Utilization Filtering : a method for reducing the inherent harmfulness of deductively learned knowledge

Shaul Markovitch
Department of Electrical Engineering
and Computer Science
The University of Michigan
Ann Arbor, MI 48109

Paul D. Scott
Center for Machine Intelligence
2001, Commonwealth Blvd.,
Ann Arbor, Michigan 48105

Abstract

This paper highlights a phenomenon that causes deductively learned knowledge to be harmful when used for problem solving. The problem occurs when deductive problem solvers encounter a failure branch of the search tree. The backtracking mechanism of such problem solvers will force the program to traverse the whole subtree thus visiting many nodes twice - once by using the deductively learned rule and once by using the rules that generated the learned rule in the first place. We suggest an approach called *utilization filtering* to solve that problem. Learners that use this approach submit to the problem solver a filter function together with the knowledge that was acquired. The function decides for each problem whether to use the learned knowledge and what part of it to use. We have tested the idea in the context of a lemma learning system, where the filter uses the probability of a subgoal failing to decide whether to turn lemma usage off. Experiments show an improvement of performance by a factor of 3.

1. Introduction

Most of the work in the field of machine learning has concentrated on trying to create programs that acquire knowledge which is used by some performance system. The main emphasis in this work has been on the acquisition process, and the general belief has been that for correct knowledge, the system's performance improves monotonically as a function of the added knowledge.

Recent works (Minton 1988; Tambe & Newell 1988; Markovitch & Scott 1988a,1988b) have drawn attention to the possibility of correct knowledge being harmful, in the sense that the system's performance without it would be better than the performance with it. One type of knowledge that has been associated with harmfulness is redundant knowledge. Redundant knowledge is not always harmful - all learning systems that use deductive processes for acquiring knowledge are introducing intentional redundancy into the knowledge base. Such redundancy can improve the system's performance significantly by saving the need to deduce again what has been deduced in the past. The reason that redundant knowledge can be harmful is that there are costs as well as benefits associated with using such knowledge. If the costs associated with an element of the knowledge base are larger than its benefits, this element is considered harmful.

This paper is concerned with a particular type of harmful redundancy that occurs in deductive problem solvers that employ backtracking in their search procedure, and use deductively learned knowledge to accelerate the search. The problem is that in failure branches of the search tree, the backtracking mechanism of the problem solver forces exploration of the whole subtree. Thus, the search procedure will visit many states twice - once by using the deductively learned rule, and once by using the search path that produced the rule in the first place.

One existing approach to avoiding harmful knowledge is not to acquire it at the first place (e.g. Minton 1988). The learner tries to estimate whether a newly learned knowledge element is potentially harmful, and rejects it if it is estimated as such. Such an approach is termed selective learning. Another approach is to delete part of the knowledge base which is estimated to be harmful based on past experience (e.g. Holland 1986; Samuel 1963; Minton 1988). That approach is sometimes termed forgetting (Markovitch & Scott 1988a). The problem with these approaches is that they are basically averaging processes - they must decide whether a knowledge element is harmful with respect to the whole problem space that the performance system faces. However, the backtracking problem is an example where the usefulness of knowledge

depends much on the context in which it is being used. Forgetting and selective learning can not account for the case where knowledge is harmful in one context and is beneficial in another.

In (Markovitch & Scott 1989a) we introduce a new framework for classifying methods for reducing harmfulness of learned knowledge called *information filtering* . Information in a learning system flows from the experiences that the system is facing, through the acquistion procedure to the knowledge base, and thence to the problem solver. An information filter is any process which removes information at any stage of this flow.

Information filters can filter the set of experiences that the learning program faces (*selective experience*) or the set of features that the program attends to within a particular experience (*selective attention*).

When the filters process information before it has been transformed to a representation that the problem solver understands we call them *data filters* - the two above filters are data filters. When the filters process information after it has been processed by the knowledge acquisition procedure, we call then *knowledge filters*. Selective learning is a knowledge filter which is inserted between the acquisition procedure and the knowledge base (we call it *selective acquisition*). Forgetting can be viewed as a filter whose input and output are both connected to the knowledge base (we call it *selective retention*).

