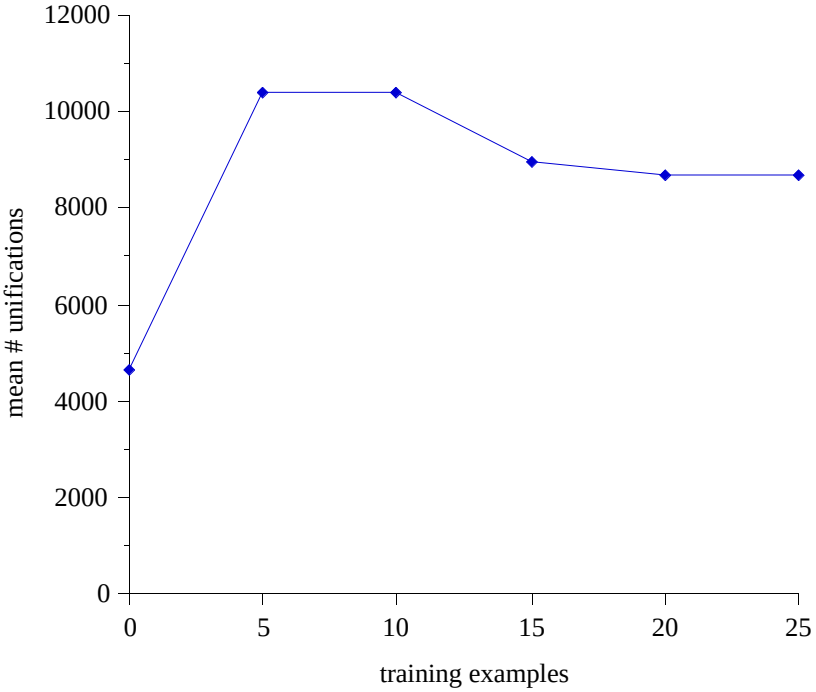

Figure 7. Performance during lemma learning. Smaller number of unifications indicates better performance. Each point in the curve represents an average performance of a test of 20 problems.

Recalling the analysis given in Chapter 2, the first observation that can be made is that this case is a blatant example of harmful knowledge. It is clear that the benefits of using lemmas can be very high - instead of proving a subgoal by an exhaustive search, the program has the answer readily available. That makes the results even more puzzling. If the benefits are so high, the costs must be much higher to cause such a deterioration in performance. The additional unifications done when the lemma does not match the current goal constitute the obvious cost associated with using lemmas, but the first argument indexing of Prolog should have reduced this problem substantially.

5.4. Selective acquisition

The first attempt at reducing the harmfulness of the learned lemmas is to acquire them selectively. To implement a selection mechanism, a heuristic for estimating the value of a lemma must be found. The heuristic that was used in LASSY is a very simple one. Define the computation value of a lemma to be the cost of the subtree that was searched in order to generate the lemma, then acquire only lemmas whose computational value is above some predefined threshold.

The rational behind this method is that the benefit of using lemmas with high computational value is large, and thus is more likely to offset the costs of using these lemmas. The computational value of a lemma is generated while proving a query under the learner control. Every node in the AND-OR search tree has a counter. Each time the interpreter performs a unification, the counters for all the nodes in the path from the current node to the root are incremented by one.

Experiment 2 was conducted in the same manner as experiment 1, but an acquisition filter was added to the system. The experiment was repeated three times, with acquisition threshold assuming the values of 10, 100 and 1000 unifications.

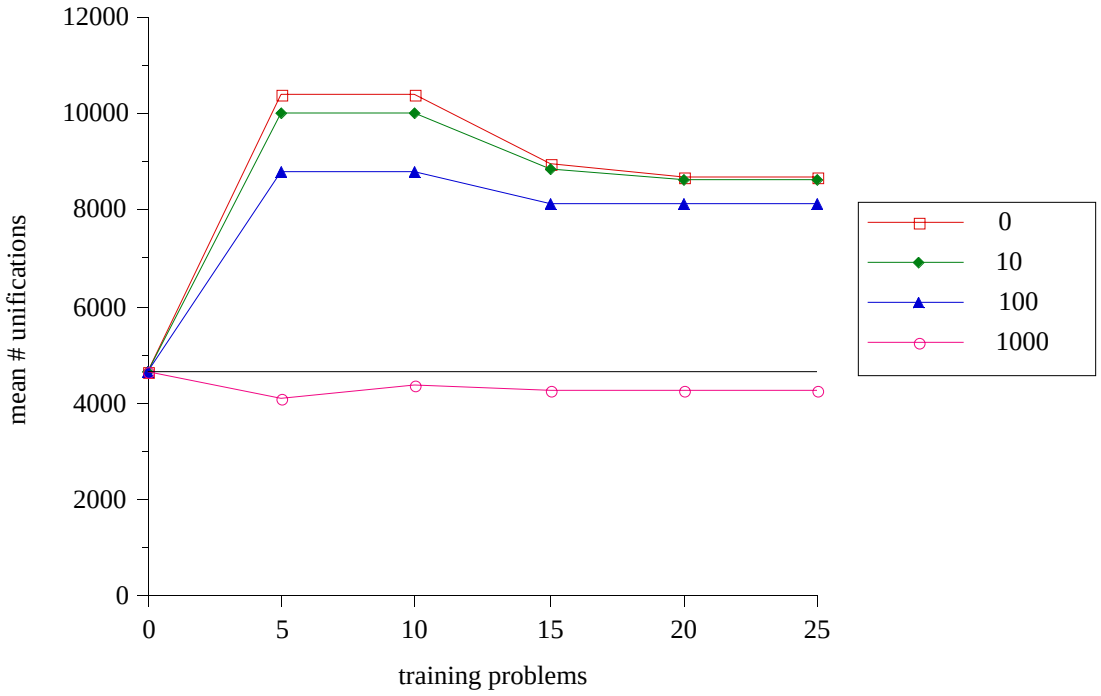

Figure 8. Performance during learning with selective acquisition. Each graph represents one

Figure 8 shows the results obtained during experiment 2. Table 4 shows the number of lemmas that the system acquired using the various threshold values. It is clear that the acquisition filters with a small thresholds (10 and 100) led to slightly better performance than unfiltered learning, but the results were still much worse than those obtained with no

Table 4. Number of lemmas left with the different values of thresholds for the acquisition filter.

Threshold value	Number of lemmas acquired
0	117
10	112
100	45
1000	19

learning. An acquisition filter of value 1000 made a significant improvement, but the learning curve is very close to the horizontal line that marks the performance with no learning. Thus it is clear that selective acquisition did not help, except in that it reduced the number of lemmas, thus reducing the harmfulness of the lemma database; but the system would do just as well without learning lemmas at all.

One possible reason for the inefficiency of the selective acquisition is that it does not take into account the usage that the system makes of the lemmas. It is quite possible that a lemma has a small computational value, but is used so often that it has greater benefit than a lemma with a very high computational value which is rarely used. The next section describes another type of filter that tries to take the frequency of use into account: selective retention.

5.5. Selective retention

In Chapter 3, and in (Markovitch & Scott, 1988a; Markovitch & Scott, 1988b), it was claimed that selective acquisition suffers from two basic deficiencies:

1. It is basically a hill-climbing approach.
2. It is hard to assess the usefulness of a knowledge element before putting it into actual use.

Selective retention (or forgetting) can overcome these two problems. It is not a hill-climbing approach since in principle it is possible to evaluate all the subsets of knowledge to select the best one to retain (although not practically, because of the exponential complexity of such process).

Selective retention can also utilize the experience that the system has had with the various knowledge elements to estimate their value.

The heuristic that LASSY uses to estimate the value of a lemma is similar to the one used by Markovitch and Scott (1988b) and to the one used by Minton (1988a). The idea is that the benefit that the system gets from a lemma can be defined as the frequency of the lemma's use multiplied by the cost saved each application. The cost saved is assumed to be the computational value, thus the only other piece of information needed is the frequency of use. The scheme that was used in LASSY is the one described in (Samuel, 1963). Each new learned lemma is assigned an initial "age". Every fixed period of system operation (100 unifications), all the "age" counters of the lemmas are incremented by 1. Whenever the system uses a lemma, its age is cut by half. This scheme gives new lemmas a chance to prove themselves, and also gives preference to more recent uses. The computational value is divided by the age to get a utility estimate.

Experiment 3 tests the retention filter in the following way:

1. The training set (the same one that was used in experiments 1 and 2) is given to the system for execution with learning turned on.
2. The utility is calculated for all lemmas and the lemmas are sorted according to their utilities.
3. 10% of the lemmas with lowest utility are removed from the lemma database.
4. The testing set is executed with learning turned off.
5. Steps 3-4 are repeated 10 times. (the number of lemmas removed at each stage is 10% of the total number of lemmas at the beginning of the experiment and not 10% of the remaining lemmas)

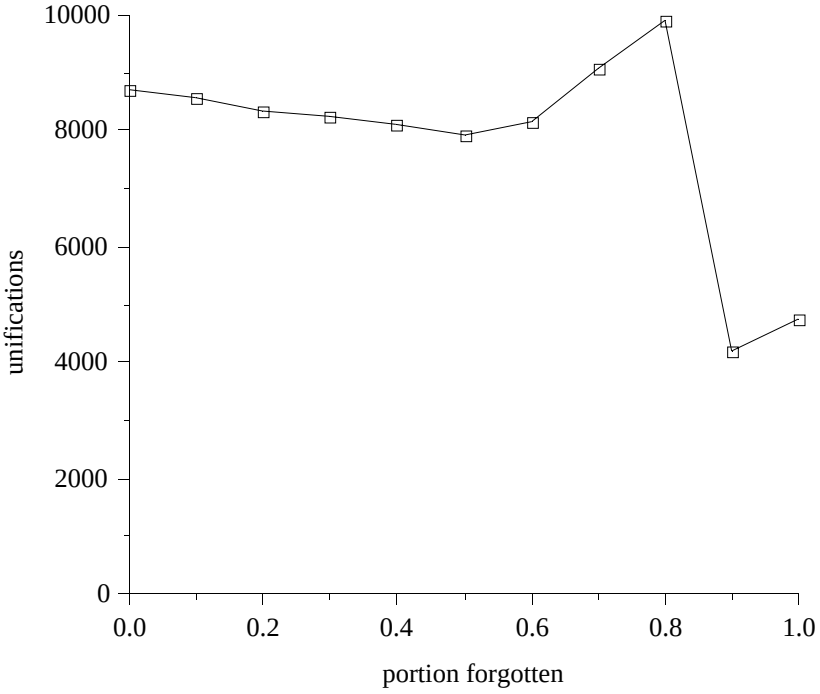

Figure 9. Performance after portions of the lemma database were forgotten. Each point on the graph represents a test of 20 problems.

Figure 9 shows the result of experiment 3. It is clear that selective retention did not solve the basic problem. What is even more paradoxical is the sudden improvement in performance when the 10% of lemmas with second best utility were removed. By now, it is time to reconsider our former assumptions. If the lemmas that were used often are the culprit , maybe the usage, rather than the matching, of the lemma is what costs most.

5.6. Selective utilization

5.6.1. The inherent harmfulness of deductively learned knowledge

The last two sections made it clear that utility in the simple form of frequency-of-use × computation-saved is not a good estimate for the value of

a lemma. Looking into the results in finer detail revealed an interesting phenomenon: there was a significant difference between the behavior of the system with queries that did not have solutions versus the behavior with queries that had solutions. To test this observation, the means for experiment 1 were recalculated for the two groups (will be called from now SOLVABLE and UNSOLVABLE) of queries separately. Figure 10 shows the results for the SOLVABLE set .

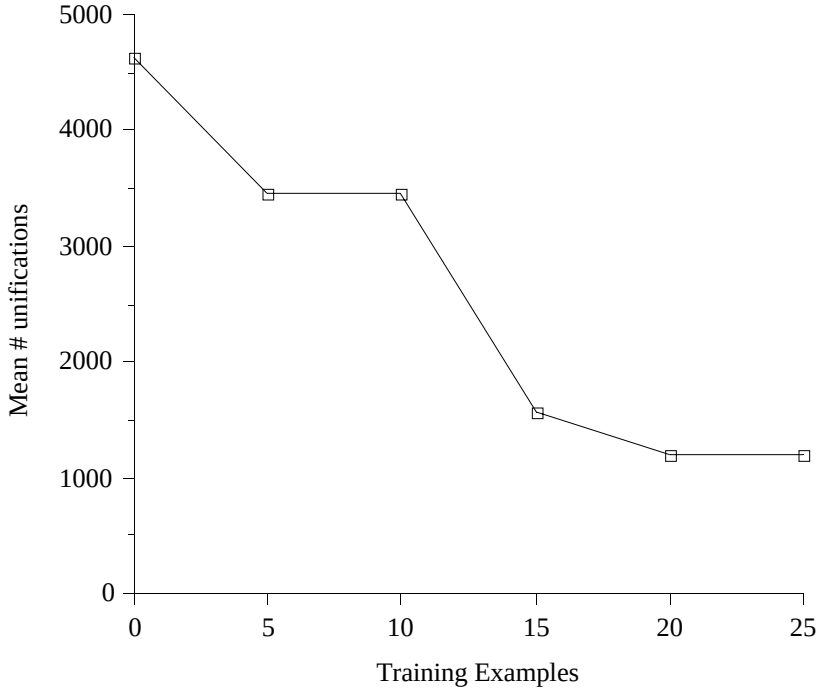

Figure 10. The learning curve of experiment 1, taking into account only problems that are solvable.

The graph for the SOLVABLE set is a decent learning curve where the system performance after learning is 3 times as good as the performance before learning. That curve indicates that learning lemmas *can* be useful for a certain type of problem. Figure 11 shows the results for the UNSOLVABLE set.

The graph for the UNSOLVABLE set has a similar shape to that of the original experiment, but the deterioration in the system behavior is much worse. The reason for the flat shape of the UNSOLVABLE graph is that a maximum of 50,000 unifications was set to make the experiments feasible[1]. Most of the unsolvable problems reached this limit after 5 training examples. Without the limit, the graph for the UNSOLVABLE set would rise much higher.

The two graphs make it clear that lemmas cost much more when used in queries that fail. What happen when a goal fails? The backtracking mechanism forces the interpreter to search the whole tree. Solutions that are generated by the lemmas are regenerated by the rules that were used to create the lemmas. This problem is not restricted to lemma learning - it applies to all problem solvers that utilize deductively learned knowledge and incorporate backtracking into their search

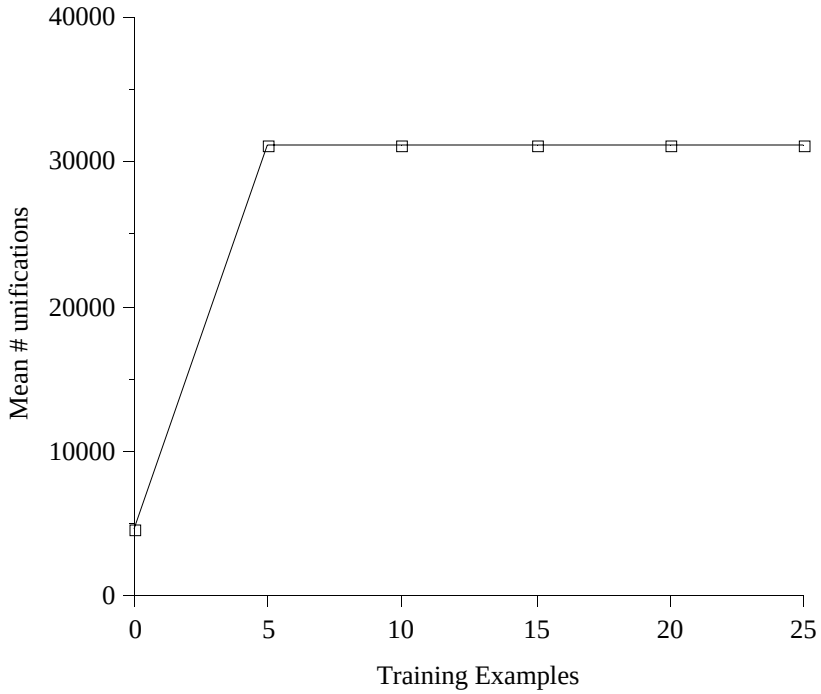

Figure 11. The learning curve of experiment 1, taking into account only problems that are unsolvable.

[1]

procedure. The next section defines the scope of the problem: deductive learners.

The Backtracking Anomaly

Assume that a search program performs a depth first search (with backtracking) in the space illustrated in Figure 12. Assume that the program is given the problem of getting from state A to state D. Assume that during this search, the learning program learned a new macro - it is possible to get from B to C (we will call this macro/rule B-C). Assume that the problem solver receives another problem - to get from A to E. Assume that there is no route from B to E. The problem solver will get to B and use the macro B-C to get to state C. The search from C will not lead to E, thus the problem solver will backtrack to B. Since there is no route from B to E, the problem solver is bound to search the whole subtree of B, including the search that generated B-C in the first place. The whole subtree under C will be searched twice because the problem solver will get to C twice - once by using the macro B-C, and once by going through the path that generated B-C. Thus, the system would be better off not using the macro in the first place.

