Machine Learning, 10, 113–151 (1993)

Information Filtering: Selection Mechanisms in Learning Systems

SHAUL MARKOVITCH SHAULM@CS.TECHNION.AC.IL
Computer Science Department, Technion, Haifa 32000, Israel

PAUL D. SCOTT SCOTP@ESSEX.AC.UK
Department of Computer Science, University of Essex, Colchester CO4 3SQ, United Kingdom

Editor: Bruce Porter

Abstract. Knowledge has traditionally been considered to have a beneficial effect on the performance of problem solvers but recent studies indicate that knowledge acquisition is not necessarily a monotonically beneficial process, because additional knowledge sometimes leads to a deterioration in system performance. This paper is concerned with the problem of harmful knowledge: that is, knowledge whose removal would improve a system's performance. In the first part of the paper a unifying framework, called the *information filtering model*, is developed to define the various alternative methods for eliminating such knowledge from a learning system where selection processes, called filters, may be inserted to remove potentially harmful knowledge. These filters are termed selective experience, selective attention, selective acquisition, selective retention, and selective utilization. The framework can be used by developers of learning systems as a guide for selecting an appropriate filter to reduce or eliminate harmful knowledge.

In the second part of the paper, the framework is used to identify a suitable filter for solving a problem caused by the acquisition of harmful knowledge in a learning system called LASSY. LASSY is a system that improves the performance of a PROLOG interpreter by utilizing acquired domain specific knowledge in the form of lemmas stating previously proved results. It is shown that the particular kind of problems that arise with this system are best solved using a novel utilization filter that blocks the use of lemmas in attempts to prove subgoals that have a high probability of failing.

Keywords. Harmful knowledge, information filtering, selective learning

1. Introduction

The most important development in AI research during the 1970s was the widespread acceptance of the *knowledge as power hypothesis*, which Buchanan and Feigenbaum (1982) have succinctly stated as:

> *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.*

There are two possible ways in which an intelligent system could acquire the knowledge it needs: the system could be given the knowledge by some external agency, such as the system's developers, or the system could acquire the knowledge itself through some form of learning procedures. Determining what knowledge is required and supplying it to the system is often extremely laborious and occasionally impossible. Consequently, much research activity has been devoted to the development of machine learning techniques in the hope that these will enable the process of knowledge acquisition to be fully or partially automated.

Machine learning techniques enable systems to obtain much greater quantities of knowledge than can feasibly be supplied by system developers. Unfortunately, because less intelligence is involved in the process of acquiring it, this knowledge may well be of lower quality. Consequently, systems with limited resources that make use of machine learning techniques may encounter problems caused by knowledge that is actually detrimental to the system. As learning systems are applied to more realistic and complex domains, such problems are likely to become increasingly significant.

The need for systems to consider the quality of the knowledge they acquire has long been recognized, and many systems have been developed that include components to eliminate potentially harmful knowledge: a representative sample of these is discussed in Section 3. However, until recently, most work of this type has treated the problem of eliminating harmful knowledge as a side issue that has to be dealt with as part of the more important business of building a system that learns. Consequently, such work has seldom included comparative experimental studies to investigate how effective the knowledge selection procedures are. Nor has there been any attempt at a systematic and comprehensive consideration of the various methods that can be used to detect and remove potentially harmful knowledge.

However, in recent years, research has started to appear whose main focus is the method used to select knowledge. Minton's work (1985; 1988b; 1988a) is a particularly notable contribution to this topic. At the same time, there has been a growing awareness of the many ways in which even correct knowledge can be harmful (see for example Markovitch and Scott (1988), Tambe and Newell (1988), and Wilkins and Ma (1989).

This paper is intended to further our understanding of the role of selection processes in learning in a number of ways:

1. By providing an unambiguous but generally applicable definition of the *utility* of knowledge.
2. By developing a systematic classification scheme for such selection processes that is both *descriptive* in that it can be used to describe any application of selection within a learning system, and *prescriptive* in that it can be used to guide the choice of an appropriate selection procedure.
3. By demonstrating how this framework may be used to solve a problem caused by harmful knowledge that arose during the development of a particular learning system.

In Section 2 we present a definition of the utility of knowledge and a discussion of the various ways in which knowledge can be harmful. In Section 3 we describe the *information filtering framework* that we propose as a classification scheme for the selection processes used in learning systems. We demonstrate the descriptive powers of the framework by showing how it can be applied to a wide variety of existing systems and discuss the advantages and disadvantages of the various types of selection process. The prescriptive power of the framework is considered in Section 4, which is a case study involving the use of the framework to help find a way of dealing with a problem caused by harmful knowledge that arose during the construction of a learning system called LASSY. This case study emphasizes the importance of choosing the right type of selection process. The particular selection process chosen is a novel solution to a class of problems that will arise in any deductive learning system using a problem solver that can backtrack. Section 5 discusses how selection procedures can be viewed as problem solvers in their own right and may therefore include further learning procedures. Finally, Section 6 provides a summary and a discussion of the implications for future research.

2. The utility of knowledge and experience

We begin our development of a systematic approach to the problems of harmful knowledge by defining what it means for knowledge to be harmful, and by considering what characteristics of knowledge and of the context in which it is used can cause it to be so.

2.1. Costs and benefits of knowledge

Knowledge can be harmful because there are potential costs as well as potential benefits associated with its retention and use: if the costs exceed the benefits then it will be harmful.

2.1.1. Benefits of knowledge

The potential benefits of knowledge to a problem solver fall into two categories:

Better Solutions: Knowledge may influence the quality of solution found by a problem solver. The most obvious case arises when additional knowledge enables a system to solve a problem it was previously unable to solve, but other examples occur when the additional knowledge enables the system to find solutions that are in some way superior: for example, they may be cheaper to apply or have a higher probability of being correct.

More Efficient Problem Solving: Knowledge may enhance the performance of a problem solver by reducing the resources (such as time) required to find a solution. Knowledge is often added to a system specifically to reduce search time either by pruning the search tree or by eliminating the need to repeat a particular portion of the search.

It is worth noting that, while these two types of benefit are logically distinct, in practice there is a strong interaction. Additional knowledge that reduces the resources required to find a solution may have the effect of permitting the problem solver to explore more of the space and hence find higher quality solutions.

2.1.2. Costs of knowledge

Some of the potential costs of knowledge correspond directly to the potential benefits:

Poorer Solutions: Additional knowledge may have a deleterious effect on the quality of solutions found by a problem solver. The most obvious situation in which this occurs is when the additional knowledge is incorrect. However, there are circumstances in which adding correct knowledge to a system leads to the production of poorer solutions (see for example Wilkins and Ma (1989)).

Less Efficient Problem Solving: Although additional knowledge can enhance the performance of a problem solver by reducing its resource requirements, it can also have exactly the opposite effect. In particular, additional knowledge can have the effect of greatly increasing the search time required to find solutions because of additional matching time or larger branching factors: an example in which adding correct knowledge leads to an exponential increase in search time is discussed later in this paper.

There is an interaction between deleterious effects on problem solver efficiency and solution quality similar to that already noted for beneficial effects. If the performance of the problem solver is sufficiently impaired by the additional knowledge it may be unable to find solutions for problems it could previously solve.

These costs are marginal costs in that they are incurred each time a problem is solved. In addition there are also fixed overhead costs that are independent of the number of problems solved:

Acquisition Costs: The process of acquiring knowledge can be costly, either in human resources, if the knowledge is supplied by the system developers, or in computing resources, if machine learning techniques are employed. This cost is a very important factor in determining whether it is worthwhile to build a knowledge based system.

Storage Costs: Storing knowledge requires memory resources and this has a cost. While it is true that memory is now very much cheaper than it was in the early years of AI research, it can still be a significant factor. In many systems, as memory capacity is reached, a choice has to be made regarding which knowledge to retain. In such circumstances, storing a piece of knowledge has an opportunity cost: the potential value of the knowledge it displaced.

The relative significance of these fixed costs depends on the number of problem-solving runs over which they may be amortized. It should be noted however that the opportunity costs of using storage to hold a particular piece of knowledge are not fixed costs, since their effects may be manifest whenever the problem solver is used.

2.2. The utility of knowledge

Clearly the value of knowledge to a system depends on the difference between its costs and its benefits. However, these depend upon a number of factors determined by the situation in which the knowledge is used:

The Problem Set: The usefulness of some piece of knowledge obviously depends upon the problems that the problem solver using it will be called upon to solve. Thus the value of knowledge can only be defined with respect to some set of problems.

The Problem Solver: The usefulness of a piece of knowledge also depends upon the ability of the problem solver to make use of it, which in turn depends on the method employed by the problem solver to find a solution. Thus the value of knowledge can only be defined with respect to a specified problem-solving system.

Other Available Knowledge: The usefulness of a piece of knowledge depends critically on what other knowledge the system possesses. Thus the value of knowledge can only be defined with respect to the other knowledge with which it will be used.

Evaluation Criteria: Since the purpose of adding knowledge to a problem-solving system is to improve its performance, it is reasonable to assess the usefulness of the knowledge in terms of its effect on that performance. However, there are many dimensions of performance that might be included in such an evaluation, including quality of the solution, resources expended in finding a solution, and resources required to apply a solution.

Minton (1988a) provides an operational definition for the utility of a control rule:

$$\textit{Utility} = \textit{Average-search-time-without-rule} - \textit{Average-search-time-with-rule}$$

Similarly, Markovitch and Scott (1988) defined the value of an item of knowledge as the expectation value for the difference between the cost of solving a problem with the item and solving it without it. These can be generalized, taking account of all the factors listed above, to provide an informal definition of the *utility* of any change in a knowledge base:

Definition 1 (Utility of a Change in a Knowledge Base). *The utility of a change in a knowledge base, for a problem solver solving problems drawn from some set of problems, is the difference between the expectation values for an evaluation of the problem solver performance before and after the change is made.*

A formal version of this definition appears in Appendix A. As can be seen, Minton's definition is a special case of definition 1, in which changes take the form of modifying a rulebase by adding or deleting rules, and the evaluation criterion is search time to find a solution. One major advantage of the more general formulation is that it may be applied to systems using other criteria to evaluate performance. For example, MetaLEX (Keller, 1987) estimates the utility of subexpressions of the USEFUL concept using a heuristic function that combines both estimated benefits to problem solver efficiency (reduced time spent on evaluating the subexpression) and estimated costs to solution quality (more frequent erroneous evaluations).

The definition proposed above is applicable to an arbitrary change in any knowledge base. The type of changes that may occur will depend on the particular representation scheme employed. Thus when knowledge is represented by numerical parameters, changes will take the form of arithmetical operations on parameter values. However, there is an important class of knowledge representation schemes in which the knowledge base can be viewed as a set of *knowledge elements* that can be added and removed independently. These elements may take many forms including logical expressions, production rules, or links in a network. The majority of the systems discussed in this paper use this type of representation scheme. Since changes in such systems take the form of adding or removing knowledge elements, it is possible to define the utility of a single knowledge element:

Definition 2 (Utility of a Knowledge Element). *The utility of a knowledge element in the context of a given knowledge base, for a problem solver solving problems drawn from some set of problems, is the utility of the change in that knowledge base made by adding the knowledge element.*

A formal version of this definition is given in Appendix A. The utility of a set of knowledge elements can be defined in a similar manner. Knowledge elements whose utility is negative are said to be *harmful.* It is important to note that harmfulness is not an inherent property of the knowledge element itself but depends upon the other knowledge the system possesses, the problems to be solved, and the methods used by the problem solver.

