On Valid and Invalid Methodologies for Experimental Evaluations of EBL

Alberto Segre*
Charles Elkan
Alex Russell

TR 90-1126
May 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Support for this research was provided by the Office of Naval Research Grant N00014-90-J-1542.

On Valid and Invalid Methodologies for Experimental Evaluations of EBL

Alberto Segre
Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

segre@cs.cornell.edu

Charles Elkan
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

cpe@ai.toronto.edu

Alex Russell
Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

arussell@cs.cornell.edu

Abstract

A number of experimental evaluations of *explanation-based learning* (EBL) have appeared in the literature on machine learning. Closer examination of experimental methodologies used in the past reveals certain methodological flaws that call into question the conclusions drawn from these experiments. This paper illustrates some of the more common methodological problems, proposes a novel experimental framework for future empirical studies of EBL, and presents an example of an experiment performed within this new framework.

Table of Contents

On Valid and Invalid Methodologies for Experimental Evaluations of EBL

Alberto Segre
Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

segre@cs.cornell.edu

Charles Elkan
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

cpe@ai.toronto.edu

Alex Russell
Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

arussell@cs.cornell.edu

1. Introduction

A number of experimental evaluations of *explanation-based learning* (EBL) have appeared in the literature on machine learning. These studies measure the performance of a learning system against the performance of a similar non-learning system. Performance is improved if the learning system can solve more problems or if similar problems are solvable more efficiently with learning; often both are the case.

Reports from previous EBL experiments commonly conclude that the *utility problem* is significant [Minton88, Mooney89, Tambe88]. This problem is that the addition of an EBL component can impose a performance penalty on a problem solver. Inspired by experimental results, the utility problem has been subsequently studied using analytic models.

Questions other than the utility problem can also be addressed experimentally. In particular, experiments have compared different generalization strategies [O'Rorke87, O'Rorke89], protocols for when to apply learning and how to use learned knowledge [Mooney89], methods for learning recursive concepts [Shavlik89, Shavlik90], and whether learned knowledge should eventually be overwritten or discarded [Markovitch88].

How much can conclusions based on experimental observations be trusted? In principle, when the relevant conditions of an experiment are replicated, we should observe similar results. In practice, our confidence depends on how carefully the experiment is designed — in short, our confidence depends on sound *experimental methodology*.

Closer examination of methodologies used in the past reveals certain flaws that call into question the conclusions drawn from these experiments. This paper proposes a novel framework for experimental studies of EBL that overcome these basic methodological flaws. Section 2 illustrates some of the more common pitfalls. Any experimental methodology must depend on assumptions about the phenomenon being studied; Section 3 presents the analytic model of problem-solving systems that underlies our proposed methodology. Section 4 discusses previous experimental work from the perspective of this model, and Section 5 presents a reconstruction of an earlier experiment. When analyzed within our experimental framework, the results of the reconstruction do not support the conclusions drawn from the original experiment. Finally, Section 6 discusses some strengths and limitations of our experimental methodology, as revealed by the experiment of Section 5.

2. Common Pitfalls in Analyzing Performance Data

Consider a hypothetical experiment comparing two versions of the same problem solver, where the second version is augmented with an EBL component. The experiment meters the performance of the two problem solvers on the same set of five problems over two trials. For each attempt at each problem, we record whether or not the problem is solved and how long it takes to get a solution. No learning takes place in the first trial; it is used as a baseline. In the second trial, the problem solver is allowed to apply its EBL component to each successful solution — the newly acquired knowledge is then available to the problem solver when solving subsequent problems in the sequence.

For EBL to make sense as a learning strategy, the problem solver must necessarily be resource limited, since traditional EBL does not increase a problem solver's domain knowledge. EBL, one hopes, accelerates the problem solver's performance by modifying its search space or changing the order in which the search space is explored. Thus, in each trial the problem solver is allotted 1 CPU second to solve each problem presented in sequence.

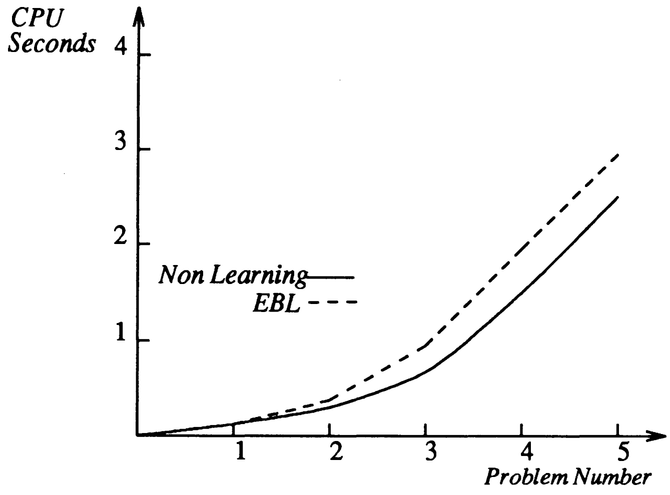
For expository purposes, the data shown in Table 1 is artificial. Real data for a more comprehensive experiment is given in Section 5. At least one subset of that experiment's data (Problems 2.0.8, 2.2, 2.2.4, 2.3 and 2.6) exhibits the same behavior as our artificial data under appropriate experimental conditions.