The reason that the maximum value that the graph shows is close to 30,000 rather than close to 50,000 is that the UNSOLVABLE set includes some problems that the system could execute in a reasonable time. This problems reduced the average performance to about 30,000 unifications.

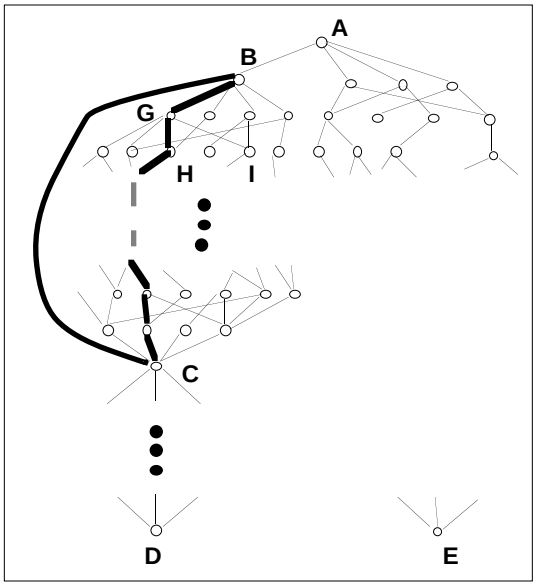

Figure 12. A sample search space that demonstrates the backtracking anomaly. BC is a learned macro.

The same problem will occur in a system that uses backward reasoning, such as Prolog. Assume that the learning program learns lemmas which are instantiated subgoals that were proved during the learning phase. If there is a rule:

$$p(X) \leftarrow q(X) \,\&\, r(X)$$

then the interpreter will use lemmas of the type q(c), where c is a constant, to generate bindings for X. If r(X) rejects all the bindings generated by q, then q will be forced to reinvoke the rules that generated the lemmas in the first place. In addition to the fact that the execution of q(X) by itself will be more costly than it would have been without using lemmas at all, r(X) will be invoked twice on every binding that was generated by q's lemmas (since the same binding will be generated again using the rules). Thus the system performance will be harmed by using the lemmas instead of being benefited.

Let's look at a more concrete example. In the experimental database that is used for the experiments described here, there is a definition for the relation *link*:

linked(X,Y) ← directly_linked(X,Y).

linked(X,Y) ← directly_linked(X,Z), linked(Z,Y)

The relation directly_linked is not directly available to the system but is defined in terms of more primitive relations. Assume that each machine is directly_connected to a maximum of one other machine. During learning the system acquires several lemmas of the form directly_connected(machine1,machine2). Assume that a query of the form linked(machineK1,machineK2) was given to the system, and that the two machines are not connected. Without lemmas, if the greatest distance between machineK1 and other machines is N, then the procedure linked will be invoked N times before the interpreter will come back with the answer "no." With lemmas, directly_linked will generate each machine twice, thus the first linked subgoal will be called twice, the second will be called 4 times, and the last will be called 2^N times. Thus, the example shows that the cost of using lemmas can be exponentially high.

The backtracking anomaly should not be taken lightly. Almost every problem solver spends a large portion of its search time exploring failure branches. When Prolog is used as it is meant to be used - as a declarative language - the rate of failures during the proof process is very high. The lemmas learned by our lemma learning system described in Section 3 were found to be harmful in many cases when the rate of failure within a proof was high. All rule systems which use depth first search (with backtracking) are bound to have similar problems if they use deductively learned knowledge.

Possible Solutions

In this section we will explore several possible solutions to the backtracking problem. One possible solution is to add to the problem solver a procedure that checks whether a node has been visited before. In a case like the search problem of Figure 12, that will save the program the second search of the subtree of C. There are two problems with adding such a check to the problem solver. The first is that such a test has potentially very high costs in terms of both memory and time. The second problem is that it does not eliminate the problem - it only reduces the costs that are caused by the problem (the problem solver would still get to state C twice).

Prolog itself has no facility for implementing such a feature. POST-Prolog was modified to experiment with a version of such a check: Whenever a child of an OR node was returned successfully, the bindings that were generated by the child were compared to the bindings that had been generated before by its siblings. If two sets of bindings were identical, the interpreter marked the new child as a failure and went to the next alternative. Such a mechanism would not stop q(X) from generating the same bindings twice, but would save r the necessity of running again with the same bindings . The check improved the performance of the interpreter by a factor of two, compared with the performance without the check. The problem is that the improvement was measured by unifications. Such a measurement does not take into account the overhead of maintaining and testing for membership in the binding list.

A second possible solution is to use *lemma groups*. The idea behind lemma groups is that if an OR node had failed during *learning* time, the learner can be sure that all possible bindings to the subgoal were found, and can learn them all as a group. together with a lemma that says "and

these are all the possible bindings" (by using the Prolog cut operator). In such a case, there will be no need to go to the rules if all the axioms fail, since the interpreter knows that the rules can not generate new bindings that have not already been tried yet.

Lemma groups can be very beneficial, especially for problems with high failure rate. In some cases, for problems that required very large space trees with a high rate of backtracking, using lemma groups made execution up to 30 times faster. Unfortunately, lemma groups have their own disadvantages. It is extremely hard to maintain lemma groups in a system that changes over time. If a new axiom is added to the database, there is a possibility that the lemma group is not valid anymore.

Another possible solution is to use selective acquisition or selective retention to try and reduce the harmfulness of lemmas. Sections 5.2 and 5.3 demonstrated that these two solutions do not work. Now it is clear why: whether a lemma is harmful or not is not a property of the lemma itself, but depends on the context in which it is being used. A lemma can not be blamed for the fact that some subgoal that was executed further in the computation stack rejected all the solutions generated by the lemma or by a subgoal that precedes the subgoal that used the lemma.

This is exactly the kind of situation where selective utilization can be effective. A utilization filter is called during problem solving time; thus it can use the current state of the computation to decide whether to use the lemmas or not. The utilization filter will be used in order to reduce the probability that lemmas will be used where they can be harmful. Since using lemmas in a subtree of a goal which fails is *bound* to be harmful, the filter tries to turn off lemma usage when it estimates that the probability for such a failure is high.

5.6.2. Implementing a utilization filter to reduce the harmfulness of lemmas

A utilization filter function was implemented in LASSY that decides whether to use lemmas or not whenever a new OR node is created and added to the search tree. Basically the filter tries to minimize the use of lemmas in the subtree below a subgoal that is likely to fail. Using lemmas in a subtree that fails is bound to have detrimental effect on the search time because backtracking will force the interpreter to search the whole subtree.

Since it is impossible to know in advance whether a goal will fail, the filter estimates the probability of failure from past experience. In the current scheme, if the probability of a goal failing is above some threshold, the filter disables lemma usage for the subtree below the goal.

The probabilities are updated during the learning phase. Currently, there are four types of failure probability that the system maintains:

1. The probability of a goal with a specific predicate failing.
2. The probability of a goal with a specific predicate, and specific arguments bound, failing.
3. The probability of a specific goal within a specific rule body failing.
4. The probability of a specific goal within a specific rule body, and with specific arguments bound, failing.

The reason that the context of the rule is taken into account is that many times a subgoal within a rule body will succeed at first, but will eventually fail because a subsequent subgoal in the rule body keeps rejecting the bindings generated by the subgoal. For example, assume that a database of the living members of a family contains the following rule:

greatgrandfather(X) ← male(X) & parent(X,Y) & grandparent(Y,Z)

If the rule is used to find a greatgrandfather (i.e. greatgrandfather(X) is called with unbound X), then the probability of parent(X,Y) failing within this rule is very high. The reason is that male(X) will generate males, parent(X,Y) will succeed in finding children of the given males, but grandparent(X,Y) will keep failing because most living people are not greatgrandparents. On the other hand, the probability of parent(X,Y) failing by itself is lower since a substantial portion of the people in the database are parents. This example demonstrates why it is preferable to use more specific information.

The binding information specifies which arguments are bound and which are not (regardless of the values that arguments are bound to). The above example illustrates why bindings can be significant to the probability of failure. A goal greatgrandfather(X) is likely to succeed when X is not bound, assuming that there is at least one greatgrandfather in the database. However, the probability of greatgrandfather(c) failing, where c is some constant, is very high since most people in the database are not greatgrandparents.

Whenever a subgoal is called (an Or node is created), the learning program updates 4 CALL counters associated with the 4 types of probability described above. Whenever a subgoal fails (has tried all its OR branches, thus exhausting the whole search subtree), the learning program updates 4 FAIL counters. A probability is computed by dividing the FAIL counter by the CALL counter.

During problem solving, whenever the interpreter creates a new OR node for a subgoal, it consults the failure probability to decide whether to append the lemmas for the subgoal predicate to the database axioms for it. If the probability of failure is above a preset threshold, lemmas will not be

used, and a flag will be propagated down the subtree of the OR node to turn off lemma usage for the whole subtree.

The probability that is taken into account is the most specific one that is available, i.e. it first looks for probability type 4, and if not available, it looks for probability type 3, etc.

5.6.3. Experimental results

Experiment 4 was a replication of experiment 1 with the following changes:

1. During lemma learning, the secondary learning procedure that acquires probabilities of failure was turned on.
2. During lemma learning, the utilization filter was turned on.
3. During testing, the primary learner, which acquires lemma, and the secondary learner, which acquires probabilities were both turned off.
4. During testing, the utilization filter was turned on.
5. The threshold for experiment 4 was set to 0.5.

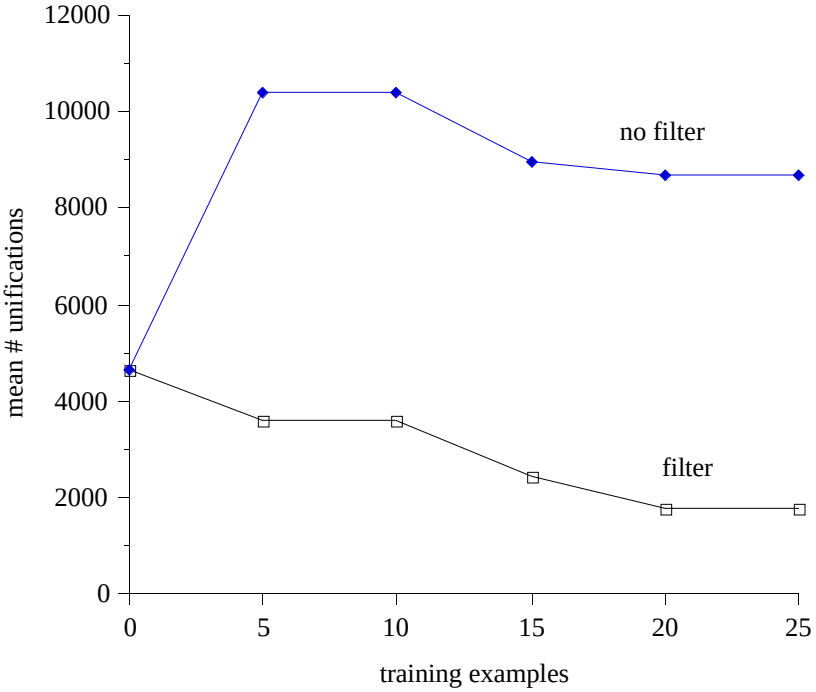

Figure 13. Learning curve with lemmas with two conditions: filter and no filter

The results of experiment 4 (Figure 13) are encouraging. Not only did the filter reduce the harmfulness of the learned lemmas, it actually brought the system to a point where performance with lemmas was 2.5 times better than performance without lemmas. Performance with the filter was 5 times better than performance without it (but if we removed remove the upper limit on number of unifications per problem, the gap would be much larger).

Since the system spends much more time on difficult problems, and since using the filter during learning results in a different processing time for the same problems, a question arises whether the traditional way of putting training examples on the X axis is accurate. It looks as if resources invested in learning is a better candidate for the X axis. The number of unifications during learning time was recorded, and Figure 14 shows the system's performance as a function of learning resources. It is clear that the patterns of the graphs in Figure 13 and Figure 14 are similar; however, Figure 14 reveals an interesting characteristic of a learning system that learns by experimenting: when learning proves to be useful, the problem solver needs fewer resources (particularly time) to perform the same tasks, thus the system needs to invest less in solving the training problems, and the system can potentially learn the same knowledge with less expense.

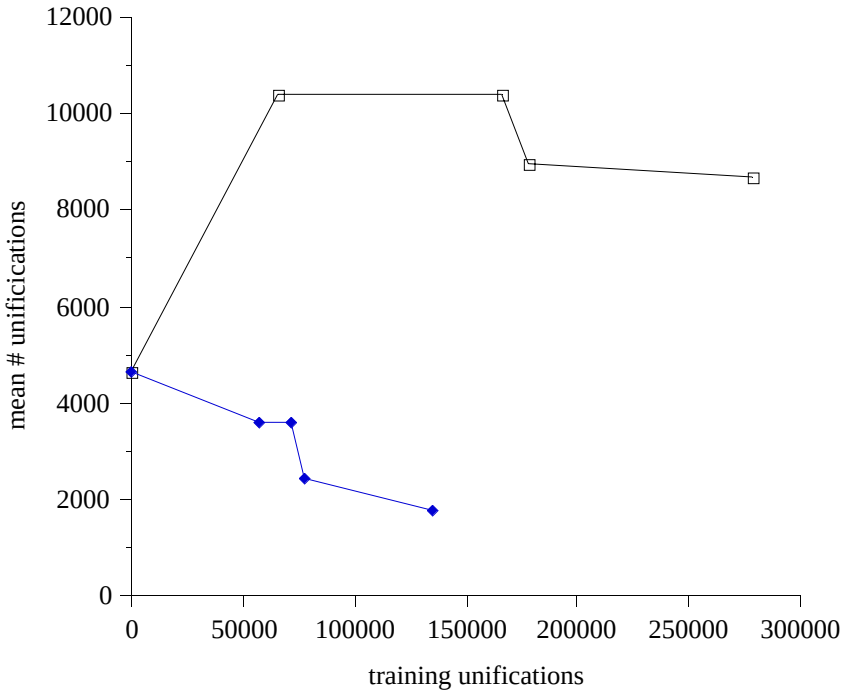

Figure 14. Learning curves with lemma learning. X axis is the number of unifications during learning.

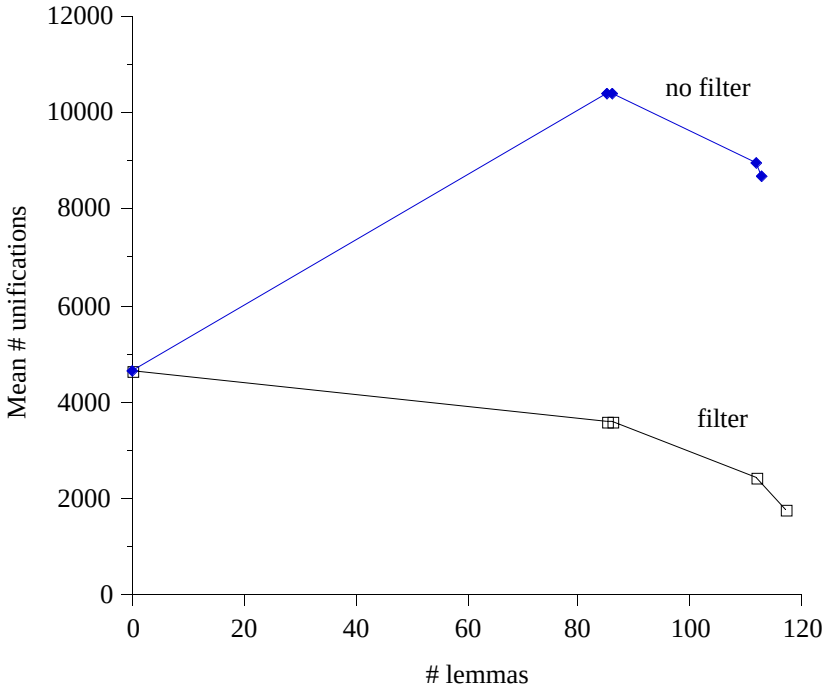

Figure 15. Learning curve. Performance as a function of the number of lemma acquired

Figure 15 shows another interesting graph - the performance as a function of the number of lemmas acquired. The patterns of the learning curves remain the same. The graph of the filtering condition is a bit misleading, since lemmas were not the only knowledge accumulated during the experiment. A secondary learner acquired the probability information, and that knowledge accounts for the difference between the two graphs.

The next experiment (5) studies the effects of using different threshold values in the utility filter. The experiment was conducted in the following way:

1. The system performed all the queries in the training set with learning turned on.
2. The testing set was performed 11 times with learning turned off, varying values of the probability threshold from 0 to 1 in increments of 0.1.