2.3. Assessing the utility of knowledge

Although these definitions are operational in nature, they do not usually provide a practical technique for assessing the utility of knowledge. Obtaining a reliable estimate of the performance of a problem solver before and after a change is made would require a substantial number of problem solving runs for every proposed change. Furthermore, the fact that the utility of a knowledge element depends on the rest of the knowledge base means that the potential utility of a new element can only be accurately assessed by considering it in conjunction with every subset of the existing knowledge base: in general, such a procedure will have exponential time complexity.

Practical systems must therefore employ heuristic methods to estimate the utility of proposed changes to a knowledge base. Any heuristic technique that attempts to identify harmful knowledge elements must be based on properties of knowledge elements and their relationship to the rest of the knowledge base, the problems to be solved, and the problem solver. It is therefore appropriate to examine some of the characteristics of knowledge elements that may cause them to be harmful:

Incorrectness: Incorrectness, which is a relation between a knowledge element and the problem domain, is the most obvious candidate for a characteristic of knowledge that can lead to detrimental effects on the performance of a problem solver. However, it is certainly not the case that incorrect knowledge is always harmful. For example, a common type of incorrect knowledge is an over-generalization that is true for most but not all of the cases encountered in the problem domain: provided the average payoff for using the generalization exceeds that for not using it, the generalization will be beneficial even though it is not correct. The majority of machine learning systems are designed to search for correct representations.

Irrelevance: A knowledge element is irrelevant if it cannot be used in deriving a solution to a problem. Hence irrelevance is a relationship between a knowledge element, the problems to be solved, and the problem solver. Irrelevant knowledge is never beneficial. It may significantly impair the efficiency of the problem solver by increasing search times (although this effect can be reduced by appropriate index schemes), and it will incur storage costs, including storage opportunity costs for any knowledge elements it supplants. Clearly irrelevant knowledge is always undesirable. Furthermore, knowledge that is only infrequently relevant is also likely to be harmful unless the benefits when it is relevant are large. Many proposed heuristics for assessing knowledge utility include a term based on frequency of use to provide an estimate of relevance (Iba, 1989; Markovitch & Scott, 1988; Minton, 1988a; Samuel, 1959).

Redundancy: A knowledge element is redundant if it is logically entailed by the other knowledge available to the problem solver using the inference mechanisms at its disposal. Hence redundancy is a relationship between a knowledge element, the rest of the knowledge base, and the problem solver. Redundant knowledge can have a considerable effect on problem solver efficiency, but, unless this effect influences the proportion of the solution space that the problem solver has sufficient resources to explore, it will not have any impact on the quality of solutions found. Redundant knowledge is frequently very useful in reducing search times since it enables the results of searches that have already been made to be used again without further searching—a technique that is often termed *compiled search*. This technique is the basis of learning programs that use deduction rather than induction to extend their knowledge bases (Dejong & Mooney, 1986; Files, Hart, and Nilsson, 1972; Korf, 1985; Laird, Rosenbloom, & Newell, 1986; Mitchell, Keller, & Kedar-Cabelli, 1986). Unfortunately, redundant knowledge can also be very harmful. Adding knowledge elements increases the space to be searched by the problem solver and, in some circumstances, this increase can greatly outweigh any benefits achieved through compiled search. An example is discussed in Section 4. Because it has the potential of being either very beneficial or very harmful, it is often difficult to devise heuristics to estimate the utility of redundant knowledge.

This list may not be exhaustive, but we are not aware of any example of harmful knowledge that is correct, significantly relevant, and non-redundant.

2.4. Utility of experience

The discussion so far has been concerned with the utility of knowledge. However, systems that learn derive some or all of their knowledge from training experiences. It is therefore possible to define the utility of training experiences in terms of the changes they give rise to in the knowledge base:

Definition 3 (Utility of a Set of Training Experiences). *The utility of a set of training experiences, for a problem solver that both uses a knowledge base to solve problems drawn from a set of problems, and uses a learning procedure to make modifications to that knowledge base derived from the training experiences, is the utility of the change in the knowledge base brought about by those modifications.*

A formal version of this definition is given in Appendix A. It should be noted that this defines the utility of the experiences entirely from the perspective of the problem solver: no account is taken of the resources expended by the learning component of the system whose cost should be amortized over all the problems the system will solve. In practice this is an important consideration in the design of a learning system. Training experiences fall into two groups:

Informative Experiences: Experiences which lead to changes in the knowledge base. These changes may be either harmful or beneficial. Since learning systems are usually designed to search for correct representations, deleterious changes produced by experiences usually take the form of either irrelevant or harmfully redundant knowledge. However, the pathological behavior of Samuel's Checker Player (Samuel, 1959), in which its competence actually deteriorated as a result of playing a poor opponent, provides an example of harmful experience leading to the acquisition of incorrect knowledge.

Uninformative Experience: Experiences which do not produce any change in the knowledge base. Following the definition given above, such experiences have zero utility. However, they have a detrimental effect on the performance of the learning component since resources are expended without producing any benefits. Hence this type of training experience should be avoided.

Like utility of knowledge, the utility of experience depends on the other knowledge available to the system, the set of problems to be solved, and the methods used by the problem solver. In addition it is also dependent on the methods employed by the learning component of the system.

Systems normally estimate the potential utility of experiences before they are processed by the learning component. It is thus not possible to determine this utility by considering the changes such experience cause in the knowledge base, and hence heuristic methods must be used. A number of examples are discussed in the next section.

2.5. Summary

In this section we have proposed precise definitions, for the utilities of changes in a knowledge base and of training experiences, that unify and extend earlier attempts to define the value of knowledge to a problem-solving system. These provide an essential foundation for a systematic discussion of methods for dealing with harmful knowledge. They also serve to emphasize that the utility of knowledge is dependent on the context in which it is used: in particular, that it is dependent on the other knowledge available to the system and that this dependency makes the problem of identifying optimal knowledge bases computationally hard. We have also reviewed some of the characteristics of knowledge that are associated with negative utility and hence are of direct relevance to the development of techniques for eliminating harmful knowledge.

3. The information filtering framework

Having defined what it means for knowledge or experience to be harmful, we now turn to considering methods of dealing with the problem. There are many ways in which knowledge can be harmful and many ways of attempting to eliminate it, some of which originated in the earliest years of machine learning research (Samuel, 1959). In this section we present a unifying framework for the systematic discussion of all the various methods proposed for dealing with harmful knowledge. This framework clarifies the relationships between existing techniques and provides a tool to assist system developers in finding suitable methods for dealing with any problems caused by harmful knowledge.

3.1. The information flow model

The basis of the framework is the data flow model of a problem-solving system that learns from experience (Figure 1). Training experiences from the space of possible experiences are input to the system; certain aspects of these experiences are passed to the acquisition procedure by the attention procedure; the acquisition procedure is the learning component of the system that uses this input to generate changes in the knowledge base; finally, the problem solver uses the knowledge base to generate solutions to problems. (Note that for clarity, feedback between the various stages has been omitted from the diagram.)

It is possible to introduce procedures to eliminate potentially harmful experience or knowledge at any stage in this data flow. Such selective procedures will be termed *filters* since their purpose is to permit only data of positive utility to flow on to subsequent stages. Such filters can be classified according to their location within the data flow. Figure 2 shows that there are five such locations. A *selective experience* filter makes a selection from the

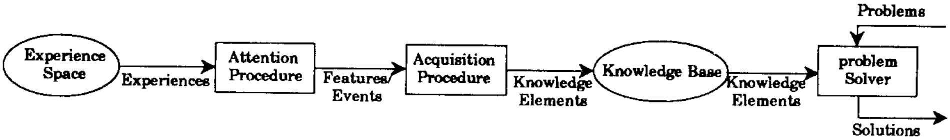

Figure 1. Information flow in a learning system.

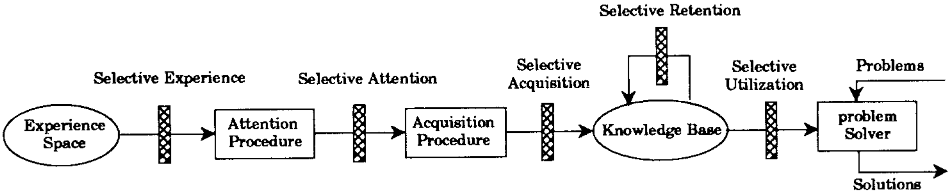

Figure 2. The five possible locations for information filters in a learning system.

set of experiences available to the learning system. A *selective attention* filter makes a selection from the various attributes of particular experiences. A *selective acquisition* filter selects from among the knowledge changes generated by the acquisition procedure before they are included in the knowledge base. A *selective retention* filter makes a selection of the items in the current knowledge base and discards those that it deems should not be retained: this process may also be called *forgetting*. Finally, *selective utilization* makes a selection from the current knowledge base of items to be made available to the problem solver. Note that in some systems the filters may serve to assign higher and lower priorities to data items rather than discard them entirely.

We now proceed to discuss each of these filters in detail in the order in which they occur in the data flow.

3.2. Selective experience

If the space of possible training experiences from which a system could learn is very large, and if there is great variation in the amount that can usefully be learned from individual experiences, then the system will make little progress in learning unless an effective method of selecting training examples is provided. This problem arises in almost all learning systems, but most circumvent it by relying on a human teacher to select informative training experiences (Winston, 1975). Systems that do not have the benefit of such external assistance therefore require some form of selective experience filter: systems that are assisted by a teacher may still benefit by applying further selection of their own.

It may sometimes be advantageous for a system to select its own training experiences even when an external teacher is available. Scott and Markovitch (1989a) argue that a learning system has an advantage over an external teacher in selecting informative examples because, unlike a teacher, it can directly access its own knowledge base. This is important because the utility of a training experience depends upon the current state of the knowledge base. Ruff and Dietrich (1989) studied five strategies for selecting experiments for a classification program and confirmed that selective experience is more effective than non-selective experience. The best strategy selected experiments that most nearly split the hypothesis space in half, but results that were almost as good were achieved by a much cheaper strategy that selected any experiment that would eliminate at least one hypothesis.

An effective experience filter should select examples that will lead to beneficial changes in the knowledge base. Existing attempts to achieve this can be grouped into three categories on the basis of the criteria used to identify informative examples.

Error Based: This type of filter uses the correct performance of the problem solver as a criterion for selecting training experiences. ID3 (Quinlan, 1986) employs a selection filter of this type: after each iteration of the decision tree building component, the current decision tree is used to classify some novel examples and those that are misclassified are added to the set of examples to be used in the next iteration. ID5R (Utgoff, 1989), an incremental variant of ID3, similarly restricts its training experience to examples that would be misclassified by the current decision tree.

The error based approach, which incorporates the idea of learning from mistakes, can be very effective, but its use would appear to be confined to those situations in which training experiences take the form of ordered pairs of problems and solutions.