Table 1
all values are CPU-milliseconds
parenthesis denote unsolved problems

Problem	1	2	3	4	5	Total
No Learning	100	200	300	900	(1000)	2500
EBL	100	275	600	(1000)	(1000)	2975

It is an experimenter's responsibility to summarize data such as that in Table 1 (perhaps through the use of graphs or charts) in a way that generates understanding that can be carried over to the design of analogous systems. Unfortunately, some of the analysis techniques used in previous experiments do not lead to conclusions with *predictive power.* The following replication of some previous analyses shows how it is possible to reach unfounded conclusions about the utility problem.

The simplest way to summarize the data in Table 1 is to sum the amount of time used by each system for all 5 problems. The non-learning system consumes 2.5 CPU seconds, while the EBL system requires 2.975 CPU seconds. By this measure, using EBL entails a 19% performance penalty. We might even plot cumulative solution time against problem number as shown in Figure 1.



Cumulative Time to Solution
Figure 1

Figure 1 appears quite convincing; it seems that our experiment confirms the presence of the utility problem in this domain. Is this conclusion justified? Perhaps not; let us rerun the experiment, extending the resource bound to 1.5 CPU seconds. The data for this set of trials is shown in Table 2.

Table 2
all values are CPU-milliseconds
parenthesis denote unsolved problems

Problem	1	2	3	4	5	Total
No Learning	100	200	300	900	(1500)	3000
EBL	100	275	600	(1500)	1001	3476

Now each system is able to solve four of the five problems. Comparing total CPU usage, we see the EBL system still carries a performance penalty of about 16%. It appears the predictions made previously are confirmed.

However, consider what happens when the resource limit is increased again, to 3 CPU seconds (see Table 3).

Table 3 *all values are CPU-milliseconds* *parenthesis denote unsolved problems*						
Problem	1	2	3	4	5	Total
No Learning	100	200	300	900	(3000)	4500
EBL	100	275	600	1560	1078	3613

Now the EBL system can solve all five problems. The time to solve Problem 5 has increased slightly, presumably due to the extra rule acquired after solving Problem 4. Nonetheless, the non-learning system consumes a total of 4.5 CPU seconds, while the EBL system consumes only 3.613 CPU seconds; a net performance *improvement* of almost 19% for EBL.

What is the methodological flaw that engenders unreliable conclusions? Relying on cumulative resource use produces unfounded predictions, since this performance figure is dependent on the resource limit imposed. As the resource limit is increased, the apparent improvement due to EBL will also increase, depending on the number and distribution of unsolved problems in each trial.

An alternative analysis procedure is to *control for correctness.* Using only those problems solved by both systems in the analysis, we ignore Problems 4 and 5 in Table 1 and obtain cumulative time values of 600 CPU milliseconds for the non-learning system and 975 CPU milliseconds for the EBL system, a 62% performance degradation for the EBL system. Table 2 yields the same figures for Problems 1 through 3, while Table 3 yields 1.5 and 2.535 CPU seconds, respectively; a 69% performance penalty. In all cases, we observe a significant performance degradation for the EBL system.

An analysis that excludes unsolved problems is typically stable across resource limits, but it is inherently biased against a learning system. EBL changes the *resource-limited competence* of a problem solver — *i.e.*, the population of problems which can be solved within a given resource bound (sometimes termed the *resource-limited deductive closure*).[1] Often performance decreases slightly on problems that can be solved without learning; this negative effect is, one hopes, outweighed by the usefulness of solving additional problems. If the analysis is restricted to problems that can be solved without learning, only the negative effect is likely to be observed. EBL's payoff, if any, is to solve problems outside the reach of a resource-limited non-learning system; precisely those problems that are excluded by controlling for correctness.

Consider extending the resource limit once again, this time by a large enough margin so that the non-learning system can solve all five problems (see Table 4).

Table 4
all values are CPU-milliseconds
parenthesis denote unsolved problems

Problem	1	2	3	4	5	Total
No Learning	100	200	300	900	6000	7500
EBL	100	275	600	1560	1078	3613

The non-learning system requires a total of 7.5 CPU seconds, while the EBL system only consumes 3.536 CPU seconds; an overall net performance improvement of almost 53% for the EBL system.

The series of examples above shows how data obtained from the same problem solvers operating on the same problem sets can show both a performance penalty and a performance improvement for EBL. If we are to have confidence in empirical results, we need an experimental methodology designed with extrapolation in mind.

[1] An EBL algorithm may change the order in which operators are considered, or the conditions under which an operator is applicable. EBL may also add new macro-operators to the operator set, or acquire control information that modifies the structure of the search space explored and/or the order in which it is explored.

3. An Analytic Model of Problem Solvers

To make extrapolation possible, we assume that all problem solvers, whether or not they have a learning component, behave according to a simple mathematical model. Any particular problem solver can be described by certain values for the parameters of the model; values which can be determined empirically using data obtained from a trial run over a series of problems. By comparing these parameters for different versions of a problem solver, we can make sound predictions about each version's performance characteristics.

3.1. Basic Assumptions

The foundation of our analytic model is to view problem solving as search. The search space explored by a problem solver is an artifact of the problem being solved, the domain theory used, and the problem solver itself. The most important characteristic of this space is its *effective average branching factor*, which we will denote b. This metric quantifies the number of alternative choices actually explored by a problem solver (as opposed to the number of existing alternative choices) at each point of the search; it is an average taken over all nodes in the search space defined by the problem solver, the domain theory, and the problem being solved. Even if supplied with identical domain theories, different problem solvers may search quite different spaces to solve the same problem. Indeed, a problem solver with an EBL component may be constantly changing the order in which nodes are explored, thus mutating b with experience.