The method that we suggest for reducing the harmfulness of deductive knowledge when used by backtracking systems falls under the third class of knowledge filters called *selective utilization*. Utilization filter is a filter which is inserted between the problem solver and the learned knowledge base. Knowledge that is not in the space used by the problem solver can not have detrimental effects on the performance of the problem solver. The utilization filter activates only a subset of the knowledge base before solving a problem, trying to deactivate knowledge elements which are estimated by the filter to be harmful within the context of the specific problem.

Information filters can be built into the system, or can be learned. We call the process of acquiring such filters *secondary learning* to differentiate it from the *primary learning* - the process of acquiring knowledge to be used by the problem solver.

We will introduce an instantiation of the backtracking problem in the context of the lemma learning module of the SYLLOG system. We will then describe an implementation of a utilization filter that is used to reduce the harmfulness of lemmas.

Section 2 contains discussion of the harmful aspects of deductive learned knowledge, in particular the backtracking problem and possible remedies to the backtracking problem. Section 3 describes our implementation of a knowledge filter to reduce the harmfulness of lemmas in a Prolog system. Section 4 describes the experiments done with the implementation and the results obtained.

2. The inherent harmfulness of deductively learned knowledge

2.1. Deductive learning

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

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

There are many learning programs that are deductive by nature. All the explanation based learning programs and macro learning programs use such a scheme (e.g. Fikes 73 ; Minton 85; Minton 88; Laird et al. 86; Korf 83; Prieditis & Mostow 1987; Mitchell et al1986; Dejong & Mooney 1986; Markovitch & Scott 1988a). The programs that use generalization are basically equivalent to the more simple schemes, except that they learn sets of explicit derivation rules instead of one rule at a time.

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

2.2. The Backtracking Anomaly

Assume that a search program performs a depth first search (with backtracking) in the

space illustrated in Figure 1. Assume that the program is given a problem to get from state A to state D. Assume that during this search, the learning program learned a new macro - it is possible to get from B to C (we will call this macro/rule B-C). Assume that the problem solver receives another problem - to get from A to E. Assume that there is no route from B to E. The problem solver will get to B and use the macro B-C to get to state C. The search from C will not lead to E, thus the problem solver will backtrack to B. Since there is no route from B to E, the problem solver is bound to search the whole subtree of B, including the search that generated B-C in the first place. The whole subtree under C will be searched twice because the problem solver will get to C twice - once by using the macro B-C, and once by going through the path that generated B-C. Thus, the system would be better off not using the macro in the first place.

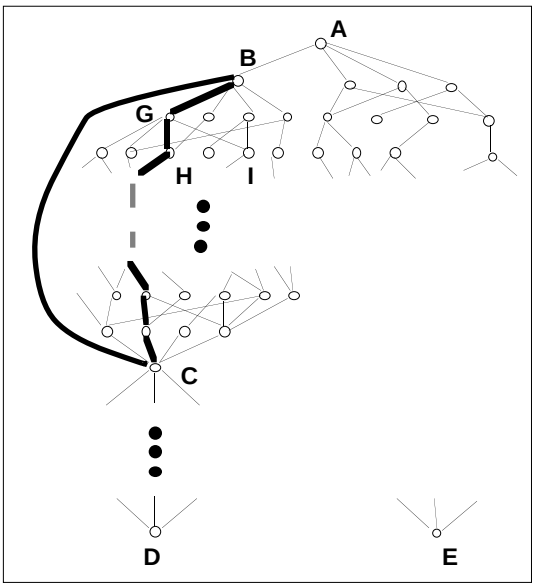

Figure 1

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

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

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

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

2.3. Possible Solutions

In this section we will explore several possible solutions to the backtracking problem.

One possible solution is to add to the problem solver a procedure that checks whether a node has been visited before . In a case like the search problem of figure 1, that will save the program the second search of the subtree of C. There are two problems with adding such a check to the problem solver. The first is that such a test has potentially very high costs in terms of both memory and time. The second problem is that it does not eliminate the problem - only reduces the costs that are caused by the problem (the problem solver would still get to state C twice).