Uncertainty Based: This type of filter selects only those experiences that are likely to reduce the uncertainty within the knowledge base. One such approach is exemplified by Mitchell's (1982) technique of selecting examples for the candidate elimination algorithm by choosing those that come closest to matching half of the remaining candidates in the version space. In this case, the uncertainty of the knowledge base takes the form of multiple alternative hypotheses that might be correct. LEX (Mitchell, Utgoff, & Banerji, 1983) employed both a simpler variant of this technique and a second selection process that generates examples for which more than one operator is currently applicable. In the latter case, the uncertainty resides in the choice of operators.

DIDO (Scott & Markovitch, 1989b) is an exploratory learning program that makes use of Shannon's information theoretic definition of uncertainty (Shannon & Weaver, 1949) to determine its choice of training experiences. Its goal is to reach a state in which it can predict the consequences of applying any operation to any entity in its domain. The knowledge base includes elements, called practical conditionals, that list the various outcomes that have been observed when a specific operation has been applied to members of a particular class, together with estimates of their probabilities. Applying the Shannon function to this set of probabilities yields the uncertainty associated with the practical conditional. DIDO generates further training experiences by finding the most uncertain practical conditional and applying the corresponding operation to members of the class in which it is found.

Like the error based approach this technique can be extremely effective. Its use would appear to be restricted to those knowledge bases that permit multiple alternatives to be stored.

Miscellaneous Heuristics: Lenat's AM (1983) employs a bewildering array of heuristics for experience selection that defy any attempt at classification. The result of applying these heuristics is a measure of how "interesting" a potential experience is: those that are most interesting are explored first.

The most important role of a selective experience filter is to conserve the resources of the learning component by reducing the effort expended on uninformative examples. It can also reduce the amount of irrelevant knowledge that is acquired. However, because it deals with experiences before they have been transformed into knowledge, a selective experience filter cannot contribute to preventing the acquisition of harmfully incorrect or redundant knowledge.

3.3. Selective attention

If the individual training experiences that a learning system encounters are complex, the number of potentially relevant combinations of features of these experiences will be very large. Hence the system will make very slow progress in learning unless an effective method for focusing attention on potentially important aspects of training experiences is provided. This is the role of a selective attention filter. Comparatively little work has been done on this topic: the majority of machine learning systems either work with relatively simple examples or have a built-in attentional bias. However, the current rise in interest in learning relations is likely to create a greater need for this type of selection process.

The OBSERVER module of PRODIGY (Minton, 1988a) is an example of a selective attention filter. The experiences used to train the explanation based learning component are traces of the behavior of the problem solver. The OBSERVER scans these trace trees and identifies any instances of target concepts. When an example is identified, PRODIGY uses a set of example selection heuristics to filter out those examples that do not appear likely to produce useful control rules. Those examples that pass through this filter are passed to the explanation based learning component.

MACLEARN (Iba, 1989) is a macro learning system that employs a selective attention filter. As in PRODIGY, the experiences are traces of the search tree. Instead of looking at all possible macros (sequences of states), the system considers only those sequences that are between states whose values are peaks in the evaluation function relative to a path in the search tree.

Gennari's CLASSIT-2 (1989) provides an example of the use of a selective attention filter in an incremental concept formation system. An information theoretic measure termed "salience" is computed for each attribute of a concept: the system then gives the most salient features the highest priority in its search for a good clustering.

Like the selective experience filter, the main role of a selective attention filter is to conserve the resources of the learning component by reducing the effort wasted on aspects of the training experiences which do not yield useful knowledge.

3.4. *Selective acquisition*

A selective acquisition filter is the first type of selection process in the data flow that occurs after the experience has been transformed into knowledge, the form in which it may be used by the problem solver. It is therefore better able to determine whether that knowledge is likely to be beneficial or harmful. However, it must make this determination before any attempt has been made to use the knowledge in problem solving, so it has less information at its disposal than subsequent filters.

Existing methods of selective acquisition fall into two main groups according to whether the knowledge they select is intended to lead to better solutions or to improve the efficiency of the problem solver:

Acquisition Filtering for Better Solutions: A selective acquisition filter that discards poor representations can be used to refine the output of learning systems that employ a generate-and-test strategy to find good representations of problem domains. For example, in Meta-DENDRAL (Lindsay, Buchanan, Feigenbaum, & Lederberg, 1980) the RULEGEN procedure generates a set of rules which is then passed to the RULEMOD program, which acts as an acquisition filter by searching for a small subset of rules that account for all the data. Rules are evaluated using a scoring function that assigns higher scores to those that correctly predict peaks not predicted by other rules.

Instance based learning systems, such as IB4 (Aha, Kibler, & Albert, 1991), provide a very interesting demonstration of the power of selective acquisition. In such systems, knowledge takes the same form as the training experiences and hence learning reduces to a process of selective acquisition applied directly to untransformed experiences. Thus IB4 simply adds only those training instances that would be classified incorrectly to its representation of the domain. (Other filters employed in IB4 are discussed below.)

Acquisition Filtering for Greater Efficiency: Selective acquisition filters may also be used to eliminate knowledge that is likely to impair the efficiency of the problem solver. For example, after PRODIGY (Minton, 1988a) has learned a new control rule it produces an initial estimate of its utility, based on the training example that produced the rule, by subtracting the time cost of matching the rule from the time saving the rule would have provided by eliminating search. MACLEARN (Iba, 1989) employs a *static filter* to discard macros which have large expanded length before they are stored in the system's knowledge base because such macros tend to have more preconditions and are thus less often applicable.

In some systems selective acquisition filters are used to improve the quality of the solutions and the efficiency of the problem solver. INDUCE 1.2 (Dietterich & Michalski, 1981) learns structural descriptions of concepts using a generate and test strategy. A set of candidate generalizations is generated by dropping single conditions in all possible ways; this set is then filtered using several evaluation criteria including coverage of the training set, specificity of the generalizations, and a user-defined cost.

The principal advantage and the principal limitation of selective acquisition both derive from the fact that it must make evaluations of knowledge before it is used by the problem solver. If the knowledge is harmful it is desirable to remove it before it reaches the problem solver since its use will incur some costs. On the other hand, if the knowledge is evaluated before it has been tried in problem solving, there is no evidence available about how it interacts with other knowledge and affects the behavior of the problem solver, so the selection decision is much less informed. A further limitation of selective acquisition is that it is essentially a hill-climbing technique. For the majority of systems, its most effective role appears to be as a supplementary filter, eliminating obviously harmful knowledge immediately while leaving most of the task to subsequent selection processes.

3.5. Selective retention

Selective retention (also referred to as *forgetting*) is a filtering process in which knowledge may be removed from a knowledge base. As such, it can only be applied to those systems using a knowledge representation scheme in which the knowledge takes the form of elements that may be added or removed individually. The critical difference between selective retention and selective acquisition is that the former assesses the utility of knowledge after it has already been made available for use by the problem solver. Consequently it has more information available upon which to base its estimates of knowledge utility.

As in the case of selective acquisition, retention filters can be classified according to whether the knowledge they select is intended to lead to better solutions or improve the efficiency of the problem solver:

Retention Filtering for Better Solutions: Possibly the earliest example of the use of a selection filter in a machine learning system is the retention filter used in Samuel's (1959) rote learning checker player. This system learned by saving board positions. Because memory was limited, storing a position incurred significant opportunity costs. Samuel's program therefore monitored the frequency with which boards were used and discarded those used least often to create space for new board positions.

Holland's (1975; 1986) genetic algorithm also operates in a system in which storage is limited, but in this case the limit is deliberately imposed so that selective retention, based on "fitness" of individual members of the current set of knowledge elements, will lead to an increase in the average "fitness," where "fitness" is a domain specific function provided by the user. In fact in Holland's system, all knowledge elements are removed at each iteration but those that have been selected contribute to the new population of knowledge elements (in proportion to their "fitness") through a process of recombination while the rest do not.

ID3 (Quinlan, 1986) employs a very severe retention filter. At the end of each iteration the decision tree is evaluated using further training examples; if it is not satisfactory the entire tree is discarded and learning begins again with an augmented training set. The instance based learning system IB4 (Aha, Kibler, & Albert, 1991) maintains records of how successful stored instances have been in making classifications and discards those that perform badly, thus providing a degree of noise tolerance.

The Hypothesis Filtering Method (Etzioni, 1988) is an important generalization of this technique for improving the quality of knowledge produced by a learning system by adding a retention filter. It consists of the specification for a filter that can be attached to the output of any learning module that generates knowledge in the form of testable hypotheses and is guaranteed only to accept hypotheses that are accurate and reliable to an arbitrary degree.[1] Thus the original learning system is converted into a PAC learner (Valiant, 1984). Because the filter operates by repeatedly trying the hypothesis and applying statistical tests to the results, it would be computationally expensive to use in practice.

Retention Filtering for Greater Efficiency: Although all the retention filters described above make some contribution to problem solver efficiency through reduced match time, other systems include retention filters specifically for the purpose of reducing the problem solver's resource requirements. The macro learning system MacLearn (Iba, 1989) simply removes macros that have not been used by the problem solver. Prodigy (Minton, 1988a) makes empirical estimates of the utility of control rules, where utility is defined as:

$$(\text{Mean Time Saving} \times \text{Probability of Application}) - \text{Mean Match Costs}$$

and discards those with negative utility. Note that this method makes no attempt to discover features of knowledge elements that cause them to be harmful: it simply measures how well they perform and discards them if they are harmful.

Some remarkably crude selective retention strategies can be very effective. Funes (Markovitch & Scott, 1988) is a simple macro learner that employs a variety of selective retention filters. A filter that selected on the basis of search effort saved and frequency of use was shown to produce a large increase in problem solver efficiency. However, a repetition of the experiment using a filter that randomly discarded macros showed that the latter was equally effective. The explanation is that after learning many macros, the system has many alternative paths between nodes in the search graph. The degree of redundancy introduced by the learner was very high. Thus the removal of any subset of macros has no effect on system performance because alternative paths exist, but reduces the search time because branching factor is diminished.

As in the case of selective acquisition, there are systems that employ selective retention both to improve the quality of the solutions found and to improve problem solver efficiency. MetaLEX (Keller, 1987) attempts to improve both these aspects of the problem solver performance by eliminating harmful subexpressions from a knowledge base that is provided *a priori*. To do this it makes estimates of both efficiency and effectiveness of the problem solver.

One important advantage of selective retention over selective acquisition is that it allows the learner to take more risks while acquiring knowledge, because acquisition decisions are not final. Rather than deciding immediately, on the basis of little evidence, whether a knowledge element will be beneficial, the learner acquires the item then waits until sufficient evidence has accumulated before deciding whether the item should be retained. DIDO (Scott & Markovitch, 1989b; 1991) is an exploratory learning system that makes extensive use of selective retention in order to exploit this characteristic. New knowledge elements are generated with little computational effort on the basis of small amounts of evidence and those that are incorrect are soon removed. Thus the system is able to achieve an approximate representation of a new domain very rapidly and then proceed to refine it as more evidence is obtained.

Selective retention appears to have been much more widely used than selectively acquisition as a method for eliminating harmful knowledge; probably because it has proved to be more successful. This success can be attributed to the ability of such filters to use three kinds of information about knowledge that are not available to acquisition filters: its direct effect on the behavior of the problem solver, its indirect effects on the problem solver through interaction with other available knowledge, and the frequency with which it is used in finding solutions to problems. This additional information has two costs: first, the period during which harmful knowledge elements impair the problem solver's performance; and second, the overhead of monitoring problem solver performance. Retention filters cannot, in general, produce optimal knowledge bases. In order to do so they would have to estimate the utility of every subset of the current knowledge base, and this is not computationally feasible. Therefore most programs that implement selective retention assume the independence of single knowledge items even though such assumptions are likely to be wrong.