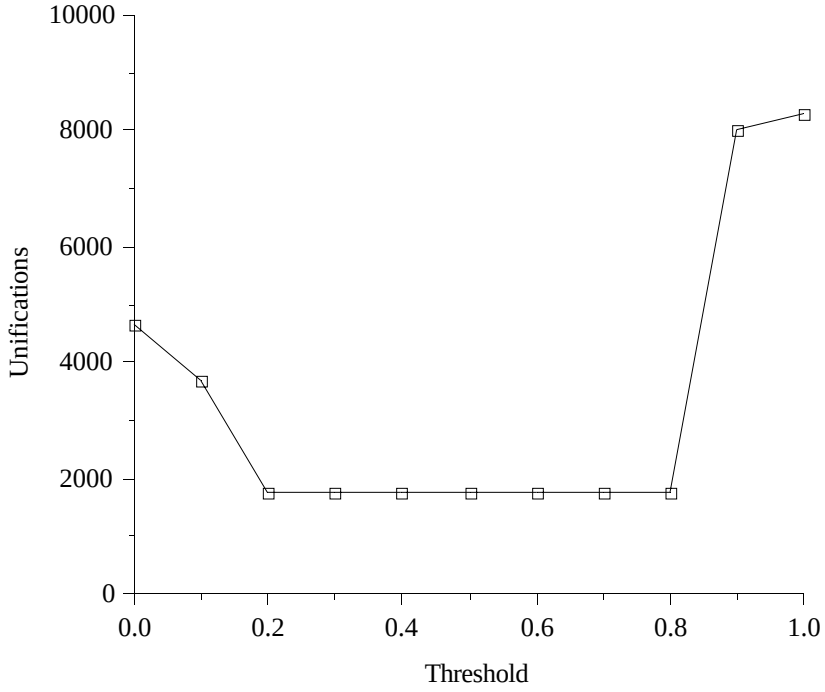

Figure 16. Performance using lemmas with different threshold values

The results of the experiments are shown in Figure 16. The U shape of the graph makes it clear that filtering should not be taken to extremes. If the filter is too refined, knowledge will not be used where it could be useful. If the filter is too coarse, knowledge will be used where it could be harmful. A general U shape was common to all the threshold experiments, but the graphs differs in their particulars. For example, the graph that appears in (Markovitch & Scott, 1989d) lacks the flat section in the middle, giving it rather a V shape. Other experiments showed a U with the lowest point to right or to the left of the point x=0.5. But the general U pattern was common to all the graphs.

Selective acquisition combined with selective utilization

One interesting question is whether the system can benefit from combining selective acquisition and selective utilization. Experiment 6 is the same as experiment 2 but both selective acquisition and filtering are turned on. The results are shown in Figure 17.

The graphs for both experiments (2 and 6) are displayed here. The f in the legend stands for filtering and nf stands for no-filtering. The use of

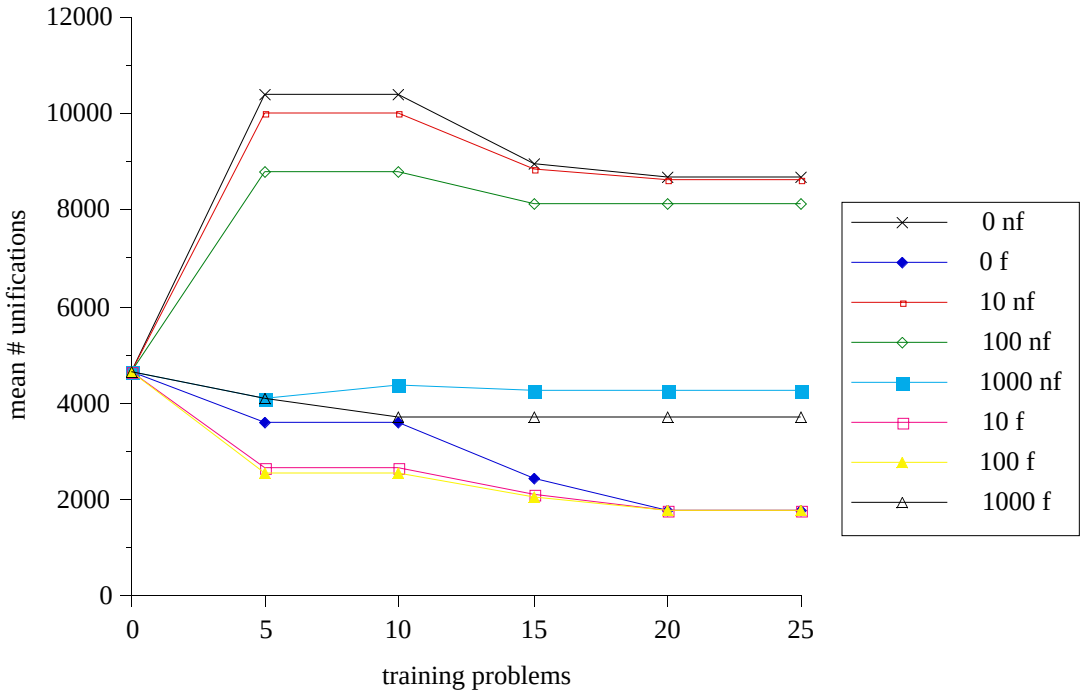

Figure 17. learning curves with selective acquisition and with filter and no filter conditions. n stands for filter, nf stands for no filter

selective acquisition did not change the final performance of the system, but made the learning curve better, i.e., the system approached its best performance more rapidly. Of course the learner cannot be too selective. The graph for a threshold of 1000 shows that being too selective make the system lose good lemmas.

Selective retention combined with selective utilization

Another interesting question is what are the effects of combining selective retention and selective utilization. Experiment 3 was repeated with the filtering turned on for the whole duration of the experiment. The results are shown in Figure 18.

The flat part of the graph, from 0% forgotten to 50% forgotten makes it clear that the system could perform just as well without 50% of its

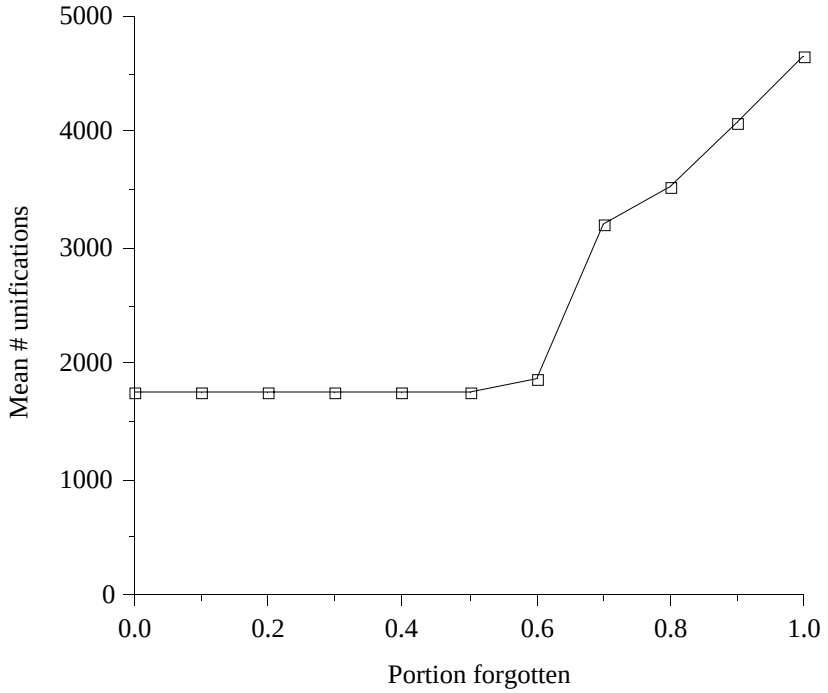

Figure 18. Performance with selective retention and filter.

lemmas. It is somewhat surprizing that there was no improvement in the system's performance after removing lemmas that were probably irrelevant. This is because the combination of filtering and forgetting changed the meaning of irrelevancy. There were procedures which were called only in the failure subtrees. The lemmas for such procedures would be assigned a very low utility, and they would be removed without any effect

on the performance of the system. If the Prolog interpreter did not use indexing for matching, the cost would probably be reduced by removing those lemmas.

5.7. Summary

Lemma learning is an example of deductive learning where the system makes implicit knowledge explicit in order to improve its efficiency. In LASSY, whenever an subgoal succeeds, the instantiated literal is added to the lemma database. Experiments performed with the lemma learner revealed a paradoxical result: using lemmas made the system performance deteriorate substantially. The lemma database proved to be a blatant example of harmful knowledge.

Two methods for reducing harmfulness of knowledge were tried: selective acquisition and selective retention. Both proved to be ineffective. A deeper analysis revealed the major source of harmfulness in lemma usage. Whenever a lemma is used in a failure branch of the search tree, the backtracking mechanism forces the interpreter to use both the lemma and the rule that was used to generate the lemma, and each solution is generated twice. The cumulative effect of this phenomenon can cause an exponential increase in search time. The reason that selective acquisition and selective retention did not work was that the harmfulness of a lemma is not a feature of the lemma itself, but depends on the context of its use. The backtracking anomaly is not unique to lemma learners, but affects any deductive learner that incorporates backtracking.

The solution proposed in this Chapter is a utilization filter. The system acquires information about the probability of subgoals with a

specific calling pattern failing. When the interpreter requests the set of lemmas for a specific goal, the filter first estimates the probability of the goal failing. If it is above a certain threshold, the filter does not let any lemma through, and the lemma usage for the search subtree of the goal is disabled. Lemma usage for a subgoal that is likely to fail is likely to be harmful, thus the filter reduces the likelihood of lemmas being harmful. Experiments that were performed using the filtering mechanism proved that selective utilization can be very effective.

5.7.1. PROLEARN vs. LASSY

PROLEARN (Prieditis & Mostow, 1987) is a learning program that tries to improve the efficiency of Prolog, thus it is interesting to compare it with LASSY. Both systems use lemmas of some kind in order to utilize past computations to speed up the interpreter. PROLEARN uses a generalized form of lemmas in the following way: whenever a subgoal is proven, a new rule is generated. The left side of the rule is the subgoal and the right side is a conjunction of all the primitive subroutines and user-defined facts. The rule is generalized using the EBG method to remove the dependence on particular bindings. PROLEARN's lemmas are more general than the lemmas generated by LASSY (which are unit clauses, typically ground).

The advantage of the general representation is that the lemma can apply to bindings that the system has not experienced before. The disadvantage is that the direct cost of using a lemma is much higher than the direct cost of using a lemma in LASSY. The direct cost of using a lemma in LASSY is one unification, whereas the direct cost of using a lemma in PROLEARN is the cost of proving all the subgoals in the body of its rule. Even in small databases, the rules that are generated by PROLEARN are likely to have many subgoals in their bodies. In addition,

the generality of the rule makes the problem even more severe because the lemma will be used more times. The easiest way of understanding the problem is to look at an example.

Suppose that in a Prolog database, the subgoal p(X) is called with 20 different bindings. Suppose that for 10 of the bindings (c_1,...,c_{10}), p(X) is provable and for the other 10 (c_{11},...,c_{20}), it is not. Suppose that p is defined by some rule R_p. Suppose that LASSY, while learning the database, acquired 10 lemmas of the form p(c_i), and PROLEARN learned one general rule $L_p = p(X) \leftarrow s_1,...,s_n$. Let's compare the cost of using the lemmas in LASSY and in PROLEARN in the following three cases (assuming no filtering is done in LASSY):

1. The goal to be proven is p(c_5) which is provable. The cost of proving the goal in LASSY is 5 unifications (in this case it will actually cost 1 unification because of the first argument index). The cost of proving the goal in PROLEARN will be 5*COST(s_1,...,s_n) which is likely to be much larger than 5.

2. The goal to be proven is p(c_{15}) which is not provable. The cost of proving the goal in LASSY is 10 (0 with the first argument index) + COST(R_p(c_{15})). The cost of proving the goal in PROLEARN will be COST(s_1,...,s_n) + COST(R_p(c_{15})). This is a case where the lemma usage in both systems is harmful, since the rule that was used to generate the lemmas had to be used. The difference between LASSY and PROLEARN is that for LASSY the overhead is the number of lemmas for the particular predicate (and when indexing is used the cost is 0) whereas PROLEARN must execute the body of its learned lemma.

3. The goal to be proven is p(X), and there is another goal in the computation stack, following p(X), which fails (for all bindings of p).

Assume that the cost of rejecting a binding of X is C. LASSY will generate all the bindings using the lemmas and then the same bindings using the rule, thus the cost will be 20*C+10+COST(R_p(X)) (where 10*C+10 is overhead). PROLEARN will use its two rules (the lemma and the original rule), and the cost will be 20*C+COST(s_1,...,s_n)+COST(R_p(X)) (where 10*C+COST(s_1,...,s_n) is overhead).

Thus, in all three cases, the cost of using lemmas in PROLEARN is higher than the cost of using lemmas in LASSY. Of course, if we take into account the filtering that LASSY uses, then the difference becomes much larger. Prieditis and Mostow acknowledged the problem of the possible harmfulness of lemmas, but did not identify the major source of harmfulness (the backtracking anomaly).

6. THE INDUCTIVE COMPONENT: LEARNING FOR SUBGOAL REORDERING

6.1. Subgoal reordering

It is well known that the efficiency of a Prolog program depends strongly on the order in which its subgoals are executed (Naish, 1985a; Naish, 1985b; Sterling & Shapiro, 1986; Warren, 1981). Even expert programmers can have problems when ordering the subgoals in rules. A rule whose body contains 8 subgoals can be ordered in more then 40,000 different ways, so it is quite likely that an expert programmer may order such a rule in a suboptimal way.

Several researchers have explored the possibility of using a program to reorder the subgoals. One of the first programs that automatically reordered goals was Chat-80 (Warren, 1981). Chat-80 is a natural language question answering system built by Warren and Pereira. The system translates the natural language questions into Prolog queries. A major part of the program is a planning procedure that reorders the query before processing it.

One problem with Chat-80 is that it uses cheapest first heuristics for reordering. Smith and Genesereth (Smith & Genesereth, 1985) showed that the cheapest first heuristic can easily lead to very expensive orderings.

Also, the example in 6.6.2 of this thesis demonstrates a case where the cheapest first heuristic fails miserably.

Another shortcoming of Chat-80 is that it handles only queries and does not reorder the subgoals in the database rules. This is acceptable under the assumption of Chat-80 that the queries are supplied by naive users, but the database is written by an expert. It is not acceptable under the assumptions underlining this work, that the database itself is written by a naive user.

Warren looks only at the number of solutions (which he calls "cost") without considering the cost of searching the AND-OR tree in order to get those solutions. This would be fine if he assumed (like Smith and Genesereth did) that all the relations are available explicitly in the database. However, he claims that the planning procedure would handle "virtual relations" (relations that are expressed by rules) as well. To be able to handle rules, the program needs to take into consideration the size of the search tree.

The cost of finding a solution by finding an explicit unit clause is assumed by Warren to be constant, based on the full indexing that the Chat-80 system provides. This assumption unnecessarily restricts the generality of the solution.

Another shortcoming of Chat-80 is its source for the number of solutions of instantiated goals. In Chat-80 the information is provided by the system builder, which means that Chat-80 will need an expert to supply the table for number of solutions.

MU-Prolog (Naish, 1985a), unlike Chat-80, does order subgoals within a rule's body. MU-Prolog uses *priority* declarations to control calls to database procedures. Database procedures are procedures that are made

up of a collection of ground, unit clauses. Whenever a goal should be pushed into the computation stack, if the goal happens to be a call to a database procedure, it is added to a separate queue of delayed calls. Whenever all non-database calls have been solved, the delayed calls are ordered using the cheapest first heuristic.

Like Chat-80, MU-Prolog has the shortcoming that the number of solutions for a procedure (which is actually the number of clauses, in the case of database procedures) plus the (independent) probability of a match for each argument are given by *priority* declaration entered by the programmer. Naish mentions that the probabilities can be found "by taking statistics over some period of a typical use," however, there are no further suggestions about how to implement such a mechanism, and what exactly is typical use.

MU-Prolog only orders calls to database procedures, and thus does not need to be concerned with the cost of finding solutions (which is the same as the number of solutions, in the case of database procedures). Naish is aware that the cheapest first heuristic is not optimal, but uses it to choose the next goal. The fact that subgoals from different rules (under the same subtree) can be intermixed makes the list that needs to be ordered potentially very large (the number of leaves in the subtree), thus a hill-climbing search (like cheapest first) is almost a must in order to make the ordering feasible.

Another problem with both Chat-80 and MU-Prolog is that they assume that the probabilities of different arguments matching is independent. Thus they keep N numbers for a procedure with N arguments. This assumption is not very likely to be true in realistic databases. For example, a relation employee(Number,FullName,...) is

likely to have a very similar number of solutions for the three cases where any or both of the first two arguments are bound.

The work of Smith and Genesereth (1985) is by far the most extensive study of the problem of ordering goals. The work described in this chapter was largely inspired by their work (which is part of the MRS project). One major limitation of MRS is that only calls to database procedures are ordered. Being able to order goals that are calls to procedures that involve inferences is a basic requirement for improving the interpreter. MRS performed an ordering which is much more sophisticated than the ordering performed by Chat-80 or MU-Prolog. The ordering is basically found by performing a best-first search in the space of partial sequences of goals.