Prolog itself has no facility for implementing such a feature. We modified a Prolog interpreter to allow a version of such a check: Whenever a child of an Or node was returned successfully, the bindings that were generated by the child were compared to the bindings that had been generated before by its siblings. If two sets of bindings were identical, the interpreter marked the new child as failure and went to the next alternative. Such a mechanism would not stop q(X) from generating the same bindings twice, but would save r the necessity to run again with the same bindings . The check improved the performance of the interpreter by a factor of two compared with the performance without the check.

A second possible solution is to use what we term "lemma groups". The idea behind lemma groups is that if an OR node had failed during *learning* time, the learner can be sure that all possible bindings to the subgoal were found, and can learn them all as a group together with a lemma that says "and these are all the possible bindings" (by using the Prolog cut operator). In such a case, there will be no need to go to the rules if all the axioms fail, since the interpreter knows that the rules can not generate new bindings that have not already been tried yet.

Lemma groups can be very beneficial especially for problems with high failure rate. In some cases, for problems that required very large space trees with a high rate of backtracking, using lemma groups made execution up to 30 times faster.

Unfortunately, lemma groups have their own disadvantages. It is extremely hard to maintain lemma groups in a system that is changed over time. If a new axiom is added to the database, there is a possibility that the lemma group is not valid anymore.

The third solution is to use a knowledge filter in order to reduce the probability that lemmas will be used where they can be harmful. Since using lemmas in a subtree of a goal which fails is *bound* to be harmful, the filter tries to turn off lemma usage when it estimates that the probability for such a failure is high.

3. Implementing Knowledge filters

We have investigated the idea of knowledge filters within the context of the lemma learner of our SYLLOG system . The learning system uses inductive and deductive learning mechanisms to accelerate proof generation in Prolog databases. Lemma learning (Kowalski, 1979; Loveland 1969) is one deductive mechanism used to increase the execution speed, and the anomaly described above was encountered during experimentation with lemmas. Our lemma learner differs from PROLEARN (Prieditis & Mostow, 1987) in that it does not perform generalization over the lemmas that are learned.

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

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

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

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

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

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

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

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

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

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

propagated down the subtree of the OR node to turn off lemma usage for the whole subtree.

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

4. Experimental results

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

The problems for learning and for testing were generated randomly from a given problem space. For the experiments described in this section we have used a problem space specified as a list of three elements. The first element is a list of domain names with their associated domains, where each domain is a list of constants from the database. The second element is a list of predicates with a domain specified for each of their arguments. The third elements is a list of problem templates with associated weights. A problem template is a list consisting of a predicate name and a list of flags, one for each argument. A flag of value 1 means that the argument in a problem that satisfies that template should be bound to a constant. A flag of value 0 means that the corresponding argument should be a variable.

To generate a problem from a problem space a problem template is selected with probability proportional to its weight. Then a problem is generated by selecting a random member of the appropriate domain for each of the bound arguments.

An experiment comprises two phases: a learning phase followed by a performance & testing phase. Visiting-check and lemma groups were turned off for the whole duration of this experiment. During the learning phase, a training set of 50 problems was randomly generated in the way described above, and the problem solver was called to solve those 50 problems. During the learning phase the learning program generated two types of information: positive lemmas, and failure probabilities.

During the performance phase, a set of 20 problems was randomly generated using the same method and the same problem space specifications as in the learning phase. The batch of 20 problems was solved, varying values of the probability threshold (the values tried were 0, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 1.01, where 0 threshold means never use lemmas, and 1.01 means always use lemmas).