3.6. Selective utilization

One of the factors that influences the utility of knowledge is the problem to be solved. It is therefore possible for knowledge to be very useful in the context of one problem but very harmful in the context of another. In such a situation its average utility might be close to zero and hence it would probably be deleted by a retention filter. Thus the system will lose the benefits it might have obtained when solving those problems for which the knowledge is beneficial.

This type of situation, in which the utility of knowledge varies greatly as a function of the problem to be solved, cannot be handled satisfactorily by any of the filters discussed so far. It requires a selective utilization filter. Such filters differ from those already discussed in that they do not remove knowledge permanently but merely prevent the problem solver from accessing it for some period.

Selective utilization has only recently been identified as a method for reducing the harmfulness of learned knowledge, so there has been little research on the topic. Mooney's explanation based learning system EGGS (Mooney, 1989) includes a utilization filter that is crude but surprisingly effective: the system only uses learned rules that will completely solve a problem, and thus no use is made of learned knowledge to prove subgoals. LASSY (Markovitch & Scott, 1989c), which is discussed in detail in the next section, makes heuristic estimates of the likelihood that goals or subgoals will succeed. Learned knowledge is not used in attempting those goals that appear likely to fail.

IB4 (Aha, Kibler, & Albert, 1991) also makes use of a utilization filter, although it does not base selection on the problem to be solved. As noted above, this system has a retention filter that maintains records of how successful stored instances have been in making classifications and discards those that perform badly. These same records are also used to prevent newly acquired instances from contributing to classification decisions until sufficient evidence has accumulated to demonstrate that they are reliable.

Protos (Porter, Bareiss, & Holte, 1990), like IB4, filters its exemplars such that a new case is matched only with the subset of the exemplars that are most likely to be correct. Like IB4, it uses the record of success in previous classifications, but in addition Protos uses a mechanism called "remindings" that associates, positively or negatively, with various strength, between case features and categories or exemplars. Experiments showed a significant drop in performance (classification accuracy decreased from 100% to 58%) when the utilization filter was disabled.

It would also be possible to view the use of indexing schemes to accelerate lookup in the knowledge base as an example of utilization filtering since they make a subset of knowledge available to the problem solver. However, since the set of matching elements retrieved is not changed by the use of indexing, it is probably more useful to regard the latter as a technique for speeding up matching rather than a knowledge filter. Note that, although indexing can reduce the harm caused by increased matching time, it cannot reduce the much more serious harm caused by increased branching factors.

The principal role of utilization filters is to take account of the dependency of knowledge utility on the problem to be solved by selecting only that portion of the knowledge base that will help in the solution of a given problem. Their main disadvantage is that they contribute to 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.

Finally, it is worth noting that utilization filters may provide an explanation of how humans, whose knowledge bases frequently contain inconsistent facts and beliefs, are nevertheless able to perform deductive inference. A utilization filter that activated consistent subsets of knowledge when solving a particular problem would avoid the problems arising from the fact that anything may be deduced from a contradiction.

3.7. Summary

In this section we have developed a framework for classifying methods for dealing with harmful information in learning systems. This framework, which is based on the stages in the information flow where selection may take place, identifies five types of selection process: selective experience, selective attention, selective acquisition, selective retention and selective utilization. By considering a wide variety of existing systems that incorporate experience or knowledge selection, we have demonstrated the *descriptive* power of the framework. Table 1 provides a summary of the examples discussed.

In our survey we have also explored the advantages and disadvantages of each type of filter. The main conclusions were as follows:

Selective Experience: Useful for saving learner resources and preventing the acquisition of irrelevant knowledge when the space of possible training experiences is large.

Selective Attention: Useful for saving learner resources and preventing the acquisition of irrelevant knowledge when the individual training experiences are complex.

Selective Acquisition: Useful for eliminating obvious harmful knowledge before it reaches the problem solver. Limited by lack of evidence regarding the effect of the knowledge when used.

Selective Retention: Useful for eliminating knowledge whose harmfulness is not dependent on the particular problem being solved. Can make use of wide range of evidence about the effect of knowledge in use, including its interaction with other knowledge.

Selective Utilization: Useful for dealing with knowledge that is beneficial in some contexts and harmful in others.

This information can be used to give the framework a *prescriptive* role by using it to guide the choice of a suitable filter.

4. Selecting the right filter: A case study

Having shown how the information filter framework may be used to a describe and categorize existing systems for eliminating harmful knowledge, we now consider its role in the design and development of learning systems. In order to do this we will describe the steps taken in finding a solution to a problem caused by harmful knowledge that arose during the development of a learning system called LASSY.

4.1. The LASSY system—An overview

LASSY (Learning And Selection SYstem) is a program that learns to improve the efficiency of a Prolog interpreter operating in a particular domain by solving problems chosen from

Table 1. Selection mechanisms in some existing learning systems.

System	Filter	Description	Evaluation Metric
Checker Player (Samuel)	Retention	Discards least useful board positions	Frequency of use
Genetic Algorithms (Holland)	Retention	Randomly retains elements with probability proportional to their fitness	Fitness defined in a domain specific manner
MetaDENDRAL (Buchanan & Feigenbaum)	Acquisition	Attempts to find smallest set of rules that accounts for data	Rules that correctly predict peaks not predicted by other rules score higher
Version Space (Mitchell)	Experience	Chooses experience that will reduce version space by greatest amount	Selects experience that comes closest to matching half remaining hypotheses
ID3 (Quinlan)	Experience	Selects only misclassified instances	Correctness of classification
	Retention	Discards decision trees that do not classify correctly	Correctness of classification
INDUCE (Michalski & Dietterich)	Acquisition	Eliminates candidate generalizations using several evaluation criteria	Includes coverage, specificity, and user defined function
LEX (Mitchell, Utgoff, & Banerji)	Experience	The Problem Generator constructs new practice problems	Prefer problems that will refine partially learned heuristics
	Attention	The Critic marks positive and negative instances in the search area	Select search steps on the lowest cost solutions as positive
MetaLEX (Keller)	Retention	Removes subexpressions that are estimated to be harmful	A weighted combination of estimated cost and estimated benefit
DIDO (Scott & Markovitch)	Experience	Performs experiments on classes with high uncertainty	Prefer experiences involving objects of classes with higher uncertaintities
	Retention	Deletes useless hypotheses	Rule is useless if it predicts the same as inherited rules
PRODIGY (Minton, Carbonell, Gill)	Experience	Generates experiments when discovers incomplete domain knowledge	Incompleteness
	Attention	The OBSERVER selects training example out of the trace tree	Training example selection heuristics eliminate "uninteresting" examples
	Acquisition	Estimates utility of newly acquired control rules and deletes those unlikely to be useful	Eliminate rules whose cost would outweigh saving, even if always applicable
	Retention	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

Table 1. (continued)

System	Filter	Description	Evaluation Metric
MACLEARN (Iba)	Attention	The macro proposer uses peak-to-peak heuristics	Propose only macros that are between two peaks of the heurisic function
	Acquisition	Static filtering. Only macros estimated to be useful are acquired	Redundancy test (primitive) and limit on length and domain specific test
	Retention	Dynamic filtering. Invoked manually	Frequency of use in solution
FUNES (Markovitch, Scott)	Retention	Various heuristics to decide what macros to delete	Random, Frequency of use × Length
CLASSIT-2 (Gennari)	Attention	Attributes with low salience are ignored	Salience
Hypothesis Filtering (Etzioni)	Retention	Runs a test on sample 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
IB4 (Aha & Kibler)	Acquisition	Acquires misclassified instances	Correctness of classification
	Retention	Removes instances that appear to be noisy	Confidence interval of proportions test
	Utilization	Only instances that have proved reliable are used for classification	Confidence interval of proportions test
EGGS (Mooney)	Utilization	Learned macros are used only if they solve the problem	Macros that do not solve the problem worth nothing

that domain. It was developed using the information filter framework and hence includes examples of all five types of selection filter. The full system, which includes two independent learning systems, is quite complex. Here only an overview will be presented with an emphasis on those features of the program necessary for an understanding of the role of the information filters considered. A complete description of the entire system appears in Markovitch (1989).

4.1.1. LASSY's goal: Improving the efficiency of logic programs

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 declarative component and a control component. Ideally, logic programming would always be declarative, but unfortunately this goal cannot be completely realized. Hence both types of logic programming are necessary (Kowalski, 1985).

Many researchers have suggested that logic programming languages such as Prolog could be an ideal tool for implementing intelligent databases (Brodie & Mattias, 1986; Gallaire & Minker, 1978; Parker, Carey, Golshani, Jarke, Sciore, & Walker, 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 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 required control knowledge. The first is to require the user to supply that knowledge. Several Prolog extensions have been built that incorporate facilities that allow the user to specify control (Clark & McCabe, 1979; Gallaire & Lasserre, 1982; Pereira & Porto, 1982; Naish, 1985). The objection to this approach is that the language becomes completely procedural, and its advantage over conventional programming languages disappears. The alternative approach is to build a learning program that can acquire the required knowledge without external help.

LASSY is a system that we have developed to explore the latter approach. It makes use of a variety of learning techniques to acquire knowledge about domains that take the form of Prolog databases. This knowledge is then used to accelerate the search processes of a Prolog[2] interpreter operating on those databases. LASSY was designed to satisfy two fundamental constraints which are necessary if the system is to be of practical use:

Semantic Equivalence: Learning should not change the semantics of the given database; that is, the set of derivable theorems before and after learning should be identical.

User Transparency; The user should not be aware of the existence of the learning system or the use of the knowledge it acquires (except perhaps by noticing changes in the speed of execution).

4.1.2. The architecture of LASSY

The basic architecture of LASSY is shown in Figure 3 using a dataflow perspective similar to that of Figures 1 and 2. It will be seen that the system includes both deductive and inductive learning components. The deductive component is a lemma learning system that acquires lemmas that may be used in subsequent proofs. Unfortunately some of these lemmas

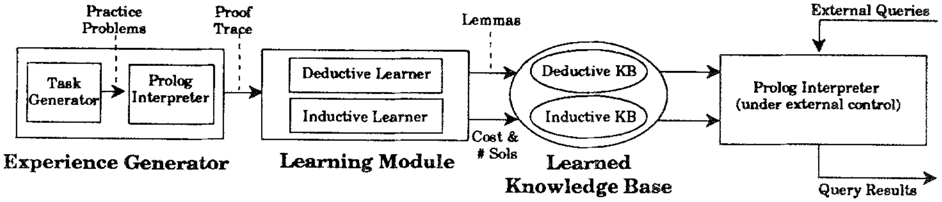

Figure 3. The basic architecture of the LASSY system.

prove to be harmful and it is the use of filters to eliminate such lemmas that forms the subject of this section. The inductive component learns to predict the costs and calling frequencies of different classes of subgoals: this information is then used by the interpreter to re-order subgoals. This inductive system has been described elsewhere (Markovitch & Scott, 1989a; Markovitch, 1989) but will not be discussed further here since our present concern is with the harmful knowledge generated by the deductive learning component.