Our basic assumption is that, whatever the details of a particular problem solver, the space it explores with a given initial domain theory is of size exponential in a coefficient capturing the *intrinsic difficulty* of a problem p_i; we will denote this coefficient d_i. This assumption makes explicit our belief in the existence of a (possibly unknown) difficulty value for any problem modulo a particular domain theory. The intuition underlying d_i is that if we know the difficulty of a given problem *a priori*, we should be able to make a prediction about how long it will take a particular problem solver of known b to find a solution.

Under the view of problem solving just described, the time t_i to solve a problem of difficulty d_i using a problem solver/domain theory combination of effective average branching factor b is expected to be:

$$t_i = c \cdot b^{d_i}$$

where c is a normalizing constant. The purpose of an experiment comparing two problem solvers is to determine their respective b and c parameters by measuring t_i over a set of problems of known d_i. Using standard statistical curve-fitting methods, we can then obtain experimental estimates of b and c for each problem solver. If b for one is lower than b for the other, then the first problem solver will operate systematically faster than the second. The notion of faster depends on how we measure t_i; the most obvious metric is CPU time. We will return to this issue below.

In order to collect data points of the form (d_i, t_i) from an experiment, we first need a way to evaluate d_i for each problem p_i. The ideal definition of intrinsic difficulty would depend on only on the initial domain theory and the problem under consideration, and not on any particular problem solver. For example, consider a situation calculus planning domain theory where the number of steps in a plan is proportional to the depth of the corresponding proof. Harder problems correspond to deeper proofs and thus longer plans. Plan length can be used as a difficulty metric; the length of solutions for a given problem population may be generated by the experimenter rather than by the system.

In many domains an implicit difficulty metric is hard to find. In these situations, it is acceptable to use attributes of a solution obtained by a control problem solver as approximations of d_i. Typically one would select the non-learning system as the control system; cost of solution or size of solution generated are often acceptable approximations of d_i. Unfortunately, using this approximation of implicit problem difficulty entails solving every problem with the non-learning system; problems which can't be solved by the non-learning system must be excluded from the analysis resulting in the aforementioned bias against EBL. We return to this subject in Section 5.

3.2. Backward-Chaining Problem Solvers

Domain-independent characterizations of EBL have traditionally used proof trees as a convenient idealization of explanations [Hirsh87, Mitchell86, Mooney86]. Most experimental validations of EBL use backward-chaining definite-clause provers as the underlying performance engine [Mooney89, Prieditis87, Shavlik89], even though most other EBL implementations (*e.g.*, PRODIGY [Minton90], GENESIS [Mooney88], ARMS [Segre88], etc.) rely on more specialized inference engines.

The language for representing the knowledge of a definite-clause prover is that of pure PROLOG; its inference strategy, however, may be significantly more sophisticated. In general, these provers recursively explore an AND/OR tree implicitly defined by the domain theory. To understand this search, it is not sufficient to simply specify the underlying search strategy (*e.g.*, breadth-first, depth-limited depth-first, etc.); for example, heuristic child-node ordering is a factor known to have considerable impact on the cost of constructing proofs.

In order to evaluate the data collection and analyses of previous EBL experiments, it is necessary to go into some detail about AND/OR tree searching. The basic searching operation on a partially-explored AND/OR tree is to *expand* a leaf node of the tree. This operation has three possible outcomes:

- the node can be discovered to be solved;
- the node can be discovered to be failed; or
- the node can be discovered to have children.

In the third case, we say the children of the expanded node are *generated*; new children are new unexpanded leaves. Search algorithms typically differ in how they select which as-yet-unexpanded node to expand next.

Each OR node in the implicit AND/OR tree searched by a definite-clause prover corresponds to a subgoal that must be unified with the head of some matching clause in the domain theory. Each candidate unification represents an alternate path to search for a proof of the subgoal. The operation of expanding an OR-node breaks into two phases. The *indexing* phase consists of retrieving all clauses in the domain theory whose heads unify with the subgoal. If no clauses are found, then the current node can be considered failed. The *sorting* phase orders the child clauses retrieved according to some specified order — for example, place facts first and then other clauses in definition order. Once the candidate clauses are retrieved and sorted, they become available for recursive expansion.

Each AND-node corresponds to the body of a clause in the domain theory. Some clauses correspond to simple facts from the database; these nodes are considered solved. Other clauses consist of a set of sibling *subgoals*. To solve this latter kind of clause, a set of mutually consistent answer substitutions must be found that satisfy each subgoal. Even if the existence of an answer substitution is a statistically independent event for each sibling, the chance that mutually consistent answer substitutions exist for the entire set decreases as if the events were negatively correlated.

3.3. Resource Measures for Backward-Chaining Problem Solvers

The peculiarities of a particular problem-solver implementation affect the choice of dependent variable t_i when devising an experiment. Measuring CPU time makes the experiment dependent on the underlying hardware, as well as implementation aspects of the problem solver that have nothing to do with learning. As an example, consider the results obtained when repeating our hypothetical experiment of Section 2 using an identical resource limit of 2 CPU seconds on three slightly different problem solver architectures (Table 5).