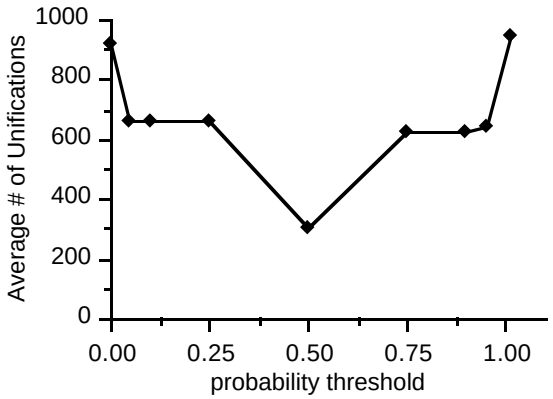

Figure 2.

The results of the experiments are shown in Figure 2. A number of features of the result graph are worthy of note. First, the performance with lemmas (and no filter) was slightly worse than the performance with no lemmas at all. This is another example of knowledge that is harmful. Second, the performance was substantially improved by using knowledge filtering. The performance with filter of 0.5 threshold is three times better than the performance without filter (and also three time better than the performance without lemmas).

The U shape of the graph makes it clear that filtering should not be taken to the extremes. If the filter is too refined, knowledge will not be used where it could be useful. if the filter is too coarse, knowledge will be used where it could be harmful.

5. Conclusions

The problem of inherent harmfulness of deductive knowledge when used within failure branches of the search tree was presented. The existing solutions for reducing harmfulness of knowledge, selective learning and forgetting, could not cure this problem, because the question of whether a piece of knowledge will be harmful does not depend in this case on the knowledge itself, but on the context in which it is being used.

We suggested a different approach, called utilization filtering to reduce the harmfulness of deductive knowledge. The filter that we have implemented uses probability estimates of a goal failing to decide whether to disable lemma usage for the search subtree of that goal.

The experiments we have conducted have shown that utilization filtering is a very effective mechanism for reducing the harmful effects of

knowledge caused by the backtracking problem. The average performance of the system was improved by a factor of 3 using the filter.

The threshold method used here is rather primitive, and we are currently working on a more sophisticated ways of using the failure probabilities. These methods consider the expected benefit from using lemmas in case of a success, and the expected cost of using lemmas in case of failure. In order to use lemmas, the probability of failing multiplied by the expected cost should be lower than the probability of succeeding multiplied by the expected gain.

Utilization filtering is not limited to deductive learning systems. The SYLLOG system also contains an inductive component that uses statistics about the costs of executing goals to reorder subgoals in order to increase efficiency (Markovitch & Scott 1989b). Dynamic ordering is usually needed in such cases, because the expected cost of executing a subgoal depends on what arguments are bound. The problem with dynamic ordering is that it has high cost. A utilization filter turns ordering off if the expected cost is higher than the expected benefit .

References

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

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

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.

Korf,R.E., 1983, *Learning to Solve Problems by Searching for Macro-Operators*, Pitman, Marshfield, Mass.

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

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

Loveland. (1969). A Simplified Format for the Model Elimination Procedure. JACM; July 1969 , 349 - 363.

Markovitch,S. & Scott,P.D., 1988a *The Role of Forgetting in Learning*, Proceedings of Fifth International Conference on Machine Learning, June 1988.

Markovitch, S. & Scott, P.D., 1988b, *Knowledge Considered Harmful*, (TR #030788), Center for Machine Intelligence, Ann Arbor, Michigan

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

Markovitch, S., & Scott, P. D. (1989). Automatic Ordering of Subgoals - a Machine Learning Approach (TR 008). Center for Machine Intelligence, Ann Arbor, Michigan.

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

Mitchell,T.M., Keller,R.M. & Kedar-Cabelli,S.T. , 1986 , *Explanation-based Generalization: A Unifying View* , Machine Learning 1 pp 47-80

Prieditis, A.E. and Mostow, J. PROLEARN: Toward a Prolog Interpreter that Learns. Proceedings of the sixth National conference on Artificial Intelligence, Seattle, WA, 1987.

Samuel,A.L., 1963 , *Some Studies in Machine Learning Using The Game of Checkers* , In *Computers and Thought*, Eds E.Feigenbaum & J.Feldman, McGraw-Hill, New York.

Tambe, M., Newell A., 1988 *Some Chunks Are Expensive*, Proceedings of Fifth International Conference on Machine Learning, June 1988.