4.1.3. Lemma learning in LASSY

Lemma learning is a special case of deductive learning, a process in which a system acquires, as additional knowledge, items that are entailed by the application of its inference mechanisms to the knowledge it already possesses. Thus a deductive learning program transforms knowledge from its implicit form to its explicit form, and hence the knowledge added is necessarily both correct and redundant. Since redundant knowledge can be either harmful or beneficial, deductive learning systems are obvious candidates for improvement by use of knowledge filters. Furthermore, such systems can typically generate a great deal of knowledge from a small number of training examples so the need to be selective is particularly acute. There has been extensive interest in deductive learning techniques in recent years (Dejong & Mooney, 1986; Fikes, Hart, & Nilsson, 1972; Iba, 1989; Korf, 1985; Laird, Rosenbloom, & Newell, 1986; Markovitch & Scott, 1988; Minton, 1985; Mitchell, Keller, & Kedar-Cabelli, 1986) so any progress in this area should be of widespread relevance.

Lassy's lemma learning mechanism is straightforward: 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 a *lemma database.* Whenever the Prolog interpreter tries to prove a goal (a new OR node is created), the lemma database is appended before the original 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 would have been used to generate the lemma. No generalization, such as that performed by Prolearn (Preiditis & Mostow, 1987), is implemented in Lassy because such a process would produce lemmas that were more generally applicable, and it would therefore be correspondingly harder to assess their utility. We plan to experiment with filters for generalized lemmas in future versions of the program.

During learning, Lassy receives a set of queries to prove: lemmas proved during the search for solutions are added to the lemma database. The following example shows the effect of unfiltered lemma learning on a trivial Prolog program after the single training query *ancestor(john,A)* has been given to the system:

```
Original Database: parent(peter,john).
                   parent(john,mary).
                   parent(mary,alic).
                   ancestor(X,Y) ← parent(X,Y).
                   ancester(X,Y) ← parent(X,Z), ancester(Z,Y).
```

Lemma Database: *ancester(john,mary).*
ancestor(mary,alice).
ancestor(john,alice).

4.2. Experiment 1: Unfiltered lemma learning

As will be clear from the example above, unfiltered lemma learning will lead to the rapid accumulation of lemmas; thus it is likely that some form of knowledge filter will be needed if lemma learning is to be an effective means of enhancing a Prolog system. Several alternative knowledge filters were investigated through a series of experiments. The problem domain used for all the experiments was a Prolog database that provides a model of a local area network in our laboratory and which is used to assist non-experts in fault finding. It comprises about 50 rules (including recursive rules) and several hundred facts.

The first experiment was run in order to investigate the effect of unfiltered lemma learning on system performance. Since the purpose of Lassy is to improve execution time this provides an obvious measure of performance. We used the number of unifications needed to solve a problem, rather than cpu time, because this provides a machine independent measure. It should be noted that the Prolog interpreter uses an indexing scheme for matching so any detrimental effects on performance are almost entirely due to increased branching factors. All other learning mechanisms and selection processes were turned off for the whole duration of the experiment, which was conducted in the following way:

1. A set of 25 problems was randomly generated to form the *training set.*
2. Another set of 20 problems was randomly generated to form the *test set.*
3. All learning was turned off, and the system executed the whole test set. The total number of unifications performed by the system while solving the whole test set provides a measure of system performance.
4. Lemma learning was turned on and the system executed 5 problems from the training set.
5. Lemma learning was turned off, and the system executed the whole test set. The total number of unifications required was recorded.
6. Steps 4 and 5 were repeated four more times with different training problems.

Figure 4 shows the results obtained. They provide a clear example of harmful knowledge since the use of lemmas degraded system performance by a factor of 2. This may appear a surprising result because the benefits of using lemmas can be very high: instead of proving a subgoal by repeating an extensive search, the program already has the earlier solution available. Since the potential benefits are high, the costs must be even higher to cause such a deterioration in performance. Some of these costs will be a consequence of the additional unifications done when a lemma does not match the current goal, although this effect will be substantially reduced by the first argument indexing of the Prolog interpreter. Clearly this is a situation that calls for the use of filters to eliminate the harmful effects of knowledge.

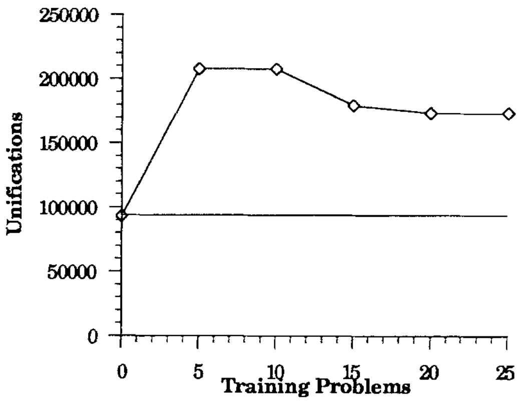

Figure 4. Performance during unfiltered lemma learning. Results indicate the number of unifications required to solve a test set of 20 problems. A smaller number of unifications indicates a better performance.

4.3. Experiment 2: Selective acquisition

This is not a task for either selective experience or selective attention. The number of training examples is small and the procedure used to randomly generate problems ensures that the training and test sets contain similar problems. Furthermore the individual training examples have only a small number of features. Thus the first filter to be considered is selective acquisition.

Since it is clear that the costs of indiscriminate lemma learning greatly outweigh the benefits, an obvious strategy is to restrict acquisition to those lemmas that have potentially large benefits: that is, those that eliminate a large number of unifications when used as part of a solution. The *computational cost* of lemma, defined as the number of unifications performed in the search for a proof of the lemma when it was created, provides a reasonable estimate of this saving.

Experiment 2 was a replication of experiment 1 (using the same training and test sets) with the addition of an acquisition filter that only accepted those lemmas whose proof had cost more than a threshold number of unifications. Three separate runs were carried out with the threshold set at 10, 100, and 1000 unifications. A threshold of 1000 eliminates almost all the lemmas, leaving only 19 of the 117 acquired when no filtering takes place: a significantly higher threshold would have eliminated all of them.

Figure 5 shows the results obtained. The acquisition filters with 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 learning. Raising the threshold to 1000 led to a significant improvement, but the learning curve is very close to the horizontal line that marks the performance with no learning: the harmful effects are eliminated by rejecting almost all the lemmas, leaving little possibility of producing any benefits. Clearly this acquisition filter is not an effective way of dealing with the harmful effects of lemma learning in this system.

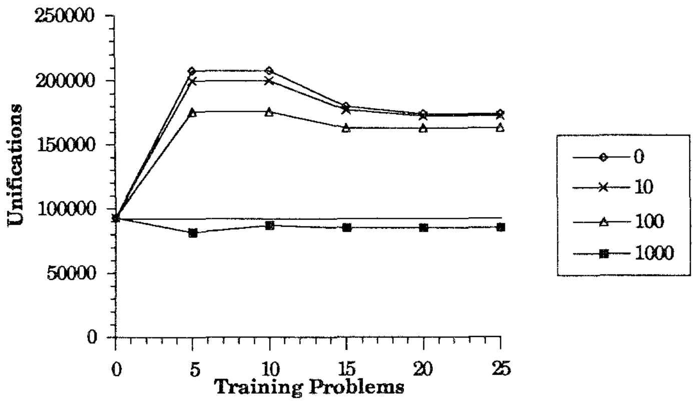

Figure 5. Performance using a selective acquisition filter. Each graph represents an experiment with a different threshold for lemma acquisition.

4.4. Experiment 3: Selective retention

The selective acquisition filter bases its decisions on an estimate of the saving that could be achieved by use of a lemma but takes no account of the frequency with which it is used. This information can only be obtained by making the lemma available to the problem solver and monitoring its use. This is a task for a selective retention filter.

For the next experiment that acquisition filter was removed and replaced with a retention filter that used a heuristic measure of lemma utility combining potential saving and frequency of use. Similar retention filters have been employed by Minton (1988a) and by Markovitch and Scott (1988). Potential saving was estimated using computational cost, as in the acquisition filter. The most obvious estimate of frequency of use is a count but this has the disadvantage of being biased in favor of lemmas that are acquired earlier since they have more opportunities to be applied. We therefore employed an "aging algorithm," a technique widely used in operating systems for virtual memory management and first employed in machine learning by Samuel (1959). Each 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 provides an estimate that is inversely proportional to the probability of the lemma being used in a time period and weighted towards more recent events. The utility of a lemma is then defined as its potential saving divided by its age.

The following experiment was carried out to test the effectiveness of this retential filter:

1. The system was trained using the whole of the training set used in earlier experiments, 117 lemmas were acquired.
2. These lemmas were sorted according to their utility estimates.
3. The 12 lemmas (10% of original set) with lowest value were removed from the lemma database.

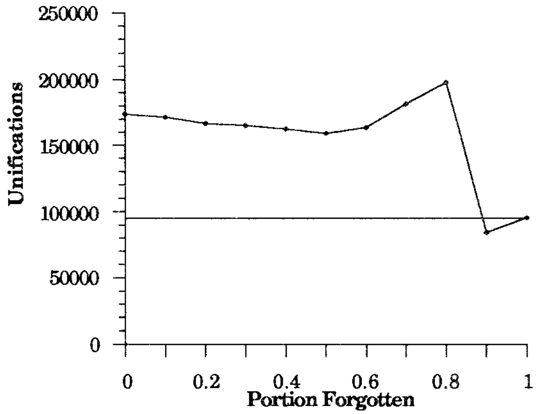

Figure 6. Performance using a selective retention filter.

4. The test set was executed (with learning turned off) and the required number of unifications recorded.
5. Steps 3 and 4 were repeated eight more times.

Figure 6 shows the results obtained. Those for the cases when either all or none of the lemmas were retained are the same as the corresponding results in Experiment 1. The findings are somewhat similar to those obtained for the acquisition filter in that the filter did not succeed in bringing performance to the level achievable with no learning at all until almost all the lemmas had been rejected. The steep drop when the penultimate group of lemmas were removed suggests that most of the detrimental effects are caused by a small group of lemmas that are used frequently. The rise preceding it implies the loss of some beneficial lemmas: this is encouraging because it provides the first evidence we have that some lemmas can be useful. However, it is clear that the selective retention filter is no more effective than the acquisition filter in transforming lemma learning into a method for improving the performance of the Prolog interpreter.

4.5. Selective utilization in LASSY

While the results obtained so far do not actually prove that no retention or acquisition filter could solve this particular harmful knowledge problem, they do suggest it might be more profitable to consider selective utilization. It will be recalled that such filters are appropriate when the utility of knowledge varies very greatly between problems. Closer examination of the results obtained for unfiltered learning reveals that lemmas were extremely harmful when used with queries that failed but that they were actually beneficial in finding solutions to those that succeeded. This difference can be exploited to build a utilization filter. However, before discussing how this can be done, we will consider just why it is that lemmas can be so detrimental in the context of a problem with no solution.

4.5.1. The backtracking explosion

When the interpreter fails to prove a goal using a lemma, the backtracking mechanism forces the interpreter to continue the search using the original facts and rules. If there is in fact no solution this subsequent search must include a regeneration of the lemma: hence that portion of the search performed after the lemma is invoked will be repeated. Furthermore, this duplication of search may occur many times for a given lemma and thus exponentially increase the search time. This problem, which we term the *backtracking explosion*, provides a clear example of harmfully redundant knowledge. The problem is not peculiar to either lemma learning or Prolog. It can arise in any problem solver that uses deductively learned (and hence redundant) knowledge and incorporates backtracking in its search procedure.

