From: AAAI-88 Proceedings. Copyright ©1988, AAAI (www.aaai.org). All rights reserved.

Quantitative Results Concerning the Utility of Explanation-Based Learning

Steven Minton
Computer Science Department[1]
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Although previous research has demonstrated that EBL is a viable approach for acquiring search control knowledge, in practice the control knowledge learned via EBL may not be useful. To be useful, the cumulative benefits of applying the knowledge must outweigh the cumulative costs of testing whether the knowledge is applicable. Unlike most previous EBL systems, the PRODIGY/EBL system evaluates the costs and benefits of the control knowledge it learns. The system produces useful control knowledge by actively searching for "good" explanations -- explanations that can be profitably employed to control problem solving. This paper summarizes a set of experiments measuring the effectiveness of PRODIGY's EBL method (and its components) in several different domains.

1. Introduction

The capability to learn and exploit search control knowledge is critically important for domain-independent problem solvers (e.g., theorem provers, planners) due to the exponential size of the search spaces they typically confront. Recent research has shown that explanation-based learning (EBL) is a powerful technique for learning search control knowledge (including macro-operators [3, 7], chunks [13], and search heuristics [11, 14]). However, EBL is not *guaranteed* to improve problem solving performance. Indeed, in many cases performance may even degrade. The problem is that control knowledge has a hidden cost that can often defeat its purpose -- the cost of testing whether the knowledge is applicable as the search is carried out. To actually improve efficiency, an EBL program must generate control knowledge that is *effective* -- its benefits must outweigh its costs. Previous research in EBL has ignored this issue, which I refer to as the *utility* problem; most researchers have simply demonstrated that EBL can improve performance on particular examples. In practice, it is much more difficult to improve performance over a population of examples than it is to improve performance on isolated examples.

This paper discusses the PRODIGY/EBL system, a learning system that searches for effective control knowledge. After a brief discussion of the utility problem, I give an overview of the system, and then focus on a set of comprehensive experiments testing the performance of the PRODIGY/EBL method and its components. The results reveal the significance of the utility issue, and the relative effectiveness of PRODIGY's EBL method. See [9] for detailed descriptions of the experiments, the PRODIGY system and a more formal investigation of EBL and the utility problem.

[1]This research was supported in part by an AT&T Bell Laboratories Ph.D. Scholarship, in part by ONR under Contract N00014-84-K-0415 and in part by DARPA (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499, monitored by the Avionics Laboratory. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of these agencies.

2. EBL and the Utility Problem

Table 2-1 shows a high-level specification of the input and output of EBL, adapted from Mitchell *et al.* [11]. EBL begins with a high-level *target concept* and a *training example* for that concept. Using the *domain theory*, a set of axioms describing the domain, one can explain why the training example is an instance of the target concept. The explanation is essentially a proof that the training example satisfies the target concept definition. By finding the weakest conditions under which the explanation holds, EBL will produce a *learned description* that is both a generalization of the training example, and a specialization of the target concept. The learned description must satisfy the *operationality criterion*, a test which insures that the description will serve as an efficient recognizer for the target concept.

Given:

- **Target Concept: A concept to be learned.**
- **Training Example: An example of the target concept.**
- **Domain Theory: A set of rules and facts to be used in explaining how the training example is an example of the target concept.**
- **Operationality Criterion: A predicate over descriptions, specifying the form in which the learned description must be expressed.**

Determine:

- **A description that is both a generalization of the training example and a specialization of the target concept, which satisfies the operationality criterion.**

Table 2-1: Specification of EBL

As an example (also adapted from [11]) consider the target concept (SAFE-TO-STACK *x y*), that is, object *x* can be safely placed on object *y* without object *y* collapsing. Let us suppose our training example is a demonstration that a particular book, Principles-of-AI, can be safely placed upon a particular table, Coffee-Table-1. If our domain theory contains assertions such as those shown below, we can construct a proof that Principles-of-AI is safe to stack on Coffee-Table-1, because all books are lighter than tables. The resulting learned description would therefore be (AND (IS-BOOK *x*) (IS-TABLE *y*)).

```
DOMAIN THEORY:
  (IS-BOOK PRINCIPLES-OF-AI)
  (SAFE-TO-STACK x y) if (OR (LIGHTER x y)
                             (NOT-FRAGILE y))
  (LESS-THAN w 5-LBS) if (AND (IS-BOOK x)
                              (WEIGHT x w))
  ...
```

The actual purpose of EBL is not to learn more about the target concept, but to re-express the target concept in a more "operational" manner. While the precise definition of

operationality may vary depending on the learning system, efficiency for recognition is normally the implicit basis for any operationality criterion [6]. (Otherwise, if efficiency is of no concern, the target concept could be used as is, since it is exactly defined by the domain theory.)