The advantage of employing such a procedure is that it can find good orderings that could not be otherwise found by the more simplistic heuristics (like the cheapest first). The main problem with using a search procedure is that its cost may outweigh its benefits. MRS uses the adjacency theorem to prune the search, but Section 6.6.2 proves that the theorem does not apply in the general case of non-database calls. Another method that MRS uses to reduce the probability of prohibitively expensive search is *run-time cost monitoring* which works in the following way: problem solving starts without ordering. The cost of solving the problem is monitored. If the cost becomes higher than some preset threshold, the ordering mechanism is triggered. The basic idea is that the system invests resources in hard problems only. The problem with this approach is that if the ordering procedure is triggered, the resources invested up to that point are wasted (since the system will have to restart solving the whole problem). If the threshold is low, the resources wasted will be low, but the

overhead can be too high. If the threshold is set to a high value, ordering will be applied only in hard cases, but the overhead of discovering that the problem is hard will be too high. To summarize, run-time cost monitoring discovers too late that a problem is hard.

Smith and Genesereth's work, like Warren's and Naish's, requires that the user enters the number of solutions required by the ordering procedure. Such a requirement makes the system non-transparent to the user, making it unusable under the condition specified in Section 4.2. LASSY overcomes this limitation by using machine learning techniques to accumulate the domain specific knowledge needed for the reordering. The next section will outline the basic approach taken by LASSY to eliminate most of the problems described above.

6.2. The basic approach

The approach described here develops the basic method of Smith and Genesereth (1985) to include rules as well as ground clauses. Large part of this chapter appears also in (Markovitch & Scott, 1989a). The problem is that the cost and number of solutions of subgoals must be obtained before ordering can be performed. Since the basic assumption is that a novice user/programmer is using the system, the program can not depend on external sources to supply this information and must therefore obtain this knowledge for itself.

The first question to be answered is exactly what information the learning program needs to acquire. Since the ordering program will try to find an ordering with minimal cost, it is clear that we need information that will enable a good estimate of the cost of executing a sequence of

subgoals to be made. Section 6.3 analyzes that cost, and defines a formula that approximates the costs and number of solutions of subgoals by using averages over the subgoal calling patterns.

How is that information to be acquired? The same experiences that were used for lemma learning can also be used for the learning of costs and number of solutions. These experiences are generated by solving practice problem. The learning program collects information about costs and number of solutions while proving the practice queries. The task model biases the experience of the learning system towards those that are likely to be more informative.

After deciding what knowledge the system needs to acquire and how to acquire it, we need to decide how to use that knowledge. LASSY has followed Smith and Genesereth (1985) in performing a search of the space of partial sequences in order to find the optimal one. The main problem with this approach is that an exhaustive search is of exponential complexity, thus performing it may cause the costs of ordering to outweigh its benefits. The solution given here is to perform a resource-bound A* search

6.3. Cost analysis

The main goal of the ordering procedure is to find the best ordering of a given set of conjuncts with respect to some criteria, using the given resources. The cost estimates used to evaluate an ordering should approximate as closely as possible the cost of executing the subgoals in the given order. In this section we will try to analyze the cost of executing a sequence of conjuncts. The analysis builds mainly on the work described in

(Genesereth & Nilsson, 1988; Naish, 1985a; Smith & Genesereth, 1985; Warren, 1981).

Given a goal G and a rule $P \leftarrow P_1,...,P_n$ where P matches G, what is the cost of proving $P'_1,...,P'_n$ (where P'_i is P_i under the binding generated by unifying G and P)? For simplicity, we will follow Smith's (Smith, et al., 1985) simplification of computing the cost for "all solutions" queries.

Define $COST(P_i)$ as the resources needed to compute the whole search tree of the subgoal P_i. Define $SOL(<P_1,...,P_n>)$ as the set of all bindings generated by executing the sequence of subgoals <P1,...,Pn>. Define $P|_b$ to be a subgoal P after substituting its variables according to binding b.

P1 will be invoked only once. P2 will be invoked once for each solution generated by P1. P3 will be invoked once for each solution of <P1,P2> etc., so

$$COST(<P_1,...,P_n>) = COST(P_1) + \sum_{b\in SOL(P_1)} COST(P_2|_b) + ... + \sum_{b\in SOL(P_1,...,P_{n-1})} COST(P_n|_b) =$$

$$= \sum_{i=1}^{n} \sum_{b\in SOL(<P_1,...,P_{i-1}>)} COST(Pi|_b) \qquad [6.1]$$

The problem with formula [6.1] is that the whole rule has to be executed in order to compute it. To solve this problem we will first transform the formula to a form that uses averages rather than the particular values.

$$\sum_{b\in SOL(<P_1,...,P_{i-1}>)} COST(P_i|_b) = |SOL(<P_1,...,P_{i-1}>)| \underset{b\in SOL(<P_1,...,P_{i-1}>)}{MEAN}\left[COST(P_i|_b)\right] \qquad [6.2]$$

$$|SOL(<P_1,...,P_{i-1}>)| = \prod_{j=1}^{i-1} \underset{b\in SOL(<P_1,...,P_{j-1}>)}{MEAN} |SOL(P_j|_b)| \qquad [6.3]$$

Incorporating [6.2] and [6.3] into [6.1] yields:

$$COST(<P_1,...,P_n>) = \sum_{i=1}^{n}\left[\prod_{j=1}^{i-1} \underset{b\in SOL(<P_1,...,P_{j-1}>)}{MEAN} |SOL(P_j|_b)|\right] * \underset{b\in SOL(<P_1,...,P_{i-1}>)}{MEAN}\left[COST(P_i|_b)\right] \qquad [6.4]$$

The next step is to find a good approximation of [6.4] that can be computed without computing the whole rule. Given a subgoal P_i, instead of taking the mean over all the bindings generated by the previous subgoals, we will use the mean over a superset of P_i. The superset will be the *Calling Pattern* for P_i. The calling pattern is defined after (Debray, & Warren, 1988), but we use a binary domain, with the elements *ground* and *nonground*, while Debray and Warren (1988) use a more sophisticated domain with four elements (empty, closed, free and don't know).

Definition: Given a goal $P(t_1,...,t_n)$ where $t_1,..,t_n$ are already substituted under the current binding, the calling pattern for that goal is $P(C(t_1),...,C(t_n))$ where $C(t_i) \in \{0,1\}$. $C(t_i)=1$ iff t_i is a ground term under the current substitution. Thus, for example, the calling pattern of the goal parent(john,Y) is parent(1,0).

As mentioned by Debray and Warren (1988), the reason for using such patterns is to approximate the unbounded number of literals that may appear while proving a query, by a finite set of equivalence classes. Having such a finite set will enable us to collect various statistics needed for the ordering process (for example, cost and number of solutions). We do not claim that calling patterns are the best partition possible, and further research should be done to explore the possibility of more sophisticated classification of the arguments (possibly using semantic information).

Given a database with M predicates, the number of possible calling patterns is

$$\sum_{i=1}^{M} 2^{\,ARITY(Pi)}$$

The learning program will consider only a small subset of the calling patterns - those encountered during the training phase.

Let P be a goal and $<g_1,...,g_k>$ a sequence of goals. Let $CP(P,<g_1,...,g_k>)$ be the calling pattern of P assuming all variables of P which appear in $<g_1,...,g_k>$ are bound. Let COST(cp) and NSOLS(cp) be the means for the cost and number of solutions for the calling pattern cp computed after solving a set of problems. The modified formula for the approximated cost is:

$$COST(<P_1,...,P_n>)=\sum_{i=1}^{n}\left[\prod_{j=1}^{i-1}NSOLS(CP(P_j,<P_1,...,P_{j-1}>))\right]*COST(CP(P_i,<P_1,...,P_{i-1}>)) \qquad [6.5]$$

The inductive learning procedure, described in the next section, builds a table of costs and number of solutions for the calling patterns. The ordering procedure, described in Section 6.6 uses this table to estimate the cost of subgoal sequences using formula [6.5].

6.4. Learning average costs and number of solutions

Most of the cost and number of solutions information of ground predicates (predicates whose clauses are all ground) can be retrieved by scanning the database and counting. Thus, before the learning program is engaged in its learning-by-doing process, it accumulates the statistics about the ground clauses - their number, and the sizes of the domains of their arguments. We term this process *static learning*. Averages that can not be collected by the static learning process are acquired through a *dynamic learning* phase.

LASSY performs dynamic learning by continuously generating queries using the task model and updating the inductive knowledge base while solving the queries. The inductive knowledge base is a table with an entry for each predicate. Each entry contains average information for the

predicate regardless of the calling pattern, and an entry for each calling pattern encountered during learning. Each such entry contains counters for the number of times that a subgoal matching the calling pattern was invoked, for the total cost of executing the subgoal, and for the total number of solutions, so that the average information can always be retrieved from the table.

When the interpreter is working under learning system control, it updates the entries in the table. Whenever a subgoal is called, the appropriate entry for the calling pattern is created if necessary, and the calling counter is incremented. Whenever a solution is generated, the solution counter is incremented. Whenever backtracking from a subgoal occurs, the subgoal tree's cost is added to the total cost. The cost of searching the tree of a subgoal is computed in the following way: whenever the interpreter creates a node, its cost is set to zero. Whenever the interpreter tries to unify a goal with a head of a clause, the costs of all the nodes in the path to the root are incremented by 1. Whenever a new subgoal is pushed onto the computation stack, the number-of-solutions counter of the former subgoal is incremented by 1. An entry in the table may look like: ((**ancestor** 0 1) 27 93 16782) which means that a subgoal with the predicate **ancestor**, with the first argument unbound and the second argument bound, was invoked 27 times, has generated 93 solutions and had a total cost of 16782 unifications. Thus the average cost for that calling pattern is 621.55 unifications and the average number of solutions is 3.44.

6.5. Handling insufficient knowledge

When the ordering procedure requests an estimate for the costs or number of solutions of a particular calling pattern, the requested information is retrieved from the inductive database. The retrieval is fast since all the information is stored in hash arrays. But what happens if the learner did not yet have experience with the particular calling pattern? In such a case, a set of heuristics is used to estimate the costs or number of solutions, based on the existing information:

1. If the predicate is a ground predicate (there are no rules for the predicate in the database), and none of the arguments in the calling pattern is bound (all zeros), then the number of solutions and the cost are the number of clauses for this predicate in the database.
2. An upper bound for the number of solutions and costs is searched for. If information is available for a more *general* calling pattern, then this information is used as an upper bound. Calling pattern A is more general than calling pattern B if they have the same predicate and all the bits of calling pattern A are less than or equal to the corresponding bits in calling pattern B. In other words, for every unbound variable in B, there corresponds an unbound variable in A and for every bound variable (or ground term) in B, there corresponds a bound or unbound variable in A. A subgoal that is represented by a less general calling pattern can not have more solutions than a subgoal represented by the more general calling pattern. The same is true regarding costs.
3. If the predicate is ground then the number of clauses in the database is an upper bound on the number of solutions (and costs).

4. A lower bound is searched for in the same manner, using information about less general calling patterns. If a lower bound is not available, it is set to 1.

5. The lower bound is set to be the maximum of all the lower bounds found. The upper bound is set to be the minimum of all upper bounds found. If both are available, then the procedure returns the mean of the lower and upper bounds.

6. If bounds are not available, averages for the predicate are used. The learning procedure keeps account for the average number of solutions and average costs for the predicate regardless of the calling pattern. If such an average is available, it is returned as the average for the particular calling pattern.

7. If no information whatsoever is available for the particular predicate, the very rough estimate of the number of clauses for the particular predicate in the database is returned (as suggested in (Kowalski, 1979)).

The estimates given by the above heuristics are not likely to be very accurate; however, the heuristics will have to be used only when the system does not yet have enough experience. After engaging in enough learning, the system is likely to have experience with all calling patterns, and the above heuristics will not be used.

6.6. Use of the learned knowledge by the ordering procedure

Finding optimal ordering is generally an NP complete problem. An exhaustive search will involve evaluation of N! sequences for N subgoals in the worst case. The work described here deals only with the ordering of subgoals within a rule's body. A more complete system could allow the

ordering of all current subgoals within the computation stack. However, the number of possible orderings would be so high that it would not be feasible. In the next section we will give a precise definition of the space searched for an optimal ordering. The states are the partial sequences, and the final states are complete sequences.

6.6.1. The search space definition

Given a set of subgoals $P = \{P_1,...,P_n\}$, we can define a search space $SP = <S,O,S_i,S_f,g>$ where:

$S = \{<P_{j_1},..,P_{j_k}> \mid \{P_{j_1},..,P_{j_k}\} \subseteq P\}$ is the set of states.

$O:S \rightarrow 2^S$ is the operator.that generates successor states and is defined as

$O(<P_{j_1},..,P_{j_k}>) = \{<P_{j_1},..,P_{j_k},P_t> \mid P_t \in P - \{P_{j_1},..,P_{j_k}\}\}$

$S_i = <>$ is the initial state

$S_f = \{<P_{j_1},..,P_{j_k}> \mid \{P_{j_1},..,P_{j_k}\} = P\}$ is the set of final states.

$g(s_1,s_2) = NSOL(s_1) * COST(s_2)$ is the cost of moving from state s_1 to state $s_2 \in O(s_1)$.

Given such a space, we can perform an A* search to try and find the path with minimal cost. We need to associate with each state the g value (which is a requirement of the A* algorithm), the number of solutions for the subsequence, and the binding pattern for the subsequence (i.e. which of the clause's variables is bound). Thus, when applying the operator to get the successor states, we have to calculate the new g, the new number of solutions, and the new binding pattern.

6.6.2. Pruning the search

The problem with an exhaustive search is that in the worst case, it can expand n! nodes. If the search procedure reaches two states that consist of exactly the same set of subgoals (but in different order, otherwise it would be considered the same state), the state with the higher g value can

be removed from the set of states that are candidates for expansion, since there is no way that the state with the higher value can be a part of the optimal ordering. That prunes the number of states that can be expanded to 2^n in the worst case.

Smith and Genesereth (1985) proved the Adjacency Theorem which reduces substantially the space of possible orderings that should be searched to find an optimal ordering. The theorem was proved in the context of optimizing a conjunctive query in a database consisting entirely of positive ground literals (this type of problem is called "Sequential constraint satisfaction" in (Genesereth & Nilsson, 1988)).

An interesting question is whether the Adjacency Theorem applies also in the case where an execution of a subgoal can carry an arbitrary cost. Unfortunately, the answer is no. To prove this, we will show a counter-example to one of the corollaries of the theorem. The corollary says:

> Given a conjunct sequence of length two, the less expensive conjunct should always be done first (Genesereth & Nilsson, 1988).

The reader should note that "less expensive" in this context means "has a smaller number of solutions." Assume that we have a set of two conjuncts {P1,P2} that we want to order. For simplicity, assume that P1 and P2 do not share variables. Assume that P1 has 100 solutions, and has a search tree that costs 10000. Assume that P2 has 2 solutions and has a search tree that costs 10.

COST(<P1,P2>) = 10000 + 100*10 =11000

COST(<P2,P1>) = 10 + 2 * 10000 = 20010

Thus we have a case where the most expensive conjunct should be done first.

In practical problem spaces the number of nodes expanded is much smaller than 2^n. Primarily, this is because there is a big difference between the optimal ordering and many non-optimal orderings: there are many cases where the set of subsequences that are more expensive than the complete optimal sequence is large. The search procedure will never expand such states. The fact that many times n is small also helps to reduce the size of the search.

6.6.3. Heuristics

Finding a good heuristic function for evaluating the minimum cost of moving from the current state to the final state (the h component in the A* terminology) could make the search much faster. The problem is that such a heuristic is hard to find.

Using the obvious transformation to the Traveling Salesman Problem, each city will be mapped to a subgoal and the task of finding a path with minimal cost between the cities mapped into the task of finding an ordering with minimal cost. Several heuristics have been developed for solving TSP (Pearl, 1984). However, there is one difference that makes the ordering problem "harder" than the TSP - the cost of getting from one "city" to another can only be known when visiting that "city," and it depends on the path that was used to get there. That difficulty makes most of the heuristics used in the TSP useless for a search for optimal ordering.

We have implemented the following simple heuristic: given a state s,

$$h(s)=NSOLS(s) * \underset{P_i \in P - s}{MIN} COST(CP(P_i,s))$$

The above heuristic function is an underestimate of the actual cost. One could claim that this is an advantage, since A* is admissible when h gives an underestimate (Nilsson, 1980). However, the size of the search

space suggests that admissibility is less important than the reduction of search time. The problem with the function h defined above is that it is not a very "informed" one. Nilsson (1980) defines the relation "more informed" between two heuristics functions - function h1 is more informed than function h2 if h1 always gives a better estimate than h2. However, even after relaxing the admissibility constraint, we could not develop a better heuristic.

6.7. Selective utilization when reordering

Although we have claimed that the time that will be spent on the search will be much lower than the worst case, we need something more substantial to make the ordering worthwhile. The whole idea of the ordering was to invest time in the ordering process itself, in order to save greater time during the execution of the goal. If we don't have any reasonable bound on the search time, how can we make sure that we do not actually lose by performing the ordering?