Table 5
all values are CPU-milliseconds
parenthesis denote unsolved problems

Problem	1	2	3	4	5	Total
No Learning	100	200	300	900	(2000)	3500
EBL1	100	250	500	1200	770	2820
EBL2	100	260	540	1336	882	3118
EBL3	100	275	600	1560	1078	3613

The different values were generated using different indexing models for the problem solver; EBL1 assumes constant-time indexing, EBL2 assumes a logarithmic time (in the number of database entries) indexing scheme, and EBL3 (our original problem solver) relies on a linear-time indexing strategy. While constant-time indexing is a bit optimistic, logarithmic time indexing schemes are in common use. If we use CPU time to measure the contribution of learning in EBL3, then we are imposing an implicit bias *against* EBL, as the EBL system is paying a premium price (linear vs. logarithmic) for each rule it adds to the domain theory. A better choice is to break the experiment's dependency on the indexing scheme by using a different attribute for t_i.

One might use the number of nodes expanded or depth of search as a solution cost metric. Unfortunately, using one of these measures usually imparts an unfair bias *in favor* of EBL. Expanding a node involves generating its child nodes; as new rules are added to the domain theory by EBL, the average number of child nodes can be expected to increase.[2] Since expanding a node is not a constant time operation, the time required to search to a fixed depth or to expand a fixed number of nodes will increase. Thus EBL will consume significantly more real time than a non-learning system when expanding a like number of nodes (or searching to equivalent depth). A good attribute for t_i must be impartial to the learning procedure used.[3] Using the number of nodes generated as t_i presents fewer problems, since generating a node can usually be assumed to be a constant time operation.

In summary, the problem of finding an adequate measure of resource use for evaluations of EBL is not an easy one. A survey of the literature in logic programming, search, and automated theorem proving reveals that these communities are also struggling to find adequate performance metrics for benchmarking systems whose search spaces are not fixed (such as when comparing multiprocessor and serial PROLOG implementations). Nonetheless, some performance metrics are clearly better than others, and the use of effective average branching factor seems to be the lesser of evils.

---

[2] In Section 5, we describe some implementation techniques which keep the cost of expanding a node roughly constant.

[3] EBL systems that acquire search control heuristics instead of macro-operators suffer from an analogous problem; as more heuristics are acquired, the time to derive an ordering at each choice point increases.

4. Previous Experiments and their Methodologies

Section 3 presents an analytic model of how problem-solving systems behave. In this section, we examine the methodology of some previous experimental evaluations using the insights gained from this model, deriving some general lessons about collecting useful performance data and analyzing it reliably.

Experimental evaluations of EBL attempt to draw conclusions about changes in the performance of a problem-solving system by measuring a predefined performance metric over several trials. Each trial consists of solving selected problems using a different version of the system. The different versions of the system are chosen to isolate and display the effects of the experimental hypothesis being tested. The objective of any experiment of this type is to produce results that can be used to predict the future behavior of different problem solvers. From this point of view, the critical choices in experimental design concern the resource limit, the performance metric, the domain theory, the problem set, the learning algorithm and the learning protocol.

4.1. Resource Limit

As discussed in Section 2, improving the resource-limited competence of a system implies somehow imposing a resource limit. As explained in Section 3.3, using a limit on search depth or on number of nodes expanded usually introduces a bias in favor of a learning system. On the other hand, a resource limit on number of nodes generated imparts a bias against learning systems, since learning systems are likely to generate more nodes (while still possibly expanding fewer nodes) than their non-learning counterparts. Resource limits based on elapsed CPU time introduce extraneous implementation dependencies. In short, there is no resource limit specification that does not impose some bias on the measurements collected. It is the experimenter's responsibility to minimize this bias.[4]

[4] Some past experiments have imposed multiple resource limits (*e.g.*, limits on both depth and nodes expanded, or on both CPU time and depth). This is usually done so that a depth-first search strategy can be used in search spaces where termination cannot be guaranteed; by specifying a depth limit, the prover is forced always to backtrack eventually. Using more than one resource limit makes results difficult to interpret. A better way to obtain completeness for depth-first search strategies without having to impose an overall depth bound is to use iterative deepening [Korf85].

Among the most complete analyses of the performance contribution of an EBL component to a problem-solving system is O'Rorke's work [O'Rorke87, O'Rorke89], based on a reimplementation of the early *Logic Theorist* (LT) system [Newell63]. In this study, a theorem prover with an EBL component was compared against non-learning and rote-learning versions of the same prover on a population of 92 problems drawn from Russell and Whitehead's *Principia Mathematica* [Whitehead13]. The problems are presented to each system in their original order. For historical reasons, O'Rorke's study relies for the most part on what by today's standards is a quirky, linearly-recursive, breadth-first theorem-proving architecture. While the LT prover's impoverished design makes studying the effects of learning difficult, it has the advantage of being well-specified.

O'Rorke uses a resource limit based on the number of nodes generated as opposed to the number of nodes expanded, imparting a bias against an EBL system. However, the LT prover handles node expansion in a non-standard fashion, attempting to match child nodes to facts in the database as soon as they are generated, rather than when they are recursively expanded. Since LT learning components acquire only new database facts, the cost of generating a node grows with learning. This implies a bias in favor of an EBL system, since it is paying a fixed cost for what is a growing cost operation. These conflicting sources of bias make the evaluation difficult.

4.2. Performance Metric

The problem of choosing a metric to measure the amount of work performed by the problem solver is closely related to the problem of imposing a resource limit. In addition, there is the issue of taking non-homogeneous problem difficulties into account. Previous experimental evaluations of EBL have either held problem population constant across trials, included the cost of failed problems in the analysis, or simply ignored the issue altogether. None of these solutions are adequate.

As discussed in Section 2, controlling for correctness (totaling resource usage for successfully solved problems only) leads to a bias against a learning system. Including the cost of failed problems — *i.e.*, incrementing the total by the resource limit for each failed problem — causes sensitivity to the initial resource bound. Allowing the problem population to differ across trials also usually results in a bias against EBL, since those problems solved by EBL which could not be solved by a non-learning system are often more difficult.