One can visualize a standard EBL program operating as follows. After being given a training example, EBL produces a learned description which is a generalization of the example. If the next training example is not covered by this description, another learned description is produced. If the third example is not covered by either of the two previous examples, another description is learned, and so on. Thus, the program incrementally re-expresses the target concept disjunctively, where each disjunct is one of the descriptions learned from an individual trial.

Supposedly the operationality criterion insures that each of the individual learned descriptions can be efficiently tested. However, this scheme ignores the cumulative cost of testing the descriptions. Furthermore, as traditionally viewed, the operationality criterion does not consider how the learned description will be used to improve the performance system, which determines its benefit. In practice, as pointed out by Keller [6], the operationality criteria employed by EBL systems to date have been largely unrealistic, or even non-existent.

For these reasons, if we consider EBL systems that learn control knowledge (and almost all implemented EBL systems fall into this category), learning may actually slow down the system. This has been documented in EBL macro-operator learning systems by the author [7], and more recently, in the SOAR system by Tambe and Newell [15] and in the PROLEARN system by Prieditis and Mostow [12]. The degradation phenomenon does not have to be artificially induced in these systems; it can occur under normal operating circumstances. One reason that the problem has not received much attention until now is that extremely few EBL systems have been extensively tested.

2.1. Will the Utility Problem Go Away?

It is sometimes claimed that the utility problem will be "solved" by the development of highly parallel hardware and/or powerful indexing schemes. This opinion is based on the belief that either of these developments would make matching (and by extension, memory search) extremely inexpensive. However, this is unlikely to be true, for the following reasons. First, the learned descriptions produced by EBL are neither bounded in number nor in size. Secondly, matching even a single conjunctive description containing variables is NP-complete [9]. (The complexity of matching may be even worse, e.g. PSPACE-complete, for more complex descriptions [4, p.233].) In the worst case, the behavior of the system may be very poor as the learned descriptions grow in number and size.

Therefore, while fast hardware and good indexing schemes can be extremely useful for matching the learned descriptions, in general, they cannot solve the utility problem. If a learning system can generate arbitrary formulas as search control knowledge, then there will always be potential matching problems. Instead, the solution to the utility problem is to avoid learning overly expensive descriptions in the first place. The system should be sensitive to the costs and savings of the descriptions it learns relative to its computational architecture.

3. Overview of PRODIGY

The PRODIGY system extends the STRIPS problem solving framework [3] by separating search control knowledge and domain knowledge. Thus, instead of relying on complex, built-in search strategies, the PRODIGY problem solver uses an approach we refer to as "casual commitment". If no control rules are present to guide a decision, the problem solver makes a quick, arbitrary choice, employing a simple default control structure. Presumably, if a decision is important, appropriate control rules can be acquired, either automatically or through the user's intervention.

The problem solver's search is conducted by repeating the following decision cycle:

1. A node in the search tree is chosen. A node consists of a set of goals and a state of the world.
2. One of the goals at that node is chosen.
3. An operator relevant to fulfilling the goal is chosen.
4. Bindings for the variables in the operator are selected. If the instantiated operator is applicable, then it is applied to the state, otherwise PRODIGY subgoals on the operator's unmatched preconditions. In either case, a new node is created.

Control rules modify the default behavior by specifying that a particular candidate (a node, goal, operator or bindings) should be either selected, rejected, or preferred over another candidate.

For example, the following control rule is relevant to solving blocksworld problems; it states that if (ON *x y*) and (ON *y z*) are both goals at the current node in the search tree, then the latter goal should be solved first:

```
IF (AND (CURRENT-NODE node)
        (CANDIDATE-GOAL node (ON x y))
        (CANDIDATE-GOAL node (ON y z)))
THEN (PREFER GOAL (ON y z) TO (ON x y))
```

This information is useful because if (ON *x y*) is solved first, subsequently solving (ON *y z*) will undo (ON *x y*). The rule is learned by analyzing a problem solving trace in which the wrong order was attempted first, and explaining why the goal interaction occurred. The language in which the rule is expressed is a form of first-order predicate calculus used throughout the PRODIGY system.