As mentioned in Section 6.1, Smith and Genesereth (1985) suggested run-time cost monitoring to find out when a problem is hard enough to justify the cost of reordering. The method does not perform reordering unless the resources spent by the problem solver became larger than some threshold. The problem with that method is that all the resources spent up to the point when the difficulty of the problem was discovered are wasted, and add to the total cost of solving the problem. If the threshold is low, then the amount of wasted resources will be small, but reordering will be performed on easy problems; thus the cost of the ordering will outweigh its

benefits. If the threshold is high, then reordering will be performed only on hard problem, but the amount of wasted resources will be large.

Since LASSY is a learning system, it can use its experience to decide whether a problem is likely to be difficult. In fact, the average costs of solving goals is accumulated by the system anyway for the needs of the reordering procedure, and can be used to estimate the difficulty of problems. The simplest way of using this information is by employing a threshold procedure similar to Smith and Genesereth's, but without the cost of discovering that the problem is hard.

Such a procedure is an example of selective utilization. The program uses the learned knowledge (average costs and average number of solutions) only when it is likely to be beneficial. Such a filter is binary, i.e. it either allows or does not allow a usage of knowledge. LASSY uses a more continuous type of utilization selector. The selector allows usage of learned knowledge up to a certain point, where it estimates that more usage will make the knowledge harmful.

The filter works by making the A* procedure a resource-bound search procedure where the bound is the expected cost of the parent goal. There is a minor problem with using the expected cost. Since LASSY continuously modifies the averages as learning proceeds, the ordering will cause a great reduction in the expected cost of the goal whose subgoals were reordered. The utilization filter will turn reordering off for that type of goal, since it will estimate that the cost is low. Without reordering, the cost will be very high, and thus the averages will climb up again, until they reach the point where it is worthwhile to perform reordering. This process of oscillation will continue forever. To avoid this, LASSY stores, in addition to the expected costs of a subgoal, also the maximum expected costs

encountered. This maximum is considered as the worst case and is used as the bound for the search.

A minor problem with using the costs is that LASSY uses the number of unifications as its basic cost unit. The natural unit for measuring the costs of a search procedure is the number of nodes expanded. To reconcile the two types of measurement, a conversion is done before the resource limit is given to the search procedure. Currently, the converter uses a fixed conversion rate that was established by empirical experimentation. In a non-experimental system, all costs could be measured in cpu time.

The search procedure will be given the amount of resources it is allowed to use as a parameter. The unit for specifying the resources is number of nodes expanded. Given a set of subgoals of size N and a limit of M resources, the search procedure will conduct an A* search until either a solution is found or the number of nodes expanded is M - (N - Length(s)), where s is the current best node, and length(s) is the number of elements in the subsequence of s. The system then proceeds with a hill climbing search - that is, it expands the best node, then the best of its successors, and so on. If we use the g function as our evaluation function, then the hill-climbing search is similar to the cheapest first heuristic (except that here we mean cheapest in terms of cost, not number of solutions). The ordering procedure also subtract a fixed amount of resources from its given limit to account for the fixed overhead of using the procedure.

The ordering procedure caches the ordering results, thus avoiding the need to search again for the best ordering of the same rule with the same calling pattern. The cache is erased whenever any learning occurs, because it may not be valid any more.

6.8. Experimental results

The database used for the experiments with the ordering procedure is the same as that used for the lemma learning experiments. The task domain was different than the one selected for the lemma experiments. The reason for choosing a different task domain is that the rule for the relation *linked* is recursive, and hence its body is enclosed within <,>, and no ordering can be performed. The task model used here includes queries that call procedures whose rules can be reordered. The only type of learning that was used in these experiments is the inductive learning described here. All other learning mechanisms were turned off for the whole duration of the experiments.

The next experiment was conducted in order to determine whether the knowledge learned and the ordering procedure are effective. The experiment consisted of the following steps:

1. A set of 20 random queries was generated using the task model; this is the test set.
2. With all learning turned off, ordering turned off, and using random ordering, the system executed the test set and the performance of the system was recorded.
3. The system performed the static learning - i.e., it scanned the databases to collect information about ground clauses.
4. With learning turned off and ordering turned on, the system performed the test set. In the graph, this point is marked as the test for 0 learning examples.
5. With learning turned on and ordering turned on, the system generated 10 random queries using the same model and executed them.

6. With learning turned off and ordering turned on, the system performed the test set.
7. Steps 5 and 6 were repeated 5 times.

The results of the experiment are summarized in Figure 19. The Y axis specifies the mean number of unifications taken over the results of the test (i.e. 20 queries). The first observation that we can make is that the best performance (after learning 15 examples) is 23 times faster than the performance with no learning at all. A second observation is that the performance after executing 15 learning tasks is 14 times faster than the performance with ordering that used only the knowledge gathered during the phase of static learning (i.e. the statistics about the ground clauses). The last observation that we make is that the LASSY did not improve its performance while performing the last 10 learning tasks. The reason for this flattening is that after performing 15 examples, the averages approached their final values, thus the ordering procedure always produced the same output.

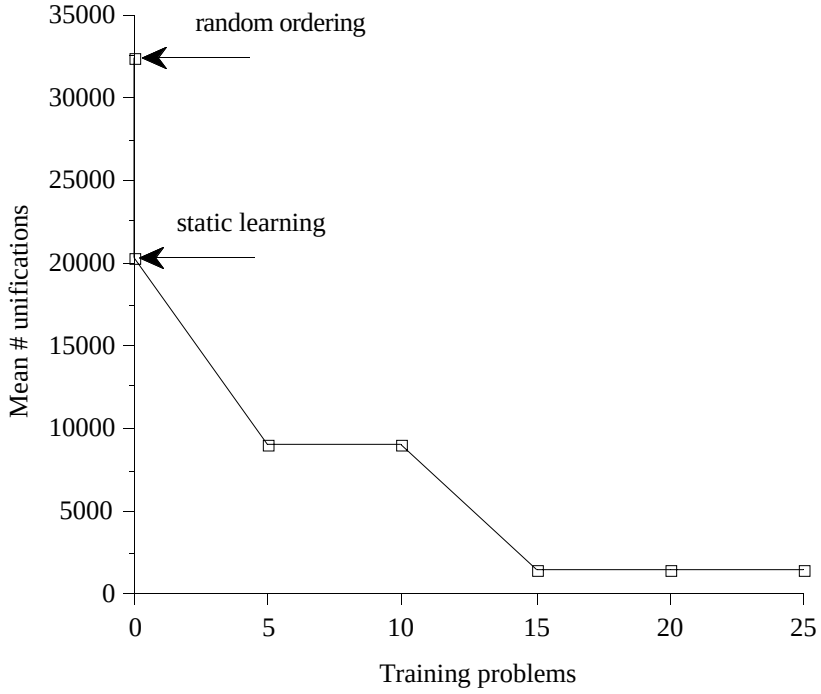

Figure 19. Learning curve while acquiring averages of costs and number of solutions for the reordering procedure. Static learning means that information was gathered from the database without solving training problems yet.

One final comment about the experiment. To finish the experiment in a feasible time, a limit of 50,000 unifications per query execution was set. During the performance of the test with random ordering, the execution of half of the queries was halted after 50000 unifications. Thus, the actual improvement in performance should be much higher than 23 times.

6.9. Combining ordering with lemma learning

Until now the deductive and inductive parts of LASSY were tried separately to allow us better understanding of the learning procedures and the selection processes. In this section we will describe an experiment

where all the learning and selection mechanisms of LASSY were active. The results are shown in Figure 20.

There are several features of the graph that are worth attention. First, it looks as if during the first part of the learning session, the lemma learner actually slowed the system progress. The reason for that phenomenon is the presence of so many adaptive parameters in the system that depend on each other. The probability thresholds, the utility measurements, the costs and number of solutions are all adjusted constantly during learning. Thus, it is possible, for example, that after learning 5 training examples many lemmas were acquired, but the threshold probabilities were not yet approaching their final values.

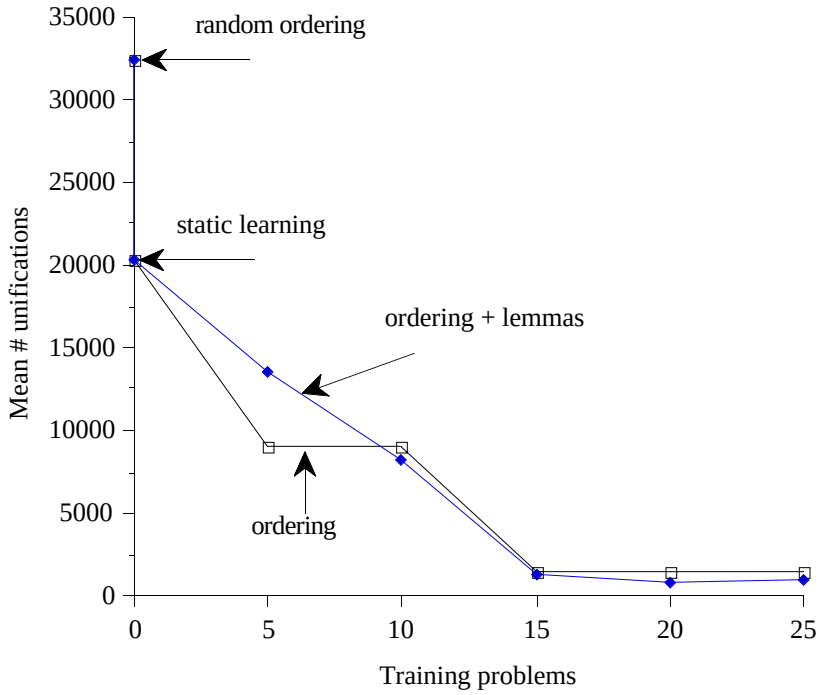

Figure 20. Performance with ordering and lemmas vs. performance with ordering and no lemmas.

Table 5. Performance with ordering and no lemmas vs. performance with ordering and lemmas

Training problems	Order, no lemmas	Order, use lemmas
(random) 0	32455	32455
(static) 0	20262	20262
5	8975	13501
10	8980	8283
15	1424	1333
20	1425	884
25	1425	889

Although, because of the graph scale, looking at the graph only, it looks as lemma learning did not improve the performance of the system, Table 5 which describes the graph, reveals that performance with lemmas and reordering was 40% better than performance without lemmas. The third interesting observation is that it looks as if the ordering procedure is much more effective than lemma learning in improving the efficiency of the interpreter. The performance with both types of learning improved the system's performance by a factor of 36.

6.10. Selective experience

The average costs and number of solutions are accumulated for the calling patterns rather than for each possible subgoal. This reduces the number of possible entries in the learned knowledge base to be manageable.

However, in realistic data bases the number of possible calling patterns will still be high. To utilize learning resources efficiently, LASSY incorporates selective experience to bias the system's experiences towards those that are more likely to make the learner produce relevant knowledge.

6.10.1. Task Modeling for selective experience

One of the parameters of the function for evaluating the value of knowledge (see Chapter 2) is the set of problems for which the knowledge is been used. It is thus essential that a selective learning system will determine what is the set of problems with respect to which it wants to improve its performance. Almost always, this set is the set of tasks that the system will receive in the future. Since the system usually has no way to *know* what its set of future tasks is, most learning systems include a model of the set of tasks that the system will receive in the future. Such a model will be called here a *task model*. In most existing learning systems, the task model is not explicitly represented, but implicitly embedded in the system architecture. By far the most common task model is that which assumes that the tasks given to the system are drawn from a set with a fixed distribution, and hence that the set of future tasks will be similar to the set of past tasks. This assumption is explicitly stated as part of the Valiant model of PAC learning (Kearns, et al., 1987; Valiant, 1984):

> "...the rule found by the algorithm will be able to predict future events *drawn from the same distribution* on which it has learned (Kearns, et al., 1987)."

The fixed distribution assumption is simple, but not necessarily realistic. If in his first day of his job, a taxi driver gets assignments within a radius of 1 mile of his station, on his second day assignments between 1 and 2 miles away from his station, and on his third day assignment which are between 2 to 3 miles away from his station, a reasonable model will

predict that the next day, his assignments will be between 3 to 4 miles from his station. The problem of creating a model of the future tasks is an instance of the prediction problem which is inherently difficult.

A useful extension to the fixed distribution model is to allow the fixed distribution to be defined over a set of classes which partitions the task space rather than on the set of individual tasks. This extension allows the system to gain experience with problems that are similar in some aspects to past problems, but different in others.

When the system acquires a model of its task future, it can use the model for several purposes. For example, a learning system can use the model to monitor its own performance. A set of test problems can be generated using the model and the system can perform the test periodically with learning turned off, in order to monitor its own performance. The system could then change its learning strategy based on its own learning curve. The capability to perform self tests is built into LASSY, but currently LASSY does not utilize that capability.

Another use of the task model is for generating training problems for learning systems which perform "learning by experimentation" or "learning by doing." If we assume fixed distribution, why can't the system just use the past problems as training problems? It can, if it got enough of them. An off-line learner can exhaust the set of past problems. In such a case, the task model can be used to generate useful training tasks. Assuming the extension to the fixed distribution model, the tasks that the system generate will be different in some aspects those it has encountered in the past, allowing the system to explore new areas of the task space.

Section 4.4 includes a description of the experience filter and the method in which it is acquired and used. Basically, LASSY creates or

updates the task model based on the set of past queries given to the system by external users. Every query is mapped into its calling pattern, and weight for the particular calling pattern is increased. Chapter 7 includes a discussion of possible ways to make the task model builder of LASSY more powerful and more realistic.

6.10.2. Experimenting with the experience filter

The next experiment was conducted in order to test whether the experience filter was useful. The former experiment was repeated, but the filter was turned off (i.e., the system generated random problems for training). The results are shown in Figure 21.

As expected, learning with the experience filter is more efficient than

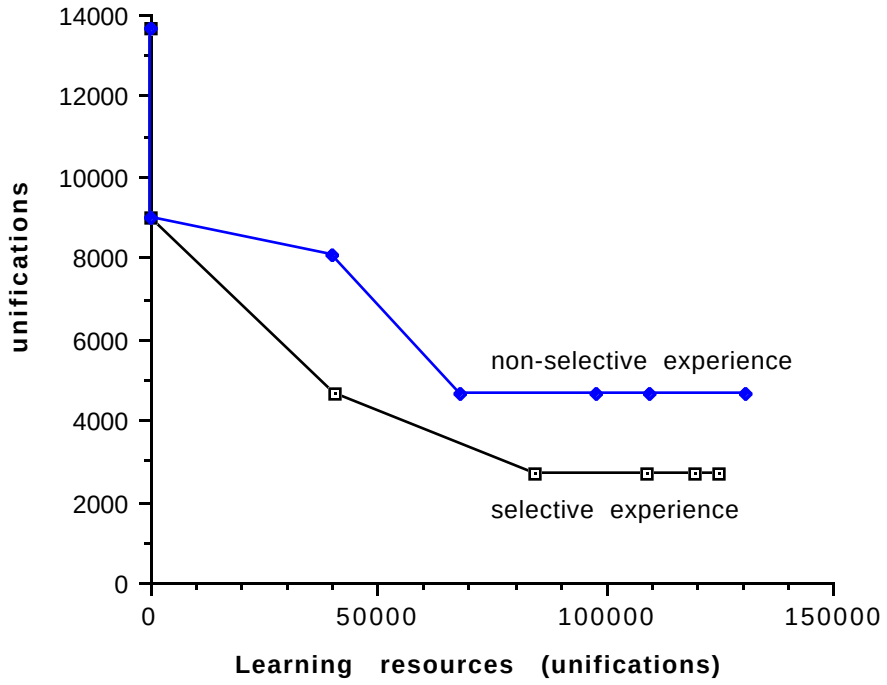

Figure 21. Learning curve while using task model vs. learning curve with no task model.

learning without it. The performance of tasks dissimilar to those in the task domain made the system acquire irrelevant knowledge. A more realistic database would show even a greater difference. One problem with the database used for the experiments is that the calling hierarchy of its procedures is a tree, or mostly a tree. That means that execution of almost any query will generate potentially relevant knowledge. In a more realistic database, the calling hierarchy will be a forest. In such databases, solving queries in one tree will generate knowledge which is irrelevant to other trees.

6.11. Summary

One of the methods by which LASSY improves the efficiency of the Prolog interpreter is by reordering subgoals. Since the goal of the system is to minimize the cost of executing a conjunction of subgoals, a method for estimating the cost of executing such a conjunction was given. The basic terms in the formula developed were the average number of solutions and the average costs of calling patterns of subgoals.

The learning procedure developed to acquire that information uses the same experiences as the deductive learning procedure (the lemma learner). The averages are computed while proving queries that are generated by the task generator. The queries are similar to queries that were given to the problem solver by external agents in the past, thus they are likely to be similar to problems that will be given in the future (assuming a fixed distribution).

The learned knowledge is used by an ordering procedure, which performs a search in the space of partial sequences in order to find a good

ordering. The method used in LASSY is a resource bound A* which reduces the probability that the search cost outweighs its savings. Experiments performed with the inductive learner showed promising results. Performance was 23 times better with the ordering than performance with random ordering.