Consider again the hypothetical results of Tables 1 through 3; the inherent difficulty of Problem 5 (as measured by the cost of the eventual non-learning solution in Table 4) far exceeds the total difficulty of Problems 1 through 4, yet Problem 5 is excluded from the analyses. Even the performance measure "average CPU seconds per successful solution" unfairly shows a decrease in problem-solving speed for EBL.

O'Rorke's study uses an approximation of the average branching factor metric, dividing the measured number of nodes generated by the number of nodes expanded for a given problem solver. There are two problems with this approach. First, since LT's notions of generating and expanding a node do not coincide with modern parlance, some of the work usually performed at node-expansion time (*i.e.*, unification with facts in database) is frontloaded onto node generation. Since neither node expansion nor generation are fixed-cost operations in this model, there will necessarily be only a limited correlation between time usage and this approximation of average branching factor. Second (and most important), the denominator reflects the size of the space explored by the current problem solver and domain theory rather than the size of the space explored by a control system. This makes direct comparisons of average branching factor across problem solvers difficult, since there is no normalization at all.

4.3. Domain Theory and Problem Set

One of the hardest problems in devising an EBL experiment is finding an adequately large corpus of problems that can be solved by a suitable domain theory. The requirements for a domain theory and problem set are necessarily vague. It seems clear that a reasonably large set of non-trivial problems is required; the problems may be randomly ordered, or placed by a teacher in a sequence intended to facilitate learning. Some systems may improve their performance in the course of solving a single problem (by learning from solutions to subproblems), while others may learn only from problem to problem. The question of optimal problem ordering and of when to learn from subproblems has not yet been studied experimentally. Theoretical results on the difficulty of unsupervised learning [Valiant84] suggest that ordering may be crucial.

O'Rorke's reconstruction of the original LT experiment applies EBL to a problem set consisting of 92 propositional calculus problems drawn (in their original ordering) from Chapters 2 and 3 of *Principia Mathematica.* Mooney repeats the experiment on a subset of 52 of the problems using a more modern performance-engine architecture. The LT domain has at least two clear advantages. First, there are 92 solvable problems with a relatively small (only 2 rules and 5 facts) domain theory. Second, the problems of *Principia Mathematica* were written for human consumption, and not devised with automated theorem proving or machine learning in mind; no one can claim the problem set was intentionally biased in favor of any problem solver or learning strategy.

Unfortunately, not all domains are well-suited to EBL. The LT domain only supports a specialized form of macro-operator learning that we call *generalized caching.* Since LT involves constructing proofs of properties of propositions and since there are no semantic differences between syntactically different propositions (*e.g.*, P, Q, $\neg R$, etc.), one is forced to generalize from statements about a particular set of objects directly to analogous statements about every possible set of objects. For this reason, macro-operator learning is inappropriate; instead, generalized versions of previously proven propositions are directly cached as new database facts. A sterile domain such as LT, where every object is exactly like any other object, gives EBL algorithms little room to outperform rote learning.[5]

The performance gain achieved due to EBL is also sensitive to the distribution of trial problems. If the choice of each new problem is uncorrelated with the choice of previous problems, then EBL cannot be useful. On the other hand, if only a finite number of problems are ever posed, then one can do no better than memoizing solutions as they are generated for later recall. In general, the assumption normally made

[5] Interestingly enough, since the LT domain theory only consists of two operators, learning search control heuristics would not fare any better.

is that problems are chosen according to some fixed probability distribution, which is unknown but stationary.

In summary, the problem of finding adequate domains for testing EBL is still open. It seems clear that the LT domain presents in some sense a worst-case scenario for EBL performance; other semantically-richer domains may provide a better indication of EBL's power.

4.4. Learning Algorithm and Protocol

To this point, most experiments have compared a particular learning system with a non-learning system. But not all EBL systems are created equal; whether EBL is being used to derive new macro-operators or to build search-control heuristics, the effectiveness of EBL is critically dependent on any operationality pruning performed on the original proof [Elkan89a, Segre87]. Existing experiments document the effect of a particular EBL algorithm with a particular operationality criterion; a different algorithm might display dramatically different results. In addition, as mentioned in the previous section, some domains are a better match to certain specific EBL algorithms; different algorithms may not do as well.

Another parameter which must be taken into account is the learning protocol. This protocol determines how examples are presented and when learning actually occurs. Learning is typically applied to successful problem solutions only; however, in some experiments, unsuccessful problems are entered as facts in the database, perhaps as generalized by a rote-learning algorithm.[6] Not only does rote learning of unproven (and, therefore, possibly untrue) propositions seem an unusual method of augmenting a domain theory, it makes the performance contribution of a particular learning algorithm difficult to isolate.

A slightly different learning protocol has been used in other experiments. In these experiments, problems are divided into separate training and test runs; the system is allowed to learn only from training

[6] This rather strange experimental procedure seems to be a holdover from the original *Logic Theorist* work.

problems, while its performance is recorded only during test problems [Minton88, Shavlik87]. This has two advantages; first, any dependency on problem ordering during the test runs is eliminated, and, second, a close correlation between intrinsic problem difficulty and problem-solver performance should always be observed, since the domain theory is invariant across the test problems. It is hard to obtain an experimental estimate of effective average branching factor when this parameter is constantly changing, as is the case when the results of learning are available to the problem solver in subsequent problems within the same trial.