4. The PRODIGY EBL Component

PRODIGY addresses the utility problem by searching for "good" explanations – explanations that result in effective control knowledge. The learning process occurs in three stages. First, after each problem solving episode, the system considers what to explain in the problem solving trace. Second, after constructing an initial explanation, the system considers how to best represent the weakest preconditions of the explanation. The resulting description becomes the left-hand side of a new control rule. Finally, the utility of the rule is measured during subsequent problem solving to insure that it is useful.

4.1. Selecting What to Learn and Generating An Initial Explanation

A significant difference between PRODIGY and most other EBL problem solving systems is the range of target concepts that are employed. PRODIGY's target concepts are meta-level concepts, such as SUCCEEDS, FAILS, and GOAL-INTERACTION, that describe problem solving phenomena. Most earlier EBL problem solving systems, such as STRIPS [3] and LEX2 [10], were limited to learning from solution sequences. Consequently, their target concepts were essentially the same as the goals of the problem solving system. Below I list four types of target concepts currently implemented in PRODIGY. (Each type of target concept has a variation for nodes, goals, operators and bindings.)

1. SUCCEEDS: A control choice (of a node, goal, operator

or bindings) succeeds if it leads to a solution. Learning about successes results in preference rules.

2. FAILS: A choice fails if there is no solution consistent with that choice. Learning about failures results in rejection rules.

3. SOLE-ALTERNATIVE: A choice is a sole alternative if all other candidates fail. Learning about sole alternatives results in selection rules.

4. GOAL-INTERACTION: A choice results in a goal interaction if it causes an achieved goal to be undone. Learning about goal interactions results in preference rules.

The set of target concepts is declaratively specified to the system. Because there can be many training examples for the various target concepts in a single problem solving episode, target concepts are associated with *training example selection heuristics*. When examining the problem solving trace, PRODIGY uses these heuristics to pick out examples that appear to offer the most promise of producing useful control rules. For example, the success of an operator is deemed to be interesting only if other operators failed.

After an example of a target concept is selected, PRODIGY constructs an explanation. Two sets of axioms are employed: a set of architectural-level axioms describing the relevant aspects of the problem-solver and a set of domain-level axioms describing the task domain. Whereas the architectural-level axioms are hand-crafted, the domain-level axioms are automatically derived from the problem solving operators. To construct an explanation, a straightforward algorithm called *Explanation-Based Specialization* (EBS) is used. EBS maps directly from the problem solving trace into an explanation, as described in [8, 9]. No search is involved, since the explanation is determined completely by the problem solving trace. The EBS algorithm then finds the weakest preconditions of the explanation, which constitutes the initial learned description.

4.2. Compression: Improving An Explanation

The purpose of compression is to reduce the match cost of the learned description produced by EBS (and thereby increase the utility of the resulting search control rule). Compression is essentially a simplification process. PRODIGY's compressor module operates on the learned description, first employing partial evaluation [5], then applying domain-independent logical transformations, and finally calling a theorem prover which can take advantage of domain-specific simplification axioms.

To illustrate compression, let us consider a simple blocksworld example. The initial learned description, which states that (ON *x x*) is unachievable, can be simplified as shown below. To do so the compressor employs some simple equivalence preserving transformations and a domain-specific simplification axiom stating that a block is either on the table, on another block, or being held:

```
(FAILS goal node)
if (AND (CURRENT-GOAL node goal)
        (MATCHES goal (ON x y))
        (OR (AND (KNOWN (ONTABLE y))
                 (EQUAL x y))
            (AND (KNOWN (HOLDING y))
                 (EQUAL x y))
            (AND (KNOWN (ON y z))
                 (EQUAL x y))))
reduces to:
(FAILS goal node)
if (CURRENT-GOAL node (ON x x))
```

In addition to simplifying individual descriptions, the compressor can also combine results from multiple examples in order to reduce total match cost. For example, let us suppose that PRODIGY has learned a description stating that a goal (HOLDING *x*) will succeed (i.e., can be achieved) if the block *x* is on the table, and another description indicating that a goal (HOLDING *y*) will succeed if block *y* is on another block. These descriptions can be compressed into a single rule stating that a goal (HOLDING z) will always succeed, since the block z must be either on the table or on another block.