In the introduction to this chapter, the shortcomings of three existing systems that reorder subgoals were addressed. Does LASSY overcome these limitations?

1. LASSY acquires the averages for the costs and number of solutions automatically, thus avoiding the need for the user to supply this information.
2. LASSY does not use the cheapest first heuristic which can lead to poor ordering, but rather employs a search in the space of subsequences as did Smith in MRS.
3. LASSY reorders rules bodies and does not restrict itself to the top level queries.
4. LASSY orders conjuncts that involve inference and does not limit itself to conjuncts of database procedures. LASSY is the only system that does so, because its learning component acquires the averages for the costs of goals that involve inference. The other systems are restricted because the cost information is not available to them.
5. LASSY does not assume that the probabilities of the different arguments of a relation matching is independent, thus its representation allows it to keep up to 2^N for a relation with arity N. Of course, most of the time the number of entries will be much smaller because many combinations will not be encountered during learning.

6. LASSY uses selective utilization to limit the resources that can be spent on the search. This method uses past experience to decide how hard problems are and does not suffer from the deficiency of the run-time cost monitoring method, which wastes the resources that are are spent before it discovers that a problem is hard. LASSY's methods is a generalization of the run-time cost monitoring. Whereas the method used by Smith sets two thresholds to decide whether to perform ordering at all, and if yes, whether to perform search at all, LASSY's method of performing a resource bound search can be seen as using "dynamic threshold."

7. FUTURE RESEARCH

LASSY is a large-scale project, and there is great scope for future research with the system. This chapter will outline several extensions and additions that are planned for the future.

7.1. Additional experimentation with the current implementation

LASSY includes several learning and selection mechanisms as well as several parameters. The number of possible configurations is exponential to the number of parameters, thus most of the experiments described in this thesis are performed when only one learning or selection mechanism is active. A systematic analysis of all the interesting combinations of parameters should be done, and a systematic set of experiments should be conducted to carry out this plan.

All the experiments done with LASSY used the computer network database. In the future, more databases are needed, to test how well the findings reported here hold up with the new databases. In addition, it would be useful to identify what features of the databases change the behavior of the system. For example, the task model will probably prove to be more useful in databases whose calling graphs are forests. A good algorithm for generating artificial databases can be useful in creating databases with the desired characteristics.

7.2. Better task model

The task model implemented in LASSY is rather limited. The basic assumption behind the task modeling module of LASSY is that the task domain can be expressed as a distribution over a set of classes of tasks. The problem with LASSY's current task model is that the language that it uses to classify tasks is limited. Currently LASSY uses calling patterns to classify the tasks. The next extension (implemented, but not tested) will be able to create separate classes for queries that succeed and queries that fail. The only difficulty with implementing this feature is the generation of queries belonging to those classes. It is not difficult to generate queries that succeed (although it may require enormous learning resources). The experience generator can employ the interpreter by calling the predicate with all variable unbound (learning should be turned off). Then it can record the returned bindings and use them for queries that should succeed. The problem is to generate queries that fail. At this time I cannot find a feasible solution to that problem.

Currently, LASSY ignores the particular constants that are given as arguments. It is only interested in whether an argument is a variable of a constant[1]. An important extension to the task model will be to add distribution over the domains of predicates. Whenever a constant appears as the Kth argument to a predicate in a query, its weight within the domain of the Kth argument will be incremented. Whenever the task generator generates a problem, it will select constants with probability proportional to their weight. This extension is easy to implement. The problem with this

[1] The constants are collected to create the domain sets.

approach is that it assumes that the probabilities of constants appearing as arguments is independent.

To make the classification of tasks even more powerful, we can try to find semantic properties associated with constants for particular arguments. For example, it is reasonable that all (or most of) the queries of the form father(1,0) (i.e. queries that look for the children of somebody) will be called, with the constant that stands for the first argument satisfying the predicate male(X). This is a difficult addition to implement, since the problem is a full classification problem. It may be feasible with enough assumptions and simplifications.

In (Scott & Markovitch, 1989b) it was argued that one of the reasons for having the learning system generate its own experiences is that generation of informative experience requires knowledge of the current state of the system's own representation (which is not available to an external agent). The current scheme of generating training problems does not take advantage of this knowledge. The system will devote a large part of its learning resources to a class of problems that is given very often by external users even if it already knows everything that needs to be known about solving such problems. A future implementation of the task generator will also make use of an internal model of the system. The system will keep a test set drawn from its task model. Occasionally, the system will perform tests and update its performance graphs for all the problem classes. Flattening of the graph will indicate that the averages used for this class of problem have approached their final values, and the system will devote more resources to other problem classes. A possible problem that can be a result of such an addition is that averages that the

system collects will not be correct, since they will be biased by the system's current state.

7.3. Computational value depends on the pattern of caller

Selective acquisition and selective utilization use the estimated resources that an application of a lemma saves, as part of their heuristic to estimate the value (utility) of that lemma. The method for estimating the computational value of a lemma is to look at the size of the subtree under the subgoal that was used to generate the lemma. There is one problem with that approach - it does not take into account the bindings of the calling subgoal. If the system learns a lemma p(c1,c2), the lemma may be used to prove a subgoal of the form p(c1,c2), or it may be used for proving a subgoal of the form p(X,Y); and it is likely that there is a large difference in the costs of solving these two types of problems. That means that the computational value that was recorded when the lemma was learned is a good estimate only for cases where the lemma is used by a subgoal with the same calling pattern. How to find a better estimate for the computational value is an open question.

7.4. Generalizing lemmas (EBG style)

One extension to the lemma learner of LASSY that may be interesting to implement is incorporation of EBG-style rule learning, similar to the method used by PROLEARN (Prieditis & Mostow, 1987). In Chapter 5 it was shown that the costs associated with using generalized lemmas is much higher than the costs associated with unit clause

lemmas. However, this may be a challenging problem to the information filtering framework: to be able to make the value of the generalized lemmas positive.

7.5. Replacing the threshold by cost evaluation

The threshold method used for selective utilization of lemmas is rather primitive. A more sophisticated method will consider the expected benefit from using lemmas in case of a success, and the expected cost of using lemmas in case of failure. In order to use lemmas, the probability of failing multiplied by the expected cost should be lower than the probability of succeeding multiplied by the expected gain.

7.6. Truth maintenance

The current version of LASSY does not include any form of truth maintenance (Doyle, 1979). This omission makes the lemma learner of LASSY hardly usable for any realistic database which is modified often. One of the first extensions that is planned for LASSY is a simple truth maintenance system. LASSY will hold a table with an entry for each clause in the database. Whenever a lemma is learned, LASSY will add a pointer to the lemma from each clause that is part of the proof of the lemma. Whenever the database is loaded, LASSY will scan the table, and for each clause that was removed from the database, all the lemmas supported by the clause will be deleted from the lemma database.

7.7. Interfacing LASSY with other PROLOG systems.

Another feature that I would like to add to the system is the ability to use its learned knowledge with other Prolog systems. The idea is to write a procedure that will output a transformation of the POST-Prolog program that will be executable in other Prolog environments. The procedure will order the expensive rules for various calling patterns, and will output a set of output rules for each input rule. The output rules will use the standard meta-logical system predicates *var* and *novar* to make the interpreter execute the rule with the right order.

7.8. Negative lemmas

LASSY includes a module that learns and uses negative lemmas (Kowalski, 1979). However, since few experiments have been done using this module, a description of it is included here, in the chapter of future research. Negative lemmas are subgoals that failed. The idea is that if a subgoal fails, it can be worthwhile to remember it to avoid trying to prove the subgoal again. Whenever the interpreter exits an OR node with a failure (i.e., all the alternative were tried, and none of the alternatives tried returned successfully), the negative lemma learner adds the subgoal with negation to the negative lemma database. When the interpreter tries to prove a subgoal, it first uses the negative lemma database, then the positive lemma database and then the axioms.

One problem with using negated lemmas is that it will cause incorrect results if used for proving subgoals that are more general than the lemma. Assume that the learner has learned that John is not an

ancestor of Mary. This will be recorded as **not(ancestor(john,mary))**. Assume that the current goal is ancestor(X,mary). The goal will be unified with the negative lemma and the interpreter will conclude that Mary has no ancestor. To overcome this problem, the interpreter was modified to handle negative lemmas differently. A unification of a subgoal with a negative lemma is considered successful if and only if the lemma is at least as general as the subgoal, i.e., no variable in the subgoal is bound to a non-variable in the lemma as a result of the unification.

There are several potential problems with learning and using negative lemmas. First, there are usually many more negative lemmas than positive lemmas, thus the problems with excessive learned knowledge become more serious. Second, negative lemmas are extremely sensitive to modifications in the database. Any addition of clauses to the database may invalidate any negative lemma.

8. DISCUSSION

The objective of this thesis was to study the problem of excessive knowledge in learning systems. Chapter 1 specifies three goals that the thesis attempts to achieve:

1. To analyze and understand the problems associated with harmful and excessive information .
2. To set up a framework for eliminating/reducing harmful and excessive information in learning systems.
3. To test the framework in the context of a realistic and useful learning system.

Were these goals achieved?

The first goal was addressed in Chapter 2 and in part of Chapter 5. After setting the scope for the discussion on harmful knowledge, it was possible to get a functional definition for the value of knowledge element and harmful knowledge. Basically, a set of knowledge elements is harmful with respect to a specific set of tasks if the performance of the set of tasks would improve according to a given criterion by removing the set from the knowledge base. Three features of knowledge were identified as associated with harmfulness: incorrectness, irrelevancy and redundancy. An analysis of costs and benefits of knowledge was done with respect to search spaces. While most preceding works were concerned mainly with costs associated with matching, the author (as well as some other researchers) believe that the most substantial costs are associated with the increase in

the branching factor of the search space. A new type of harmful knowledge, which has not addressed in other works, was described in Chapter 5. The problem occurs when a backtracking problem solver uses deductively learned knowledge (lemma, macros) in a failure branch of the search tree. The backtracking mechanism forces the problem solver to explore the whole failure branch, visiting the same node twice (or generating the same binding twice in the case of lemmas) - once by using the learned macro, and the other time by using the rules that were used to generate the macro in the first place. This is an example of redundant knowledge that is harmful. What differentiate it from the other types of harmful knowledge is that while the other types of harmfulness may or may not occur depending on the circumstances, the backtracking anomaly is *bound* to occur in failure branches.

After analyzing the problem, the second goal of the thesis was addressed: defining a framework for reducing harmful knowledge. The model proposed was the information filtering model which was described in Chapter 3. This model views learning programs as information processing systems where information flows from the experience space that the learner is facing through the attention procedure, the acquisition procedure, via the learned knowledge base to the problem solver. The model introduced 5 types of selection processes that can be inserted within the information flow: selective experience, selective attention, selective acquisition, selective retention and selective utilization.

The information filtering model was proposed as a unifying framework. To support this argument, many existing learning systems were described in the context of the model and indeed, the model incorporated many existing systems that were not hitherto considered to be

related before. It is clear that the framework is useful in making machine learning research more structured.

The third goal was to test the information filtering framework in the context of a realistic and useful learning system. This goal was mostly achieved, but as Chapter 7 indicates, there is still much to be done. LASSY is a complex system whose goal is to improve the performance of a Prolog interpreter. In the next paragraphs, two questions will be addressed: how realistic and useful is LASSY, and what contribution it makes to the understanding of the information filtering framework.

To answer the first question, imagine the following scenario for LASSY use:

"...LASSY is installed on a computer and introduced to the users as a regular PROLOG system. When a user runs the interpreter and loads a database named DB, LASSY remembers the location of the database in the file system, and creates for itself a file named DB.learned. All the queries that are entered by the user are stored in the DB.learned. The user is a novice, without much knowledge of Prolog, however he/she has good basic understanding of logic. The user is a bit annoyed, since it takes so much time to get answers to the submitted queries. As soon as the computer is idle (at night, for example) it runs LASSY in its learning mode. LASSY queues the list of databases that it has to learn and starts learning by solving its self-generated problems. All the learned knowledge is stored in DB.learned. The next morning, the user comes again, loads DB again, and submits more queries. This time, much to the user's surprise, the computer supplies the answers to the queries much faster..."

This scenario is realistic. The experiments described in Chapters 5 and 6 showed that LASSY's performance was substantially improved after

learning. LASSY is totally autonomous - it does not need any information from the user. There are, however, some limitations that should not be ignored. LASSY is not designed to improve the performance of PROLOG when used as a programming language, but rather for improving its performance when used to describe deductive/intelligent databases. A sophisticated PROLOG program which make heavy use of cuts and i/o is not a good candidate for being learned by LASSY. With these limitations in mind, LASSY is the first system that offers a realistic solution to the problem that PROLOG is rendered unusable by novice users by its requirement that the user should supply the control.

What contribution does LASSY make to the understanding of the information filtering model? LASSY is probably the first system that was built with the information filtering model in mind. Thus, its architecture can be used as an example of how to design selective learning systems. Naturally, the deductive learner of LASSY needed to be more selective because there is such a large amount of deductive knowledge available. Implementing the selection processes in LASSY revealed the backtracking anomaly as an important problem that must be addressed. The exposure of this problem is especially significant in the face of the growing interest of the EBL/EBG community in the utility problem. The author feels that many cases of harmful knowledge that were attributed to the cost of matching were actually caused by the backtracking anomaly. LASSY also contributed to the understanding of the information filtering model by implementing the first and last filters in the cascade. While selective experience was implemented by other systems (see Chapter 3), LASSY uses the novel idea of task models in order to generate experiences that are similar to those encountered in the past. LASSY also implements a

utilization filter in an original way to solve effectively the problem caused by the backtracking anomaly.

APPENDIX: POST-PROLOG - A STRUCTURED PARTIALLY ORDERED PROLOG INTERPRETER

1. Introduction

One of the most significant developments in the field of artificial intelligence during the last decade has been the tremendous increase in interest in logic programming, and in particular in the programming language Prolog. The basic idea of logic programming (Kowalski, 1979; Kowalski, 1985; Lloyd, 1984) is the separation of algorithms into two disjoint components: the logic component and the control component. The logic component is the set of Horn clauses that describes the logic of the information used in solving problems. The control component is the manner in which the information is put to use.

In a declarative programming language, the programmer supplies only the logic component, and the system supplies the control component. In a procedural programming language the programmer supplies both. Ideally, logic programming would be always declarative, but unfortunately this goal cannot be completely realized. Hence both uses of logic programming are necessary (Kowalski, 1985).

Prolog (Clocksin & Mellish, 1981; Sterling & Shapiro, 1986; Warren, Pereira, & Pereira, 1977) uses a simple control rule : Clauses are matched and subgoals are solved in the order in which they are written down. It

has been claimed by many people that such a control facility is not sufficient and that better control strategies should be devised (Bruynooghe, 1978; Cox, 1984; Kowalski, 1985; Lloyd, 1984; Naish, 1985a; Naish, 1985b; Pereira, 1984). Kowalski (1985) analyses the various aspects of declarative use of Prolog , but true declarative use is impossible in Prolog. To use Prolog in a declarative way, the user would have to be able to specify the logic without the control; but this is impossible, since the user specifies the control just by writing his program down - even if it was not with conscious intent.

The problem becomes more apparent when the user intends to program part of his program in a declarative manner and another part in a procedural manner. For example, consider the following simple data base:

```
parent(john,mary).
parent(mary,tom).
.
.
parent(ann,jim).
grandparent(X,Y) ← parent(X,Z) , parent(Z,Y) .
ancestor(X,Y) ← parent(X,Y) .
ancestor(X,Y) ← parent(Z,Y), ancestor(X,Z) .
```

Let's assume that the intention of the programmer of this data base is that the interpreter can match clauses in any order it wishes, so long as the two clauses for ancestor are matched in the order in which they appear. Our programmer also doesn't care in what order the subgoals in the grandparent rule will be executed, but he wants the subgoals in the second ancestor rule to be executed in the order in which they appear (to avoid an infinite loop).

If the Prolog interpreter had a way to recognize this programmer's intention, it could decide for example to reverse the calls to parent when finding grandchildren (Naish, 1985b). But the interpreter has no way to distinguish between parts of the program that were written in a declarative

spirit, and parts that were written in a procedural spirit. In this appendix, an extension to Prolog will be defined, called Partially Ordered Prolog, which eliminates this problem. The programmer of Partially Ordered Prolog will be able to tell the interpreter, which parts of the program should be interpreted in a declarative way (the interpreter determines the control), and what parts should be interpreted in a procedural way (the programmer supplies the control). A revision of the above program using the proposed notation would look like:

```
{
parent(john,mary).
parent(mary,tom).
.
.
parent(ann,jim).
grandparent(X,Y) <- { parent(X,Z) , parent(Z,Y) } .
<
ancestor(X,Y) <- parent(X,Y) .
ancestor(X,Y) <- < parent(Z,Y), ancestor(X,Z) > .
>
}
```

Clauses enclosed in {} can be processed in any order, while clauses that are enclosed in <> must be processed in the order in which they appear. Subgoals are processed in a similar manner. Using such a notation, the interpreter can use its own control strategies for executing subgoals and matching clauses that are marked as unordered.