Other changes in learning protocol might be to learn only from problems of greater than a certain difficulty. A special case of this heuristic is to learn only from problems that cannot be solved by instantiating domain-theory facts. Rote learning from problems harder than a fixed threshold is exactly the caching strategy already in use by theorem provers with some success [Elkan89b]. Finally, note that some learning systems place constraints on the use and management of learned knowledge. For example, Mooney's reconstruction of O'Rorke's LT experiments imposes a chaining constraint on learned rules or macro-operators. Other systems may manage learned rules as a fixed-size cache with various rule-management strategies; new learned rules may cause previously learned rules to be removed from the domain theory.

In summary, whatever conclusions are made from experimental data are critically dependent on the particular EBL algorithm used and the learning protocol followed. It is not clear that any abstraction across protocols or algorithms is possible.

5. LT Revisited, Revisited

In this section, as an example of our suggested methodology, we describe the reconstruction of an experiment designed to test the suitability of an EBL-driven generalized caching strategy on a population of problems drawn from *Principia Mathematica.* This experiment is similar to earlier ones by O'Rorke and Mooney; we use the modernized domain theory presented in [Mooney89]. The original 92 problems used by O'Rorke undergo a similar transformation, resulting in a set of 87 unique problems for each trial run. The ordering of the problems is the original ordering from *Principia Mathematica.*

For our experiments, we have implemented a backward-chaining definite-clause prover in Common Lisp. Our prover supports iterative deepening on search depth, conspiracy size, or both (with conspiracy size preferred). Initial values for depth and conspiracy size, as well as increment values, are specified by the user and are never altered across trials. For the experiments reported here, the starting depth limit and depth increment are set to 1, essentially mimicking a breadth-first search strategy. Breadth-first behavior is necessary to ensure we are getting the shortest solution possible; this is unusually important since we will be using the cost of the non-learning system's solution as a measure of intrinsic problem difficulty.

Resource limits may be specified in terms of depth bounds, conspiracy size bounds and/or number of nodes expanded. For the work described in this paper, the prover operates in an iterative-deepening tree-depth mode with resource limits specified only on the number of nodes expanded; no depth and/or conspiracy-size bounds are given. An absolute resource limit of 2000 nodes expanded for each problem is imposed on the first experiment; subsequent experiments will be run with resource limits of 20000 and 200000 nodes expanded.[7]

The prover allows the user to specify custom child-node ordering functions. The child-node ordering used presents database facts first, followed by learned database entries (in reverse-acquisition order) and the original domain-theory rules in definition order. This child-node ordering was found most effective in preliminary experimental trials with the same domain theory and problem population.

The prover uses a *lazy-unification retrieval strategy* that delays the potentially expensive unification operation from node-generation time to node-expansion time. Node generation uses logarithmic time indexing and relies on a fast match operation that is guaranteed to produce a superset of those database entries which unify with the query. Unification occurs when the candidate choice point is recursively expanded; if the candidate match does not unify with the original query, it is discarded at expansion time.

[7] A resource limit of 2000 nodes expanded is guaranteed to be at least as large as O'Rorke's 2000 node resource limit.

The prover also employs a *lazy child-node generation strategy*; at each choice point, only the leftmost child is actually generated. Each successive sibling choice point is generated on demand at node expansion time.

By using these unusual implementation techniques, we have greatly reduced any experimental bias introduced by a node expansion resource limit. The cost of expanding a node is roughly independent of the actual number of children generated; thus it doesn't change with learning. On the other hand, the cost of expanding a node is still related to the size of the database via the indexing scheme used; our logarithmic time indexing scheme still constitutes a slight bias in favor of a learning system.

We record the number of nodes expanded in each problem solving episode. Since the time required to expand a node is roughly constant, we can rewrite the equation of Section 3.1 as:

$$n_i = c \cdot b^{d_i}$$

where n_i is the number of nodes expanded to solve problem p_i. Taking the logarithm of both sides, we get:

$$\log(n_i) = \log(b) \cdot d_i + \log(c)$$

which is a simple line equation. Given a good measure of d_i, we can find values for b and c using the *method of least squares.*

Since the LT domain does not suggest a problem-solver independent notion of problem difficulty, we use the logarithm of the size of the solution generated by the non-learning system as d_i. Thus:

$$\log(n_i) = \hat{b} \cdot \log(s_i^{nl}) + \hat{c}$$

where s_i^{nl} is the size of the proof generated by the non-learning system for problem p_i, and $\hat{b}$ and $\hat{c}$ are transformed versions of problem solver parameters b and c, respectively.

We can confirm the validity of this measure of d_i for this problem set; if it is indeed a good intrinsic problem difficulty metric, then data collected for the non-learning system should yield a good linear fit to the equation above. The *coefficient of determination* (usually denoted r^2) for the non-learning system's plot thus serves as a check on the quality of this problem difficulty metric. Unfortunately, using this approximation of implicit problem difficulty entails solving every problem with the non-learning system; problems that cannot be solved by the non-learning system must be excluded from the analyses resulting in a bias against a learning system.

This methodology, while much improved over earlier metrics, still fails to take into account the temporal ordering of the problems. The identical problem posed in two different relative positions in the problem set may result, due to the effects of learning, in quite a different number of nodes being expanded. This implies that a learning system will have a much lower degree of fit as quantified by r^2 than a non-learning system, since its b is constantly changing as more and more problems are solved.