Consider for example the search space shown in Figure 7. Assume that the space is searched by a backtracking program that learns the most basic form of macro operator: if during its search the program discovers a path from state S_1 to state S_2, it saves it as a basic transition; thus the next time S_1 is expanded, S_2 will be one of the new nodes generated. Now suppose that, during the search for an earlier goal, the system has already discovered and hence saved the macro *BC*. We now consider what happens when the system is given solvable and unsolvable problems:

Solvable: The system is given the task of getting from *A* to *D* for which a path exists. During its search the program reaches *B*, jumps directly to *C*, and eventually finds *D*, avoiding the search needed to get from *B* to *C*. In this case the macro has been very useful.

Unsolvable: The system is given the task of getting from *A* to *E* for which no path exists. Since the search is exhaustive, at some point *B* will be reached and expanded. *C* will be reached through the macro *BC* and the whole subtree under *C* will be searched. Since

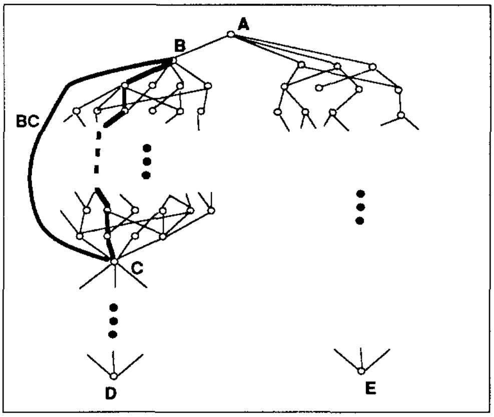

Figure 7. A hypothetical search space to illustrate the problem that arises if backtracking occurs when lemmas are employed.

the search will fail, the program will backtrack to *B* and continue exploring the other alternatives. During this search the program is bound to reach *C* again through the sequence of transitions that originally led to the creation of the macro *BC*. The space under *C* will be explored again, adding to the cost of the search. Furthermore, since the search from *B* will eventually fail, the backtracking mechanism forces the exploration of the whole subspace under *B*, thus every application of a macro in this subspace is bound to be harmful. Thus, in this case, lemmas, far from being useful, have caused a backtracking explosion.

It is important to note that whether the macro *BC* was useful or harmful did not depend on the macro itself but on the context of its use. Furthermore it is clear that the use of macros is *always harmful* when the problem is unsolvable. The size of the backtracking explosion could be reduced if the system checked whether new nodes had already been visited. However, such a scheme suffers from two disadvantages. First, the overhead costs for such checking are large. Second, although this technique would prevent duplication of the search under *C*, it would not prevent the redundant use of the *BC* macro itself and any other macros beneath *B*.

Backtracking explosions appear in the domain used for the experiments with Lassy in the following way. The database used contains the relation *linked* defined as follows:

linked(*X*, *Y*) ← *directly-linked*(*X*, *Y*).
linked(*X*, *Y*) ← *directly-linked*(*X*, *Z*), *linked*(*Z*, *Y*)

The relation *directly-linked* is not provided as a set of ground clauses in the system but is defined by rules in terms of more primitive relations.

Each machine is directly-linked to at most one other machine (*directly-linked* is directional). Suppose there are two disjoint networks as shown in Figure 8. During learning the system will acquire several lemmas of the form *directly-linked(machineX, machineY)* where *machineX* and *machineY* denote constants that identify particular machines. If the query *linked(machinel2, machine42)* is given to the system, it must eventually fail because the two machines are not connected. The greatest distance between *machinel2* and any other machines that it is linked to is 9, so, if no lemmas are used, *linked* will only be invoked 9 times before the interpreter returns with failure. However, if lemmas are used, each link between any two machines will be discovered a number of times using not only

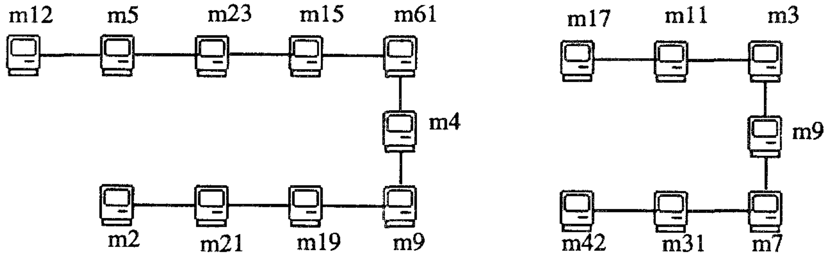

Figure 8. Two disjoint local area networks.

the *directly-linked* rules but also any pertinent *linked* lemmas. If the system has acquired a complete set of *linked* lemmas, there will be 2^9 ways in which the system could discover that *machinel2* is linked to *machine2*. All of these will be explored in the vain attempt to find a route to *machine42*, and the *linked* rules will be invoked between 0 and 9 times for each of them. This example demonstrates how the harmful effects of a backtracking explosion can be exponential in size.

It is now clear why lemmas proved to be so harmful for problems with no solution, even though they were beneficial when a solution existed. It is also clear why neither selective acquisition nor selective retention were successful in filtering out harmful knowledge: the utility of each lemma depends very heavily on the properties of the particular problem to be solved. We now describe how a selective utilization filter can be constructed to deal effectively with this form of problem dependent knowledge utility.

4.5.2. Implementing the utilization filter: Experiment 4

LASSY's utilization filter attempts to minimize the use of lemmas in subtrees below subgoals that are likely to fail. Since it is impossible to know in advance whether a goal will fail, the filter estimates the probability of failures from past experience of similar goals. During learning, the filter maintains records of the frequency of success of each *calling pattern* (Debray & Warren, 1988) of each predicate that appears as a goal or subgoal. 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, *(ancester 0 1)* is the calling pattern for queries that ask for any ancestor of a specified person. These records of calling pattern success rates are used to estimate the probability that a new goal will succeed.

During problem solving, whenever the interpreter creates a new OR node for a subgoal, it estimates the probability that the subgoal will succeed. If the probability of a goal succeeding is below some threshold, the filter disables lemma usage for the entire subtree below that goal. (This is why lemmas are stored separately rather than added to the original database.)

This utilization filter was tested in an experiment that replicated Experiment 1 with the following modifications:

- The procedure that acquires probabilities of success for each calling pattern of each predicate was active during lemma learning, but inactive during testing.
- The utilization filter was added to the system.

Various thresholds were used. Figure 9 shows the results obtained when it was set at 0.5, thus blocking the use of lemmas when the estimated probability of success was below this value. The results obtained for unfiltered lemma learning (Experiment 1) are also shown for comparison. It is clear that this filter has been successful. It has not only eliminated the harmful effects of the lemmas acquired, but also improved the system to a level where performance with lemmas is 2.5 times better than performance without lemmas.

Figure 10 shows the results obtained after all the training examples have been processed for a range of threshold values. As is clear from the rough U shape of the graph, intermediate

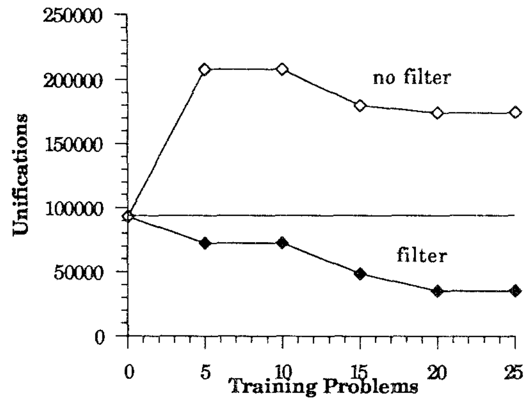

Figure 9. Performance using a selective utilization filter compared with unfiltered lemma learning.

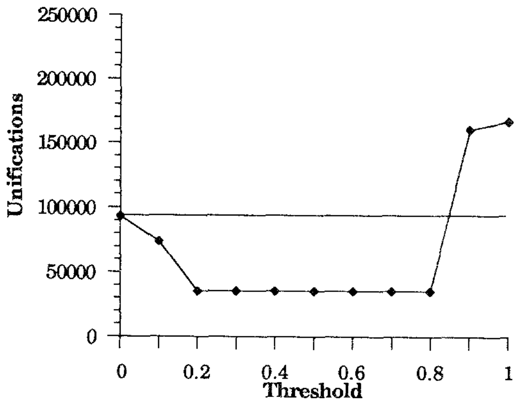

Figure 10. Performance using selective utilization filters with different thresholds for lemma use.

threshold values give the best results. If the threshold is too high, knowledge will not be used where it could be useful; if it is too low, knowledge will be used where it could be harmful. Similar results have been found in other experiments: the basic U shape is always found but the graphs differ in their particulars. For example, the graph shown in Markovitch and Scott (1989c) lacks the flat section in the middle and is thus V shaped, while other experiments have yielded an asymmetric U with the minimum to right or left of the 0.5 threshold.

4.6. Conclusions

The selective utilization filter has provided an effective solution to this particular problem of harmful knowledge. It has transformed lemma learning from a process that degrades problem solver efficiency by a factor of two into one that enhances it by a similar amount. It can of course be argued that this filter is a mere palliative that fails to address the root of the problem: the fact that the original Prolog code is inefficient. However, this argument misses the point of the LASSY system, which is to enable programs to be written in a purely declarative manner without consideration of control flow.

The particular utilization filter we developed to deal with the backtracking explosion is novel. Since the backtracking explosion may arise in any deductive learning system that uses backtracking, this filter has many potential applications. Furthermore, the cost of using it is modest, since all that is required is a table look-up operation each time a goal is set up. Hence the overhead it imposes on problem solving is not likely to outweigh its benefits.[3]

This case study clearly demonstrates the importance of choosing the right filter. The backtracking explosion could not have been eliminated by improvements to either the acquisition or retention filters since its occurrence depends on features of the particular problem to be solved. This example also demonstrates the folly of the argument, which is sometimes advanced, that the way to eliminate harmful knowledge is to make sure it is beneficial when it is acquired.

Thus problems due to harmful knowledge can be addressed systematically by first identifying how the knowledge is causing harm and then using the information filter model to choose an appropriate filter. A crude filter in the right place is likely to be much more effective than a sophisticated one in the wrong place. Conversely, if difficulty is encountered in building an effective filter at a particular stage in a system, this can provide evidence that the harmful effects depend on attributes that are not available to that particular type of filter.

5. Filters as learning systems

Information filters can themselves be viewed as problem solvers, and as such, they require knowledge. Such knowledge can have three different sources: it can be built into the filter, it can come from the learned knowledge base, or it can be acquired by another learning process. Thus a complete system may involve several learning components where some exist to provide knowledge to be used by others. It is therefore useful to introduce the terms *primary learning* and *secondary learning* defined as follows:

Primary Learner: Given a problem solver Ψ, a learning process is called a *primary learner* with respect to Ψ if the knowledge that it generates is made available for use by Ψ.

Secondary Learner: Given a problem solver Ψ, a learning process is called a *secondary learner* if it is not a primary learner with respect Ψ, but is a primary learner with respect to a process that is itself a primary learner with respect to Ψ.