2. Motivations for the removal of the linear order restriction

The main motivation for freeing Prolog from the linear order constraint is to allow a user to program in both declarative and procedural fashion. Prolog is currently a procedural language, hence the linear order. If the Prolog interpreter could be made smart enough to execute

successfully every program written in a declarative way, no order would have to be assumed. To allow the whole range between the two, partial order is needed.

The current definition of Prolog throws the burden of determining the optimal ordering upon the programmer. Furthermore, it precludes the Prolog implementor from introducing optimizing features which depend on changing the order.

This is not merely a trivial matter of being unable to achieve a minor improvement in run time through optimization. Prolog's main activity is searching sets of clauses. During the last three decades, a considerable body of theoretical work in artificial intelligence has been concerned with the question of how to optimize searches by varying the order in which alternatives are explored (See (Pearl, 1984) for a comprehensive review). Hence the fact that the order in which a Prolog system searches clauses and subgoals is determined by the order in which they are entered, is a very serious restriction.

The problem becomes more apparent with the growing interest in using Prolog to implement intelligent databases (Brodie & Mattias, 1986; Dahl, 1986; Parker, et al., 1986; Sciore & Warren, 1986; Sciore & Warren, 1988; Smith, 1986; Zaniolo, 1986). One of the main problems in using Prolog for large databases is the inefficient way in which Prolog processes queries. Looking at a large database written in Prolog, one can see immediately that most of the clauses in the database were written in a declarative spirit, and only a small portion of the database – the recursive rules and rules with side effects (such as i/o) – were was written with a procedural intention.

The main motivation for removing the order restriction is the development of the LASSY system, a learning program which

automatically generates the control. The current linear order constraint makes Prolog a closed system, in the sense that the learned knowledge can not be exploited to guide the search.

It appears therefore that there are a number of good reasons for developing a version of Prolog which is free of the strict ordering constraint. It would allow an implementor to incorporate heuristics which optimized searches. It would allow the user to specify particular heuristics which could be used. Finally, it would permit the development of learning programs which developed their own heuristics for solving problems in particular domains.

3. Partially Ordered Prolog

Given the goal of developing a logic-based language which is free of the strict ordering constraint of Prolog, one is faced with two alternatives. The first is to develop a completely new logic-based language. This seems to us undesirable for a number of reasons. The process of developing a new language up to the point where other people start using it takes a great deal of time and effort. The typical lag between the initial specification of a new language and its coming into widespread use is about ten years. The overwhelming majority of new languages fail to make this transition into acceptance by the computing community. Furthermore, there is already widespread interest in Prolog and a correspondingly large amount of both software and programming expertise available for the language. It therefore appears to make a great deal of sense to try to develop something close enough to Prolog to be viewed as a new dialect rather than a new language. Ideally the new dialect should be a superset of standard Prolog.

Therefore, a new version of Prolog was defined. It will closely resemble the standard language except that extensions will be made that allow the programmer to specify a partial order on the set of clauses, and a partial order for each set of subgoals within a rule. The entire set of clauses will thus form a partially ordered set, and each rule's body will form a partially ordered set of subgoals; hence the language is called *Partially Ordered Prolog*.

3.1. Semantics

More formally, I propose to extend Prolog's semantics by introducing a new definition for a Prolog program, and a new definition for a Prolog rule.

A Partially Ordered Prolog program is a partially ordered set (poset) $\mathbf{(S,<)}$ where $\mathbf{S}$ is a set of clauses and $<$ is a partial order. For any $\mathbf{s_i, s_j} \in \mathbf{S}$, if $\mathbf{s_i < s_j}$ then Partially Ordered Prolog will never try to match the head of $\mathbf{s_j}$ before trying to match the head of $\mathbf{s_i}$. (In standard Prolog, $<$ is a linear order).

Rules in standard Prolog comprise a head and a body, and rules in Partially Ordered Prolog will have the same structure. However, while the body of a rule in standard Prolog comprises a linearly ordered set of subgoals, in Partially Ordered Prolog it will be a partially ordered set $\mathbf{(L,<_L)}$, where $\mathbf{L}$ is a set of subgoals, and $<_L$ is a partial order. For any $\mathbf{l_i, l_j} \in \mathbf{L}$, if $\mathbf{l_i <_L l_j}$ then Partially Ordered Prolog will never try to satisfy $\mathbf{l_j}$ before trying to satisfy $\mathbf{l_i}$.

The effect of these changes will be that, at a given point in time during execution, a Partially Ordered Prolog interpreter will be confronted with a set of clauses and subgoals which could be evaluated. (In contrast, in standard Prolog there is always at most one.) A heuristic function,

supplied by the language implementor or the programmer, or acquired by the system itself, could then impose a total order on that set. In this way, this single semantic extension creates a language free from the limitations of strict ordering which were discussed above.

A good way to understand the difference between the semantics of Prolog and the semantics of Partially Ordered Prolog is by looking at the abstract interpreter for logic programs (as described in (Sterling & Shapiro, 1986)) illustrated in Table 6.

Table 6. Abstract interpreter for logic programs (Sterling & Shapiro, 1986)

Let P Be a logic program
Let G be the input goal
Res := {G}
While Res <> {} do
    1. Choose $A \in Res$, $A' \leftarrow B_1, ..., B_n \in P$ such that A and A' unify with mgu θ
    2. Exit if no such goal and clause exist
    3. Res := $[(Res - \{A\}) \cup \{ B_1, ..., B_n \}]\theta$
    4. $G := G\theta$
if Res = {} then Return **G** else return **failure**.

Standard Prolog adds the following restrictions to the abstract interpreter :

- At step 1, the left-most goal is selected.
- Once a goal is selected, the interpreter tries to unify it with heads of clauses in the order in which they appear in the program.
- If Res = $A_1, \ldots, A_n$ then at step 3 we have Res := $B_1, \ldots, B_m, A_2, \ldots, A_n$[1].

In addition to the above restrictions, standard Prolog incorporates backtracking into the algorithm. When attempting to reduce a goal, the first clause whose head unifies with the goal is chosen. If no such clause is

1 This change makes the search conducted by the interpreter a depth first search.

found the computation is unwound to the last choice made, and the next unifiable clause is chosen[1].

Partially Ordered Prolog adds fewer restrictions to the interpreter:

- When choosing $A \in Res$ in step 1, we must choose A such that there is no $A' \in Res$ such that $A' < A$, where $<$ is a partial order over Res defined as follows:

  Let A_i, A_j Res. Let $C = A \leftarrow B_1, \ldots, B_n$ the deepest common ancestor of A_i, A_j in the proof tree such that $B_k = A_i$ or A_i was derived from B_k and $B_l = A_j$ or A_j was derived from B_l.

  $A_i < A_j$ iff $B_k <_c B_l$.

- When choosing a clause C whose head is unified with the selected goal A, we must choose C such that there is no C' such that $C' <_p C$, where $<_p$ is the partial order on the set of clauses of the program P.

Backtracking is incorporated into the interpreter in the same way as in standard Prolog.

3.2. Syntax

In order to allow Partially Ordered Prolog programs to be written, the syntax of standard Prolog needs to be extended in some way that allows the programmer to specify when the relation does or does not hold between pairs of clauses and subgoals. Since the new dialect should be as compatible as possible with the old, any such change must involve only minor changes to standard Prolog syntax.

The usual way to represent a partial order over a set is by a list of pairs of elements of the set, but there are two disadvantages with this approach. First, it could get very clumsy very quickly since every ordering

---

1 For an algorithm that incorporates backtracking see for example (Nilsson, 84)

must be explicitly and separately stated, and thus many elements (clauses or subgoals) will be written many times. Second, very many pairs would have to be added to a standard Prolog program in order to guarantee that it is executed in exactly the same way in Partially Ordered Prolog. The solution used here is to define a new formalism for specifying partial relations, called poset expressions.

3.2.1. Poset Expressions

Let **S** be a set. The set of the Partially Ordered Set Expressions (Poset expressions) over **S,** denoted by **R(S)**, is defined as follows:

1. For all $s \in$ **S**, $\{s\} \in$ **R(S)**. In such cases the brackets will be omitted for readability.

2. If $A_1, \ldots, A_n \in R(S)$, then so are $< A_1 \ \ldots \ A_n >$ and $\{ A_1 \ \ldots \ A_n \}$

A poset expression $A \in R(S)$ denotes a poset $(W_A, <_A)$ where W_A is defined as follows:

1. If $A = \{s\}$, $s \in S$, then $W_A = \{s\}$.

2. If $A = < A_1 \ \ldots \ A_n >$ or $\{ A_1 \ \ldots \ A_n \}$ then $W_A = \cup_{i=1\ldots n} W_{Ai}$.

and $<_A$ is defined as follows:

$\forall$ a, a' $\in W_A$, a $<_A$ a' iff one of the following is true:

1. $A = < A_1 \ \ldots \ A_n > \ \& \ \exists$ i, j [a $\in W_{Ai}$ & a' $\in W_{Aj}$ & (i<j V (i=j & a $<_{Ai}$ a')]

2. $A = \{ A_1 \ \ldots \ A_n \} \ \& \ \exists \ A_i$ [a, a' $\in W_{Ai}$ & a $<_{Ai}$ a']

Informally, one can construct poset expressions using {} to indicate non-ordered sets and <> to indicate ordered sets. Both types of expressions can be nested to any level (see the example in the introduction section).

It is easy to prove that the above representation is complete for any finite partial relation. Let R be a partial order over S

$R = \{(a_1, a_1`), \ldots, (a_n, a_n`)\}$

A poset expression that donates R is $\{< a_1\ a_1` > \ldots < a_n\ a_n` >\}$.

Looking at the above proof, one might think that the new representation scheme does not have any advantage and that the pair notation would do just as well. However, this is not the case, and we will see that there is a large subset of the set of partial orders over S that can be represented using the poset expression notation, such that no element of S will appear twice. Using the pair notation for the same partial relations would create a large number of repetitions. Consider for example the partial order denoted by

$< \{A_1 \ldots A_n\}\ \{B_1 \ldots B_m\}>$.

To represent the same partial order using the pair notation we would need to specify n×m pairs, since each A_i appears m times and each B_i appears n times.

3.2.2. Partially Ordered Prolog Syntax

A Partially Ordered Prolog program is written as a poset expression over a set of clauses. If the brackets around the top level are omitted, the whole program is considered to have an implicit <,> around it. The body of a rule will be written as a poset expression of subgoals. If the brackets around the whole body are omitted, the whole body is considered to have an implicit <,>. The definition will be modified slightly to require "," between elements of poset expressions in a rule, so that it will be compatible with current Prolog systems. Hence, any program of standard Prolog will have syntax which is legal in Partially Ordered Prolog. Furthermore, the semantics of a program of standard Prolog will remain the same in

Partially Ordered Prolog, since such a program is considered by Partially Ordered Prolog as linearly ordered.

There is one problem with this syntax . Since a clause can appear in a poset expression more then once, it is perfectly possible to specify an order that contains a contradiction, i.e. there are A, B such that A < B and B < A. Maintaining consistency is a computationally very expensive task. This will make the task of implementing Partial Order Prolog much harder. The next section therefore introduces a slightly modified formalism that will avoid the need for consistency maintenance.

3.2.3. Structured Poset Expressions

Let **P** be a poset expression. **P** will be called a structured poset expression iff there is no element of **S** that appears in **P** more then once. It is guaranteed that a structured poset expression is consistent, since any element appears at most once, and hence a circle can not be created. A digraph that represents a poset denoted by a structured poset expression is decomposable either vertically or horizontally to a set of disjoint digraphs. A structured poset expression of the form <....> will be called an *ordered* poset expression, and an expression of the form { ... } will be called a *non-ordered* poset expression.

The structured poset expression formalism is not complete in the strong sense, i.e. there are partial orders that have no structured poset expression that denotes them; however, it is complete in the weak sense - for every partial order R, there is an extension R' such that R' can be denoted by a structured poset expression. One possible algorithm for finding such an extension to a given poset is to take repeatedly the set of minimal elements and make it a non-ordered expression, and then to

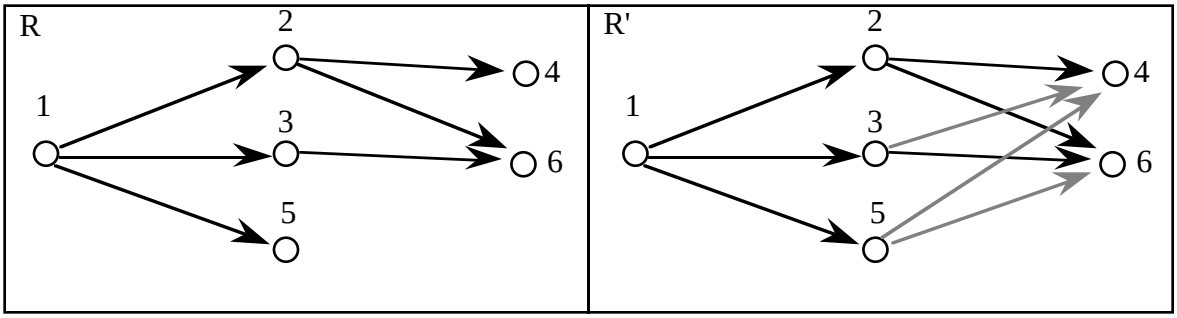

Figure 22. R' is the representation of the relation R using the poset expression notation.

create an ordered expression whose elements are those non-ordered sets in the order in which they were generated.

For example, the relation R={(1, 2)(1, 3)(1, 4)(2, 4)(2, 6)(3, 6)} can be represented by the poset expression <1 {<2 {4 6}><3 6> 5}>. Using the above algorithm we can generate the structured poset expression < 1 {2 3 4} {5 6}>. The structured poset expression denotes R', an extension of R, which has three additional pairs (see fig. 22).

3.2.4. POST Prolog: Structured Partially Ordered Prolog

In this section, a restricted version of Partially Ordered Prolog will be defined, called POST-Prolog (Partially Ordered STructured Prolog). The semantics of POST Prolog will be similar to that of Partially Ordered Prolog, but the only partial orders allowed will be those that can be denoted by a structured poset expression. The syntax for POST Prolog will be the same as for Partially Ordered Prolog, but the set of clauses and the sets of subgoals of bodies of rules will be written as structured poset expressions.

Since POST Prolog is complete in the weak sense, the only thing that can be lost by using it instead of Partially Ordered Prolog, is that the partial order that the programmer will use will be an extension of his/her intended one, and thus there may be execution points where the set of alternative

possibilities will be smaller then the set that would be available if a Partially Ordered Prolog were used. However, by using the restricted notation we save the need for consistency maintenance.

4. The relationship between Partially Ordered Prolog and other Prolog extensions

Many efforts have been made during the last decade to try to overcome the problem of inefficient execution of Prolog. Among these efforts were extensions of standard Prolog that allow the definition of control rules for controlling the program execution (Bruynooghe & Periera, 1984; Clark & McCabe, 1979; Cox, 1984; Lloyd, 1984; Pereira, 1984; Pietryzykowski & Matwin, 1982; Porto, 1984; Wise, 1984). Many of these Prolog extension use some form of what is called "intelligent backtracking." Naish (1985b) gives an overview and classification of the control constructs used by Prolog systems with extra control facilities.

There are several conceptual differences between the ideas suggested in this paper and the ideas underlying these alternative extensions. While those extensions involve adding control facilities to standard Prolog, POST-Prolog reduces the amount of control that is enforced on the user, such that the proportion between the part of the control component that is defined by the user and the part of the control component that is defined by the system, will be flexible and will be determined by the user.

While the additional control rules that most of the above systems use define **dynamic** ordering - ordering that is determined during run time and is dependent on run time parameters (such as whether a variable is bound or not), Partially Ordered Prolog offers **static** ordering - ordering that is determined before run time. The static ordering is the partial order of

execution that will be preserved for each run of the program. The dynamic ordering will be used to extend this partial order to linear order during execution. These two modes of ordering are complementary, and can be easily combined. The only dynamic re-orderings possible will be those permitted by the static ordering.

Another important characteristic of Partially Ordered Prolog is the simplicity of its use compared to most control rules of other systems which are "too sophisticated for most programmers to control." (Kowalski, 1985).

5. Implementation Issues

It is relatively easy to incorporate the suggested extensions into current Prolog systems. There are basically two parts that need to be modified. The first part is the parser. The parser should be modified to be able to read and store poset expressions. The internal representation of the database should be modified to enable the representation of partial orders. These extensions are very easy to implement.

The other part is the main loop of the interpreter itself. Any sequential Partially Ordered Prolog interpreter must include an ordering function h that can determine the order of clauses and subgoals that were not ordered by the user. The function h can be built into the system, can be supplied by the user, or can be learned by a learning system.

An experimental POST Prolog system was implemented on a Xerox 1186 Lisp machine using the Interlisp-D environment. The user can select an ordering strategy from a menu of built-in ordering strategies such as those discussed in (Kowalski, 1979; Smith & Genesereth, 1985; Warren, 1981). If we want to allow the user to specify strategies of his/her own, we

should add some facilities for specifying such strategies. Many existing dialects already contain such facilities and can use the existing facilities together with the suggested extension. For example, we can use a syntax for metarules similar to the one presented in (Gallaire & Lasserre, 1982).