Each experiment consists of three trial runs, corresponding to non-learning, rote-learning, and EBL-driven generalized caching systems. Learning was enabled for every non-degenerate solution.[8] The product of learning was a new fact (a new generalized fact in the case of EBL) which was added to the database; as mentioned in Section 3.2.4, macro-operator learning is unsuitable in this domain.

For each problem, the logarithm of the number of nodes expanded was plotted against the difficulty of the problem. The non-learning system solved only 20 of 87 problems, while rote learning solved 34 and generalized caching solved 41. For the reason explained above, only the 20 problems solved by all three systems within this resource bound were used in the curve-fitting process. The least-squares best fit line equations are shown in Table 6, where s^{nl} denotes proof size for a given problem and n denotes number of nodes expanded.

[8] Unlike both the original LT experiments and subsequent reconstructions, no learning occurs on failed solutions.

Table 6

$\log(n) = \hat{b} \cdot \log(s^{nl}) + \hat{c}$

	N	$\hat{b}$	$\hat{c}$	r^2
No learning	20	3.39	0.03	0.94
Rote learning	34	1.75	0.40	0.35
Generalized caching	41	1.73	0.27	0.33

The r^2 value of 94% for the non-learning system indicates that size of proof is a relatively good estimate of problem difficulty for this domain. As expected, there is little difference between EBL and rote learning in the LT domain; each significantly outperforms the non-learning system. The slope coefficients $\hat{b}$ given above are experimental estimates of average branching factors for each search space. Since the slope coefficient for the non-learning system is much higher than the slope coefficient for either the rote learning or the generalized caching system, we can reliably predict that the time to solve harder LT problems with the non-learning system will be significantly higher than the time to solve like problems with either of the learning systems.

All three trials were repeated with an increased resource bound of 20000 nodes. The non-learning system solved 26 of 87 problems, while rote learning and generalized caching systems each solved 46 problems. Curve fitting was repeated on the 26 problems solved by each system in this new set of trials; the results are shown in Table 7.

Table 7

$\log(n) = \hat{b} \cdot \log(s^{nl}) + \hat{c}$

	N	$\hat{b}$	$\hat{c}$	r^2
No learning	26	4.04	-0.30	0.92
Rote learning	46	3.17	-0.15	0.50
Generalized caching	46	3.05	-0.26	0.49

As before, both of the learning systems significantly outperformed the non-learning system. However, in this set of trials we note that the r^2 value is only 92%, indicating that, for these 26 problems, size of proof generated by the non-learning system is not quite as good a measure of problem difficulty. We also note that the experimental values obtained for b grow as the resource limit is extended. This is understandable, for as the resource limit is extended, problems at a given difficulty level which were previously unsolvable are now solved. This will force the slope of the least squares approximation to increase.

The resource limit was again increased by another factor of 10, and the three trials were repeated with a new resource limit of 200,000 nodes. In this set of trials, the non-learning system solved 36 problems, while the learning systems both solved 53 problems. The results of curve-fitting using 36 datapoints per trial are shown in Table 8.

Table 8

$\log(n) = \hat{b} \cdot \log(s^{nl}) + \hat{c}$

	N	$\hat{b}$	$\hat{c}$	r^2
No learning	36	4.68	-0.73	0.93
Rote learning	53	2.99	-0.04	0.44
Generalized caching	53	2.35	0.24	0.30

Again, the results of this experiment show the generalized-caching system outperforms the rote learning system, which in turn outperforms the non-learning system. This conclusion is especially compelling in view of the bias against the learning systems arising from only considering problems solved by the non-learning system. Contradicting earlier findings by Mooney, there is no evidence of the utility problem for generalized caching operating on this problem population. This is not to say the utility problem doesn't exist; however, it does clarify the relation between the utility of an EBL algorithm and the problem population to which it is applied.

In general, how reliable is the experimental methodology of this section? We believe it is highly reliable. The only intrinsic bias comes from excluding 17 datapoints and is against the learning systems; otherwise, our conclusions depend only on the particular EBL algorithm studied (generalized caching) and the experimental domain (LT). The conclusion that EBL is useful in this domain has predictive power.

6. Conclusion

The message of this paper is that in order to support reliable extrapolation of experimental conclusions, experiments comparing different problem solvers should be designed to yield a measure that is invariant to arbitrary impositions of resource limits and to different problem sets. Such a measure can be obtained under the assumption that a problem solver, regardless of the details of its design, explores a search space of size exponential in the difficulty of a problem.

The analytic model that we propose for problem solvers leads to a simple but robust methodology for experimental validations of EBL. This methodology is illustrated with a sample reconstruction of an earlier experiment. The conclusions of earlier studies testing substantially the same hypothesis are not substantiated by our reconstruction; we find that EBL has a net positive effect on the performance of a theorem prover working in the LT domain. It is important to note that even if EBL is only infinitesimally better than a non-learning system (in terms of effective average branching factor), then the EBL system will run indefinitely faster than a non-learning system on difficult enough problems. This claim is formalized as Theorem 4.4 of [Elkan89b].

An interesting implication of the analytic model is to suggest a new direction for theoretical study. If one observes experimentally that a particular learning procedure lowers the effective average branching factor of a problem solver, the obvious next step is to look for conditions under which the observed reduction can be proven to apply.