The extension to N-ary learning is obvious. The first condition in the definition of a secondary learner is required to cover the very common situation in which a learning procedure uses knowledge from the main learned knowledge base. Without this first condition, such a learning process would be both a primary learner and secondary learner.

Since experience and knowledge filters exist to enhance the effects of learning processes, they are obvious candidates for use of secondary learning. Most of the systems that incorporate selective retention employ a secondary learning process to accumulate statistics about knowledge usage. Holland's Bucket Brigade algorithm (Holland, 1986) uses a secondary learning process that updates the strength or rules: the primary learner is a genetic algorithm which uses this strength as its fitness measure. The Hypothesis Filtering algorithm (Etzioni, 1988) employs a secondary learning process that applies proposed hypotheses to sample problem set in order to estimate the distance between the hypothesis and the target concepts. These estimates are then used to filter out those hypotheses generated by the primary learner that proved to be non-PAC.

In this section we will survey the relationships between primary and secondary learning in LASSY in order to demonstrate various ways in which filters may make effective use of learning.

5.1. Primary learning and selection filters in LASSY

LASSY's basic architecture was presented in Figure 3. The problem solver is the Prolog interpreter and the system includes two primary learning components that generate knowledge intended to improve the efficiency of this interpreter: a deductive learning component which acquires lemmas, and an inductive component that acquires knowledge used to reorder subgoals. The former was described in Section 4 while an account of the latter, which learns to predict both the cost of solving a goal and the number of solutions it will have, appears in Markovitch and Scott (1989a). A complete account of the entire LASSY system is given in Markovitch (1989).

5.1.1. LASSY's experience generator

In the experiments described in Section 4, a set of training experiences was randomly generated and used in conjunction with each filter. However, in its normal mode of operation, LASSY is capable of generating its own training experiences. This is accomplished by the *task generator* which generates queries that are then solved by the Prolog interpreter: the traces of these proofs form the basic experiences from which both primary learners acquire knowledge. Thus LASSY's normal mode of knowledge acquisition is a form of *learning by experiment*.

In order to generate training examples, the task generator uses a *task model* which represents the distribution of problems that are likely to be presented within the set of all possible problems. The task model takes the form of a weighted set of calling patterns, similar to those used in the utilization filter described in Section 4. The weights indicate the frequency with which queries matching each calling pattern will be asked. The task model also includes a list of the possible ground terms for each argument of each predicate. The task generator will randomly generate queries matching a given calling pattern with a probability that is proportional to its weight. Thus the training experiences will be drawn from the same portions of the problem space as the queries that the system is required to solve, and hence the knowledge generated is likely to be relevant. Note that the criterion used to choose examples for generation can also be used to filter a set of examples supplied by an external agent: in such a situation the task generator has been transformed into a selective experience filter.

It is possible to provide the task generator with a fixed task model: this is in fact how the training and test sets used in the experiments described in Section 4 were produced. However, in normal operation LASSY acquires the task model by maintaining statistics about the queries presented to the system. Thus the system provides itself with training examples that are similar to queries that have been made in the past.

5.1.2. LASSY's selection filters

The complete LASSY system incorporates all of the five types of selective filters. Those in the deductive lemma learning system have already been described and may be summarized as follows:

Experience Filter: Uses a task model to select (or generate) tasks that are similar to those previously experienced. (See Section 5.1.1.)

Acquisition Filter: Rejects lemmas that save a small amount of computation because they have a high likelihood of being harmful. (See Section 4.3.)

Retention Filter: Retains only lemmas that are used frequently and save a significant amount of computation. (See Section 4.4.)

Utilization Filter: Blocks all lemma use for subgoals that are likely to fail. (See Section 4.5.)

The inductive learning component, which acquires knowledge used in subgoal re-ordering, includes three selection filters:

Experience Filter: This filter is common to both the deductive and inductive learning systems.

Attention Filter: The inductive learning system ignores all attributes of a subgoal except its calling pattern.

Utilization Filter: Because subgoal re-ordering is an expensive process, its cost can easily outweigh its benefit. Re-ordering is therefore blocked, and hence the inductively acquired knowledge not used, when the costs accrued in searching for a better ordering approach the expected benefits.

5.2. Secondary learning in LASSY

Of the six selective filters used by LASSY's primary learning components, three make use of secondary learning to acquire knowledge. Consequently, the system includes three secondary learning systems:

The Task Model Learner: This acquires knowledge for the experience filter that is common to both the inductive and deductive primary learning components. It builds a model that reflects the frequency with which various types of query actually arise in ordinary use of the problem solver.

The Lemma Usage Frequency Learner: This acquires knowledge that is used by the selective retention filter that operates on the results of lemma learning. It builds a model that reflects the frequency with which individual lemmas are used.

The Failure Probability Learner: This acquires knowledge that is used by the selective retention filter that operates on the results of lemma learning. It builds a model that reflects how likely it is that an attempt to prove a goal with a particular calling pattern will be unsuccessful.

Figure 11 shows how these secondary learning systems fit into LASSY's primary learning architecture. The selective utilization filter used in conjunction with inductive learning also makes use of learned information, but the information required is the information the primary inductive learning system acquires and so there is no need for a secondary learning system. The remaining filters do not use learned information.

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. Both the task model learner and the failure probability learner ignore all features of subgoals except their calling patterns, and can thus be viewed as having fixed attention filters. Apart from these rather trivial cases, there are no selection processes for the secondary learners.

Thus it can be seen that LASSY's performance is critically dependent on the use of secondary learners to provide the knowledge needed by three of its information filters. It seems likely that, as the central role played by selection in learning receives wider recognition, so the use of secondary learning mechanisms will become more important.

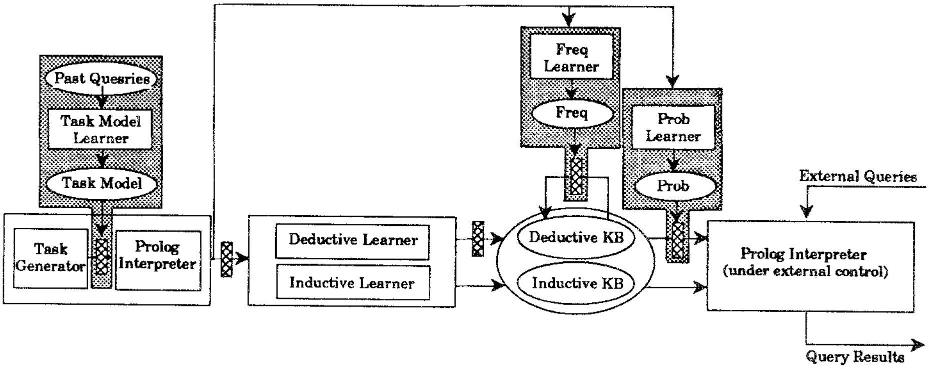

Figure 11. The secondary learning layer of the LASSY system. Areas shaded in gray are the components of the secondary learning system.

6. Discussion

In this concluding section we review the conclusions of the earlier parts of the paper and discuss their implications for further work.

6.1. Conclusions

In the introduction to this paper we set out three goals: to provide a definition of utility of knowledge, to develop a framework for classifying selection processes in learning that serves both descriptive and prescriptive roles; and to demonstrate the use of that framework in the solution of a particular problem caused by harmful knowledge. The definition of the utility of knowledge was developed in Section 2. It provides a considerably generalized version of an earlier definition due to Minton (1988a) which may be applied to any knowledge change and used in conjunction with any performance evaluation function. A definition for the utility of training experiences follows directly from it. This definition should be of value for two reasons: it provides a clear definition of what the term *utility* means that can be used by anyone working on problems concerned with the value of knowledge, and it serves to explicate the dependency of the utility on the problem solver, the problems to be solved and, most significantly, the other knowledge available to the system.

In Section 3 the information filtering framework was introduced as a classification for selection processes in learning. This defined five distinct classes of filter on the basis of their position within a data flow view of a learning system. Their advantages and disadvantages were reviewed and conclusions drawn regarding the types of selection task for which each is best suited: a summary of these conclusions was given in Section 3.7. The descriptive power of the framework was demonstrated by applying it to a wide selection of existing programs that make use of selection processes. Such a framework is of value in that it serves to expose fundamental similarities between superficially very different systems: for example, Holland's Genetic Algorithm (Holland, 1986) and Etzioni's Hypothesis Filter (Etzioni, 1988) are both critically dependent on a selective retention filter. We believe it has a major role to play in making the study of methods of dealing with harmful knowledge a more structured field.

In Section 4 we demonstrated the prescriptive capabilities of the framework by using it while attempting to deal with a problem of harmful knowledge that arose in the lemma learning component of Lassy. Attempts to use acquisition and retention filters proved unsuccessful, and further examination of the way in which lemmas could be harmful revealed that the problem arose when goals had no solution. Since this depends on the particular problem being solved, it can only be satisfactorily dealt with by a utilization filter. Such a filter was developed and proved to be effective. This case study demonstrated the importance of choosing the right type of filter to deal with a particular manifestation of harmful knowledge. The information filtering framework allows such choices to be made in a systematic way.

The particular selective utilization filter developed to deal with the problem that arose in lemma learning is novel and is of interest in its own right. It deals with a harmful consequence of redundant knowledge which we believe has not previously been identified. It can occur in any deductive learning system that uses backtracking and arises when goals fail. Our solution is a novel selection process that learns to predict when goals are likely to fail and blocks the use of lemmas in such circumstances. This use of learning to acquire knowledge used by a process which itself assists another learning process is an example of secondary learning, a concept that was defined and discussed in Section 5.

6.2. Further Work

There are many possibilities for further work within the context of the LASSY system. The system as it stands affords considerable scope for investigation of the effect of different domains and the interactions of different combinations of filters. The primary learning systems could be extended: for example, the present lemma learning could be extended to produce generalized or negative lemmas—changes that would certainly affect the knowledge produced and hence the type of filters required. The filters themselves could be improved. For example, the present utilization filter learns to predict goal failure: if these predictions could be made with greater accuracy, by considering other features of goals in addition to their calling patterns, then the utilization filter would be more effective. Alternatively, the present method of estimating failure probability could be combined with estimates of the costs for a failure and the benefits for a success to produce an expectation value for use of lemmas.

LASSY's present utilization filter could profitably be included in other systems that acquire knowledge deductively. It is possible that many such systems suffer from backtracking explosions, and the inclusion of the filter would alleviate the problem. Certainly the developers of such systems should check for the possibility by comparing the effect of acquired knowledge on goals that succeed and fail, respectively.

There is also a clear need for further study of the ways in which knowledge can be harmful. Filters are only effective as a cure once the nature of the harm is understood. While some harmful consequences of knowledge acquisition have been long understood, we suspect that there may be a number of others that, like the backtracking explosion, have remained unrecognized.

Finally, we should like to see greater recognition of the fundamental role of selection processes in learning. Darwin's great insight was to recognize that attaching a selection process to a source of variation was all that was necessary to produce a learning system. At least two machine learning paradigms demonstrate this: genetic algorithms which were developed from a Darwinian perspective, and instance based learning in which learning is reduced to simple selection. Thus the study of selection processes is a topic that merits as much attention as the study of learning systems themselves.

Notes

[1] Hypothesis filters could be also viewed as acquisition filter since they evaluate hypotheses as soon as the learning component produces them; however, their usage of the problem solver made us classify them as retention filters.