Given an ordering function h, there are four possible ordering algorithms:

- Static ordering : The database is ordered before execution time (during compile time or load time). This is possible only if h does not ask for run time information such as binding of variables, etc. The interpreter behaves exactly like a standard Prolog interpreter. This method does not cause any overhead in run time but is limited in its ability to improve efficiency. However, the method can still be used to make Prolog more declarative, and novice (or lazy) users can have large parts of their program ordered by the system.
- Semi-dynamic ordering : When selecting goals for reduction, the left-most goal is selected as in standard Prolog, but when adding a set of subgoals to the goal list, the set is sorted first using h. For simple h and short rule bodies there is only a small overhead in using this method. The database itself is ordered before looping over it (in fact, only the clauses with the head predicate equal to the predicate of the current goal are ordered).
- Depth first dynamic ordering : When pushing a set of subgoals in the front of the goal list, we keep them together as a set . When choosing a goal to be reduced, the ordering function is used to find the best goal in the first set of goals. The difference in overhead between this method and the semi-dynamic method is minor (for small bodies of rules), and

we can gain by delaying selection decisions, since more information may be available. The database is not sorted, but each time that a clause is needed for the matching procedure, h is used for selecting it. This method can be more efficient than sorting methods for queries that do not require all the solutions to be generated.

- General dynamic ordering : Each time that the interpreter needs a new goal for reduction, it uses h to select from the whole goal list. This method is the most general one, but can be extremely computationally expensive since we may have a very large set of goals in the current resolvent.

The question of which ordering method is best to use is still open. It is clear that the static ordering can benefit us because it does not add any run-time overhead and yet still increases the possibility for declarative programming in Prolog. The other three methods are more problematic. It is clear that there are cases where the overhead of using these methods is greater then the computation time they may save. It is also clear that there are cases where using these methods can save a great deal of computation. The problem of when to use ordering strategies and when to use the simple control was addressed in the LASSY system.

6. Conclusion

A new dialect of Prolog called POST-Prolog was defined, which is an extension to standard Prolog. The new dialect adds only two punctuation symbols to standard Prolog, but significantly changes the semantics of a Prolog. While a standard Prolog program is a linearly ordered set of clauses, and a standard Prolog rule body is a linearly ordered set of

subgoals, the POST-Prolog program is a Partially Ordered set of clauses, and a rule body is a partially ordered set of subgoals.

It was argued that Prolog is actually a procedural programming language , and does not support true declarative programming . POST-Prolog on the other hand, can be used to specify a program either in a completely declarative way, or in a completely procedural way, or any combination of both. POST Prolog is not yet another dialect of Prolog, but a useful extension that can be incorporated rather easily into existing Prolog systems. The usefulness of POST-Prolog was demonstrated in the LASSY system, where the learning system improved the performance of the interpreter significantly by using domain specific knowledge.

REFERENCES

Aha, D. W., & Kibler, D. (1989). Noise-Tolerant Instance-Based learning Algorithms. In *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence.* Detroit, Michigan.

Boyer, R. S., & Moore, J. S. (1977). A Lemma Driven Automatic Theorem Prover for Recursive Function Theory. In *Proceedings of IJCAI.*

Brodie, M. L., & Mattias, J. (1986). On Integrating Logic Programming and Databases. In L. Kerschberg (Ed.), *Expert Database Systems.* Menlo Park, California: The bengamin/Cummings Publishing Company.

Bruynooghe, M. (1978). Intelligent Backtracking for an Interpreter of Horn Clause Logic Programs. In *Proceedings of Colloqium on Mathematical Logic in Programming.* Hungary.

Bruynooghe, M., & Periera, L. M. (1984). Deduction Revision by Intelligent Backtracking. In J. A. Campbell (Ed.), *Implementations of Prolog.* Ellis Horwood Ltd.

Buchanan, B. G., & Feigenbaum, E. A. (1982). Forward. In R. Davis & D. B. Lenat (Ed.), *Knowledge based systems in Artificial Intelligence.* McGraw-Hill.

Carbonell, J. G., & Gil, Y. (1987). Learning By Experimentation. In *Proceedings of Forth International Workshop on Machine Learning* (pp. 256-266). Irvine, California: Morgan Kaufmann.

Clark, K. L., & McCabe, F. (1979). The Control Facilities of IC-Prolog. In D. Michie (Ed.), *Expert Systems in The Microelectronic Age.* Scotland: University of Edinburgh.

Clark, K. L., McCabe, F. G., & Gregory, S. (1982). IC-Prolog Language Features. In K. L. Clark & S. A. Tarnlund (Ed.), *Logic Programming.* Academic Press.

Clocksin, W. F., & Mellish, C. S. (1981). *Programming in Prolog* . New York: Springer-Verlag.

Cox, P. T. (1984). Finding Backtrack Points for Intelligent Backtracking. In J. A. Campbell (Ed.), *Implementations of Prolog*. Ellis Horwood Ltd.

Dahl, V. (1986). Logic Programming for Constructive Expert Database Systems. In *Proceedings of Expert Database Systems - Proceedings From the First International Workshop* (pp. 209-217): The Benjamin/Cummings Publishing Company.

Debray, S. K., & Warren, D. S. (1988). Automatic mode inference for logic programs. *The Journal of Logic Programming, 5*, 207-229.

Dejong, G., & Mooney, R. (1986). Explanation-based Learning: An Alternative View. *Machine Learning, 1*, 145-176.

Doyle, J. (1979). A Truth Maintenance Systems for Problem Solving. *Artificial Intelligence, .*

Etzioni, O. (1988). Hypothesis Filtering: A Practical Approach to Reliable Learning. In *Proceedings of The Fifth International Conference on Machine Learning*. Ann Arbor, MI: Morgan Kaufmann.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and Executing Generalized Robot Plans. *Artificial Intelligence, 3*, 251-288.

Gallaire, H., & Lasserre, C. (1982). Metalevel Control for Logic Programs. In K. L. Clark & S. A. Tarnlund (Ed.), *Logic Programming*. Academic Press.

Gallaire, H., & Minker, J. (1978). *Logic and Data Bases* . New York: Plenum.

Genesereth, M. R., & Nilsson, N. J. (1988). *Logical foundations of artificial intelligence* . Palo Alto, California: Morgan Kaufmann.

Gennari, J. H. (1989). Focused Concept Formation. In *Proceedings of The Sixth International Workshop on Machine Learning* (pp. 379-382). Ithaca, New York: Morgan Kaufmann.

Greiner, R., & Likuski, J. (1989). Incorporating Redundant Learned Rules: A Preliminary Formal Analysis of EBL*. In *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence*. Detroit, Michigan.

Hogger, C. J. (1984). *Introduction to Logic Programming* . London: Academic Press.

Holland, J. H. (1986). Escaping Brittleness: The possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning : An Artificial Intelligence Approach*. Los Altos, California: Morgan Kaufmann Publishers, Inc.

Iba, G. A. (1989). A Heuristic Approach to the Discovery of Macro-operators. *Machine Learning, 3*, 285-317.

Kearns, M., Li, M., Pitt, L., & Valiant, L. G. (1987). Recent Results on Boolean Concept Learning. In *Proceedings of Forth International Workshop on Machine Learning* (pp. 337-352). Irvine, California: Morgan Kaufmann.

Keller, M. R. (1987). Concept learning in Context. In *Proceedings of Forth International Workshop on Machine Learning*. Irvine, California: Morgan Kaufmann.

Korf, R. E. (1985). *Learning to Solve Problems by Searching for Macro-Operators* . Boston: Pitman.

Kowalski, R. A. (1979). *Logic for Problem Solving* . New York: Elseveir North Holland.

Kowalski, R. A. (1985). Directions for Logic Programming. In *Proceedings of IEEE Symposium On Logic Programming*.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in Soar: The Anatomy of a General Learning Mechanism. *Machine Learning, 1*, 11-46.

Lenat, D. B. (1983). The Role of Heuristics in Learning by Discovery: Three case studies. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning : An Artificial Intelligence Approach*. Palo Alto: Tioga.

Lenat, D. B., & Feigenbaum, E. A. (1989). On the Thresholds of Knowledge. In R. Quinlan (Ed.), *Applications of Expert Systems*. Addison-Wesley.

Lloyd, J. W. (1984). *Foundations of Logic Programming* . Springer-Verlag.

Markovitch, S., & Scott, P. D. (1988a). *Knowledge Considered Harmful* (TR No. 030788). the Center for Machine Intelligence.

Markovitch, S., & Scott, P. D. (1988b). The Role of Forgetting in Learning. In *Proceedings of The Fifth International Conference on Machine Learning*. Ann Arbor, MI: Morgan Kaufmann.

Markovitch, S., & Scott, P. D. (1989a). Automatic Ordering of Subgoals - a Machine Learning Approach. In *Proceedings of North American Conference on Logic Programming*. Ithaca, NY.

Markovitch, S., & Scott, P. D. (1989b). Information filters and their implementation in the SYLLOG system. In *Proceedings of The Sixth International Workshop on Machine Learning*. Ithaca, New York: Morgan Kaufmann.

Markovitch, S., & Scott, P. D. (1989c). *POST-prolog: Structured Partially Ordered prolog* (TR No. CMI-89-018). Ann Arbor, Michigan: Center for Machine Intelligence.

Markovitch, S., & Scott, P. D. (1989d). Utilization Filtering: a method for reducing the inherent harmfulness of deductively learned knowledge. In *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence*. Detroit, Michigan.

Michalski, R. S. (1986). Understanding the Nature of Learning: Issues and Research Directions. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchel (Ed.), *Machine Learning: An Artificial Intelligence Approach*. Los Altos,CA: Morgan-Kaufmann.

Minton, S. (1985). Selectively Generalizing Plans for Problem Solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 596-599). Los Angeles, CA.

Minton, S. (1988a). *Learning Search Control Knowledge: An Explanation-Based Approach* . Boston, mass: Klower Academic Publishers.

Minton, S. (1988b). Quantitative Results Concerning the Utility of Explanation-Based Learning. In *Proceedings of AAAI* (pp. 564-569).

Mitchell, T. M. (1979). An Analysis of Generalization as a Search Problem. In *Proceedings of Sixth International Joint Conference on Artificial Intelligence*. Tokyo, Japan.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-Based Generalization: A Unifying View. *1*, 47-80.

Mitchell, T. M., Utgoff, P. E., & Banerji, R. (1983). Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning : An Artificial Intelligence Approach*. Palo Alto, California: Tioga.

Mooney, R. (1989). The Effect of Rule Use on the Utility of Explanation-Based Learning. In *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence*. Detroit, Michigan.

Naish, L. (1985a). Automatic control for logic programs. *The Journal of Logic Programming, 3,* 167-183.

Naish, L. (1985b). Prolog Control Rules. In *Proceedings of IJCAI* (pp. 720-722). Los Angeles.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence* . Palo Alto, CA: Tioga.

Parker, S. D., Carey, M., Golshani, F., Jarke, M., Sciore, E., & Walker, A. (1986). Logic Programming and Databases. In *Proceedings of Expert Database Systems - Proceedings From the First International Workshop* (pp. 35-48): The Benjamin/Cummings Publishing Company.

Pearl, J. (1984). *Heuristics : intelligent search strategies for computer problem solving* . Addison-Wesley.

Pereira, L. M. (1984). Logic Control with Logic. In J. A. Campbell (Ed.), *Implementations of Prolog*. Ellis Horwood Ltd.

Pereira, L. M., & Porto, A. (1982). Selective Backtracking. In K. L. Clark & S. A. Tarnlund (Ed.), *Logic Programming*. Academic Press.

Pietryzykowski, T., & Matwin, S. (1982). Exponential Improvement of Efficient Backtracking. In *Proceedings of 6th Conference on Automatic Deduction*: Springer-Verlag.

Porto, A. (1984). EPILOG: A language for Extended Programming. In J. A. Campbell (Ed.), *Implementations of Prolog*. Ellis Horwood Ltd.

Prieditis, A. E., & Mostow, J. (1987). PROLEARN: Toward a Prolog Interpreter that Learns. In *Proceedings of the sixth National conference on Artificial Intelligence.* Seatle, WA.

Quinlan, J. R. (1986). Induction of Decision trees. Machine Learning, 1, 81-106.

Ruff, R. A., & Dietrich, T. G. (1989). What Good are Experiments? In *Proceedings of The Sixth International Workshop on Machine Learning* (pp. 109-112). Ithaca, New York: Morgan Kaufmann.

Samuel, A. L. (1963). Some Studies in Machine Learning Using the Game of Checkers. In E. A. Feigenbaum & J. Feldman (Ed.), *Computer and Thought*. New York: McGraw-Hill.

Sciore, E., & Warren, D. S. (1986). Towards an Integrated Database-Prolog System. In Kerschberg (Ed.), *Expert Database Systems*. Menlo Park, California: The bengamin/Cummings Publishing Company.

Sciore, E., & Warren, D. S. (1988). Integrating Databases & Prolog. .

Scott, P. D. (1983). Learning : The Construction of A Postriori Knowledge Structures. In *Proceedings of AAAI*.

Scott, P. D., & Markovitch, S. (1989a). Learning Novel Domains Through Curiosity and Conjecture. In *Proceedings of International Joint Conference for Artificial Intelligence*. Detroit, Michigan.

Scott, P. D., & Markovitch, S. (1989b). Uncertainty Based Selection of Learning Experiences. In *Proceedings of The Sixth International Workshop on Machine Learning*. Ithaca, New York: Morgan Kaufmann.

Scott, P. D., & Vogt, R. C. (1983). Knowledge Oriented Learning. In *Proceedings of The Eighth International Conference on Artificial Intelligence*. Karlsruhe, W. Germany.

Segre, A. M. (1987). On the Operationality/generality Trade-Off in Explanation-Based Learning. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. Milan, Itali.

Shalin, V. L., Wisniewski, E. J., Levi, K. R., & Scott, P. D. (1987). A Formal Analysis of Machine Learning Systems for Knowledge Acquisition. In *Proceedings of Second AAAI Knowledge Acquisition for Knowledge Based Systems Workshop*. Banff, Canada.

Shanon, C. E., & Weaver, W. (1949). *The Mathematical Theory of Communication* . University of Illinois Press.

Shrager, J. (1989). A Graph-Dynamic Model of the Power Law of Practice and the Problem-Solving Fan-Effect. *Science, 242*, 414-416.

Simon, H. A., & Lea, G. (1974). Problem Solving and Rule Induction: A Unified View. In L. W. Gregg (Ed.), *Knowledge and Cognition*. Maryland: Lawrence Erlbaum.

Smith, D. E., & Genesereth, M. R. (1985). Ordering Conjunctive Queries. *Artificial Intelligence, 26*, 171-215.

Smith, J. M. (1986). Expert Database Systems: A Database Perspective. In *Proceedings of Expert Database Systems - Proceedings From the First International Workshop* (pp. 3-15): The Benjamin/Cummings Publishing Company.

Sterling, L., & Shapiro, E. (1986). *The Art of Prolog* . Cambridge, MA: MIT Press.

Subramanian, D., & Genesereth, M. R. (1987). The Relevance of Irrelevance. In *Proceedings of International Joint Conference for Artificial Intelligence.* Milan.

Tambe, M., & Newell, A. (1988). Some Chunks Are Expensive. In *Proceedings of The Fifth International Conference on Machine Learning.* Ann Arbor, MI: Morgan Kaufman.

Tambe, M., & Rosenbloom, P. (1989). Eliminating Expensive Chunks by Restricting Expressiveness. In *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence.* Detroit, Michigan.

Utgoff, P. E. (1989). Improved Training Via Incremental Learning. In *Proceedings of The Sixth International Workshop on Machine Learning* (pp. 362-365). Ithaca, New York: Morgan Kaufmann.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM, 27*, 1134-1142.

Warren, D. H. D. (1981). Efficient Processing of Interactive Relational Data Base Queries Expressed in Logic. In *Proceedings of Seventh International Conference on Very Large Data Bases.*

Warren, D. H. D., Pereira, L. M., & Pereira, F. (1977). PROLOG- The Language and its Implementation Compared with Lisp. In *Proceedings of ACM Symp. on AI and Programming Languages.*

Wilkins, D. C. (1987). *Apprenticeship Learning Techniques for Knowledge Based Systems.* Unpublished thesis, University of Michigan.

Winston, P. H. (1975). Learning Structural Descriptions from Examples. In P. H. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill.

Wise, J. M. (1984). EPILOG: Re-interpreting and extending Prolog for a Multiprocessor Environment. In J. A. Campbell (Ed.), *Implementation of Prolog*. Ellis Horwood Ltd.

Yamada, S., & Tsuji, S. (1989). Selective Learning of Macro-operators with Perfect Causality. In *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence.* Detroit, Michigan.

Zaniolo, C. (1986). Prolog: A Database Query Language for All Seasons. In L. Kerschberg (Ed.), *Expert Database Systems.* Menlo Park, California: The bengamin/Cummings Publishing Company.