Another implication of the analytic model is to draw attention to the question of convergence for EBL. Most EBL experiments, including the one reported in Section 5, solve each problem in a trial using knowledge acquired from solving the previous problems in the trial. Thus, if the learning module is effective, the problem solver is improving systematically and its b parameter is not constant. However, b cannot be expected to decrease indefinitely. For a fixed distribution of problems, the problem solver should converge on a stable, improved, search space (an example of a search space that is essentially impossible to improve is given in [Yamada89]). This also explains why the r^2 coefficients of Section 5 are unimpressive — they concern problem solvers with a decreasing b. Altering experimental procedure in order to have separate training and test problem sets should lead to better r^2 coefficients.

We are continuing our experimental study of EBL on this and other problem populations and domains. We are testing the effects of child-node ordering, learning protocols, and the effectiveness of different EBL algorithms using the experimental methodology proposed in this paper. We hope that the analysis of earlier experiments and the experimental methodology advocated in this paper will prompt others to experiment with EBL in diverse domains, and will support reliable extrapolation of their results.

Acknowledgements

Thanks to Paul O'Rorke for providing PROLOG versions of the LT problems. We also thank Oren Etzioni, Vern Paxson, and Devika Subramanian for their comments on an early draft of this paper. Support for this research was provided by the Office of Naval Research grant N00014-90-J-1542.

References

[Elkan89a] C. Elkan and A.M. Segre, "Not the Last Word on EBL Algorithms," Technical Report 89-1010, Department of Computer Science, Cornell University, Ithaca, NY, May 1989.

[Elkan89b] C. Elkan, "Conspiracy Numbers and Caching for Searching And/Or Trees and Theorem-Proving," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989, pp. 341-346.

[Hirsh87] H. Hirsh, "Explanation-based Generalization in a Logic-Programming Environment," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987, pp. 221-227.

[Korf85] R. Korf, "Depth-First Iterative Deepening: An Optimal Admissible Tree Search," *AIJ 27*, 1 (1985), pp. 97-109.

[Markovitch88] S. Markovitch and P.D. Scott, "The Role of Forgetting in Learning," *Proceedings of the Fifth International Machine Learning Conference*, Ann Arbor, MI, June 1988, pp. 459-465.

[Minton88] S. Minton, "Quantitative Results Concerning the Utility of Explanation-Based Learning," *Proceedings of the National Conference on Artificial Intelligence*, St. Paul, MN, July 1988, pp. 564-569.

[Minton90] S. Minton, "Quantitative Results Concerning the Utility of Explanation-Based Learning," *Artificial Intelligence 42*, 2-3 (March 1990), pp. 363-392.

[Mitchell86] T.M. Mitchell, R.M. Keller and S.T. Kedar-Cabelli, "Explanation-Based Generalization: A Unifying View," *Machine Learning 1*, 1 (1986), pp. 47-80.

[Mooney86] R. Mooney and S. Bennett, "A Domain Independent Explanation-Based Generalizer," *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986, pp. 551-555.

[Mooney88] R. Mooney, "A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding," Technical Report UILU-ENG-87-2269, CSL, University of Illinois at Urbana-Champaign, Urbana, IL, January 1988.

[Mooney89] R. Mooney, "The Effect of Rule Use on the Utility of Explanation-Based Learning," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989, pp. 725-730.

[Newell63] A. Newell, J. C. Shaw and H. Simon, "Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics," in *Computers and Thought*, E. Feigenbaum and J. Feldman (editors), McGraw-Hill, New York, NY, 1963.

[O'Rorke87] P. O'Rorke, "LT Revisited: Experimental Results of Applying Explanation-Based Learning to the Logic of Principia Mathematica," *Proceedings of the Fourth International Machine Learning Workshop*, Irvine, CA, June 1987, pp. 148-159.

[O'Rorke89] P. O'Rorke, "LT Revisited: Explanation-Based Learning and the Logic of Principia Mathematica," *Machine Learning 4*, 2 (November 1989), pp. 117-160.

[Prieditis87] A.E. Prieditis and J. Mostow, "PROLEARN: Towards a Prolog Interpreter that Learns," *Proceedings of the National Conference on Artificial Intelligence*, Seattle, WA, July 1987, pp. 494-498.

[Segre87] A.M. Segre, "On the Operationality/Generality Trade-Off in Explanation-Based Learning," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987, pp. 242-248.

[Segre88] A.M. Segre, *Machine Learning of Robot Assembly Plans*, Kluwer Academic Publishers, Norwell, MA, March 1988.

[Shavlik87] J.W. Shavlik and G.F. DeJong, "BAGGER: An EBL System that Extends and Generalizes Explanations," *Proceedings of the National Conference on Artificial Intelligence*, Seattle, WA, July 1987, pp. 516-520.

[Shavlik89] J.W. Shavlik, "Acquiring Recursive Concepts with Explanation-Based Learning," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989, pp. 688-693.

[Shavlik90] J.W. Shavlik, "Acquiring Recursive and Iterative Concepts with Explanation-Based Learning," *Machine Learning 5*, 1 (March 1990), pp. 39-70.

[Tambe88] M. Tambe and A. Newell, "Some Chunks Are Expensive," *Proceedings of the Fifth International Machine Learning Conference*, Ann Arbor, MI, June 1988, pp. 451-458.

[Valiant84] L.G. Valiant, "A Theory of the Learnable," *Communications of the Association for Computing Machinery 27*, 11 (1984), pp. 1134-1142.

[Whitehead13] A. N. Whitehead and B. Russell, *Principia Mathematica*, Cambridge University Press, Cambridge, England, 1913.

[Yamada89] S. Yamada and S. Tsuji, "Selective Learning of Macro-operators with Perfect Causality," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989, pp. 603-608.