2. The interpreter used is for the POST-Prolog language, a simple extension of Prolog that supports declarative programming by allowing the user to specify only a partial rather than a total ordering of sets of clauses and sets of subgoals (Markovitch & Scott, 1989b).
3. Empirical comparisons based on cpu time confirm this conclusion but are not very meaningful because the entire system, including the Prolog interpreter, is written in InterLisp. Thus benefits (time saving in unification) appear larger than they would be with a more efficient Prolog implementation.

Appendix A: Formal definition of the utility of knowledge

In this section we present formal definitions of the utility of knowledge to problem solving systems. Because knowledge may be represented in many different ways, the most general definition specifies the utility of a change in a knowledge base:

Formal Definition 1 (Utility of a Change in a Knowledge Base)

Let: *K be the set of all possible states of a knowledge base.*

P be a set of problems.

S be a set of solutions. Note that in this context the notion of a solution encompasses all possible consequences of problem solving activity including not only any answers, correct or otherwise, that are found, but also any resources used during the process of problem solving.

$\Psi: P \times K \rightarrow S$ be a problem solver. It takes a problem $p \in P$ and a knowledge base state $k \in K$, and returns a solution $s \in S$.

$V: P \times S \rightarrow \Re$ be an evaluation criterion that evaluates a solution $s \in S$ to a problem $p \in P$. Note that this evaluation criterion may include not only properties of an answer, such as correctness or costs incurred in applying it, but also factors reflecting costs incurred in obtaining it.

Then: *U_Δ, the utility of a change in the knowledge base from state $k_i \in K$ to state $k_j \in K$ for a problem solver Ψ solving problems from problem set P, is a function defined as:*

$$U_\Delta(k_i, k_j, \Psi, P) = \bar{V}(p, \Psi(p, k_j)) - \bar{V}(p, \Psi(p, k_i))$$

where $\bar{V}(p, \Psi(p, k))$ is the expectation value of $V(p, \Psi(p, k))$ when p is chosen randomly from P.

The nature of a change in a knowledge base clearly depends on the representation scheme employed. In some systems, such as connectionist models and Samuel's Checker Player, knowledge is represented as a set of numerical parameters, and hence changes in the knowledge base take the form of changes in parameter values. Other systems use representation schemes in which knowledge consists of a set of knowledge elements that may be added or removed individually: LASSY is an example of such a system. In such systems, it is possible to define the utility of individual knowledge elements in the context of a particular state of the knowledge base:

Formal Definition 2 (Utility of a Knowledge Element)

Let: *K, P, Ψ, and U_Δ be defined as in Definition 1.*

G be a set of knowledge elements such that $K \equiv 2^G$.

Then: *U_ϵ, the utility of a knowledge element $g \in k$, where $k \subseteq G$ is the current knowledge base state of a problem solver Ψ that is solving problems from the problem set P, is a function defined as:*

$$U_\epsilon(g, k, \Psi, P) = U_\Delta(k, k - \{g\}, \Psi, P)$$

A trivial modification of Formal Definition 2 will define the utility of a set of knowledge elements.

The value of training experiences to a problem solver that learns depends upon the effect that learning from those experiences has on the system's knowledge base. Hence the utility of a set of training experiences can be defined as follows:

Formal Definition 3 (Utility of a Set of Training Experiences)

Let: *K, P, Ψ, and U_Δ be defined as in Definiton 1.*

E be a set of potential experiences from which a system may learn.

$\Lambda: 2^E \times K \rightarrow K$ be a learning procedure that, given a knowledge base state $k_i \in K$ and a set of experiences $E' \subseteq E$ returns a knowledge base state $k_i \in K$.

Then: *U_{exp}, the utility of a set of experiences $E' \subseteq E$ for a problem solver Ψ that learns using learning procedure Λ, has a current knowledge base $k_i \in K$, and is solving problems from problem set P, is a function defined as:*

$$U_{exp}(E', k_i, \Lambda, \Psi, P) = U_\Delta(k_i, \Lambda(E', k_i), \Psi, P)$$

Note that these definitions of the utility of knowledge cover only the marginal costs and benefits, including the opportunity costs of storage. As noted in Section 2, there are also fixed overheads covering acquisition and storage costs, and these should be taken into account in a complete appraisal of the worth of acquiring additional knowledge.

References

Aha, D.W., Kibler, D., & Albert, M.K. (1991). Instance-based learning algorithms. *Machine Learning*, *6*, 37–66.

Brodie, M.L., & Mattias, J. (1986). On integrating logic programming and databases. In L. Kerschberg (Ed.), *Expert database systems.* Menlo Park, CA: Benjamin/Cummings.

Buchanan, B.G., & Feigenbaum, E.A. (1982). Foreword. In R. Davis & D.B. Lenat (Eds.), *Knowledge based systems in artificial intelligence*. New York: McGraw-Hill.

Clark, K.L. & McCabe, F. (1979). The control facilities of ic-prolog. In D. Michie (Ed.), *Expert systems in the microelectronic age*. Edinburgh, Scotland: University of Edinburgh.

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.

Dietterich. T.G., & Michalski, R.S. (1981). Inductive learning of structural description: Evaluation criteria and comparative review of selected method. *Artificial Intelligence*, *16*, 257–294.

Etzioni, O. (1988). Hypothesis filtering: A practical approach to reliable learning. *Proceedings of The Fifth International Conference on Machine Learning* (pp. 416–429). 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 (Eds.), *Logic programming*. New York: Academic Press.

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

Gennari, J.H. (1989). Focused concept formation. *Proceedings of The Sixth International Workshop on Machine Learning* (pp. 379–382). Ithaca, NY: Morgan Kaufmann.

Holland, J.H. (1975). *Adaption in natural and artificial systems*. Ann Arbor, MI: University of Michigan 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 (Eds.), *Machine learning: An artificial intelligence approach* (vol. 2). Los Altos, CA: Morgan Kaufmann.

Iba, G.A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Keller, M.R. (1987). Concept learning in context. *Proceedings of Fourth International Workshop on Machine Learning* (pp. 482–487). Irvine, CA: 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. *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 (Eds.), *Machine learning: An artificial intelligence appraoch* (vol. 1). Palo Alto, CA: Tioga.

Lindsay, R.K., Buchanan, B.G., Feigenbaum, E.H., & Lederberg, J. (1980). *Applications of artificial intelligence for organic chemistry: The Dendral project*. New York: McGraw-Hill.

Lloyd, J.W. (1984). *Foundations of logic programming*. New York: Springer-Verlag.

Markovitch, S. (1989). *Information filtering: Selection mechanisms in learning systems*. Doctoral dissertation, EECS Department, University of Michigan.

Markovitch, S., & Scott, P.D. (1988). The role of forgetting in learning. *Proceedings of The Fifth International Conference on Machine Learning* (pp. 459–465). Ann Arbor, MI: Morgan Kaufmann.

Markovitch, S., & Scott, P.D. (1989a). Automatic ordering of subgoals—a machine learning approach. *Proceedings of North American Conference on Logic Programming* (pp. 224–240). Ithaca, NY: MIT Press.

Markovitch, S., & Scott, P.D. (1989b). Post-prolog: Structured partially ordered prolog (TR CMI-89-018) Ann Arbor, MI: Center for Machine Intelligence.

Markovitch, S., & Scott, P.D. (1989c). Utilization filtering: A method for reducing the inherent harmfulness of deductively learned knowledge. *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence* (pp. 738–743). Detroit, MI: Morgan Kaufmann.

Minton, S. (1985). Selectively generalizing plans for problem solving. *Proceedings of The Ninth International Joint Conference on Artificial Intelligence* (pp. 596–599). Los Angeles, CA: Morgan Kaufmann.

Minton, S. (1988a). *Learning search control knowledge: An explanation-based approach*. Boston, MA: Kluwer.

Minton, S. (1988b). Quantitative results concerning the utility of explanation-based learning. *Proceedings of Seventh National Conference on Artificial Intelligence* (pp. 564–569). Morgan Kaufmann.

Mitchell, T.M. (1982). Generalization as search. *Artificial Intelligence,*, *18*, 203–226.

Mitchell, T.M., Keller, R.M., & Kedar-Cabelli, S.T. (1986). Explanation-based generalization: A unifying view. *Machine Learning,*, *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 (Eds.), *Machine learning: An artificial intelligence approach*. Palo Alto, CA: Tioga.

Mooney, R. (1989). The effect of rule use on the utility of explanation-based learning. *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence* (pp. 725–730). Detroit, MI: Morgan Kaufmann.

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

Parker, S.D., Carey, M., Golshani, F., Jarke, M., Sciore, E., & Walker, A. (1986). Logic programming and databases. *Expert Database Systems—Proceedings from The First International Workshop* (pp. 35–48). Menlo Park, CA: Benjamin/Cummings.

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

Pereira, L.M., & Porto, A. (1982). Selective backtracking. In K.L. Clark & S.A. Tarnlund (Eds.), *Logic programming*. New York: Academic Press.

Porter, B.W., Bareiss, R., & Holte, R.C. (1990). Concept learning and heuristic classification in weak-theory domains. *Artificial Intelligence*, *45*, 229–263.

Prieditis, A.E., & Mostow, J. (1987). Prolearn: Toward a prolog interpreter that learns. *Proceedings of The Sixth National Conference on Artificial Intelligence* (pp. 494–498). Seattle, WA: Morgan Kaufmann.

Quinlan, J.R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

Ruff, R.A., & Dietrich, T.G. (1989). What good are experiments? *Proceedings of The Sixth International Workshop on Machine Learning* (pp. 109–112). Ithaca, NY: Morgan Kaufmann.

Samuel, A.L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal*, *3*, 211–229.

Scott, P.D., & Markovitch, S. (1989a). Learning novel domains through curiosity and conjecture. *Proceedings of International Joint Conference for Artificial Intelligence* (pp. 669–674). San Mateo, CA: Morgan Kaufmann.

Scott, P.D., & Markovitch, S. (1989b). Uncertainty based selection of learning experiences. *Proceedings of The Sixth International Workshop on Machine Learning* (pp. 368–371). Ithaca, NY: Morgan Kaufmann.

Scott, P.D., & Markovitch, S. (1991). Representation generation in an exploratory learning system. In D. Fisher & M. Pizzani (Eds.), *Concept formation: Knowledge and experience in unsupervised learning*. Morgan Kaufmann.

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

Tambe, M., & Newell, A. (1988). Some chunks are expensive. *Proceedings of The Fifth International Conference on Machine Learning* (pp. 451–458). Ann Arbor, MI: Morgan Kaufmann.

Utgoff, P.E. (1989). Improved training via incremental learning. *Proceedings of The Sixth International Workshop on Machine Learning* (pp. 362–365). Ithaca, NY: Morgan Kaufmann.

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

Wilkins, D.C., & Ma, Y. (1989). Sociopathic knowledge bases (Technical Report UIUCDCS-R-89-1538). Urbana-Champaign, IL: University of Illinois, Department of Computer Science.

Winston, P.H. (1975). Learning structural descriptions from examples. In P. Winston (Ed.), *The psychology of computer vision*. New York: McGraw-Hill.

Received July 6, 1990
Accepted November 8, 1990
Final Manuscript February 6, 1992