The compressor's task of minimizing descriptions' match cost is, unfortunately, undecidable. To see this, consider that the most inexpensive descriptions to match are (TRUE) and (FALSE). Therefore an optimal compressor would be able to reduce all valid formulas to (TRUE) and all unsatisfiable formulas to (FALSE). However, arbitrary first-order sentences can be represented in PRODIGY's description language, and this task is undecidable for first-order logic.

In fact, PRODIGY's compressor is not guaranteed to minimize match cost. The compressor employs a set of heuristic transformations, each of which tends to reduce match cost. In the first stage of compression individual atomic formulas are transformed to less expensive formulas (e.g., TRUE and FALSE) via partial evaluation. In the second stage of compression, domain-independent logical transformations carry out more complex manipulations such as raising common subexpressions. These first two stages terminate relatively quickly given the set of transformations currently in the system. In the third stage, a simple theorem prover applies optional, user-supplied simplification axioms, each of which encodes a transformation, using a variation of Brown's scheme [1]. Since theorem proving is a potentially unbounded process, PRODIGY will terminate this stage if it exceeds a specified time limit.

4.3. Evaluating the Utility of an Explanation

The utility of a control rule learned by PRODIGY's EBS process is measured in terms of the speed-up resulting from using the rule. Specifically, utility is given by the cost/benefit formula:

Utility = (AvrSavings x ApplicFreq) - AvrMatchCost

where **AvrSavings** is the average savings when the rule is applicable, **Applicfreq** is the fraction of times that the rule is applicable when it is tested, and **AvrMatchCost** is the average cost of matching the rule.

After learning a control rule, the system produces an initial estimate of the rule's utility based on the training example that produced the rule. PRODIGY compares the cost of matching the rule against the savings that the rule would have produced had it been in the system. Only if the new rule appears useful is it included in the active set of control rules. (This eliminates rules that are obviously poor, such as those with extremely high match cost.) During subsequent problem solving, the positive utility estimate is empirically validated by maintaining statistics on the use of the rule. The rule is discarded if its utility is determined to be negative.

5. Performance Results

PRODIGY's EBL learning component has been extensively tested in several domains. These include the blocksworld, an extension of the STRIPS robot domain (including locking doors, keys, and a robot that can push and carry objects), and a more complex machine shop scheduling domain. Scheduling tasks require the problem solver to find a legal schedule for performing a set of operations (involving a LATHE, DRILL-PRESS, GRINDER, etc.) on a variety of objects. In all three

domains PRODIGY's search space tends to grow exponentially with the size of the problem.

There are many problems from each domain in which learning produces exponential speed-up (i.e., after learning, similar problems are solved exponentially faster). However, as argued earlier, the real test of a learning method is whether it improves performance with extended use in a problem solving domain. Consequently, the main experimental results reported here concern the system's performance in each domain after a large sample of training problems have been presented.

To carry out the experiments, procedures for randomly generating problems from each domain were devised (described in [9]). Each procedure includes parameters dictating the maximum size of the problem to be generated. For example, the blocksworld procedure takes two parameters: the maximum number of blocks in the initial state and the maximum number of goals to be achieved.

In each domain, the problem solver was given approximately one hundred training problems to solve. As the training phase progressed, the maximum problem size was gradually increased. Following the training phase, learning was turned off, and the problem solver was presented with one hundred test problems, gradually increasing maximum problem size. For each problem, the problem solver was allowed to run for up to 80 CPU seconds.

Results from the test phase for the three domains are summarized graphically in figure 5-1.[2] For comparison, the graphs show not only the performance of the system with the learned rules, but also the performance of the system without control rules, and the performance of the system with a set of hand-coded control rules. (The hand-coded rules were written by other members of the PRODIGY group, with the author's help.) The graphs show how the cumulative problem solving time grows as the number of problems increases. The cumulative time is the total problem solving time *over all examples* up to that point. Thus, the slopes of the curves are positive because the y-axis represents cumulative time. Because the problems are progressively larger, and therefore tend to be more difficult, the *second derivatives* of the curves are also positive.

The table below shows the number of test problems that remained unsolved in each domain within the 80 CPU second time limit:

	Blocks domain	STRIPS domain	Scheduling domain
With hand-coded rules	0	1	4
With learned rules	2	3	7
Without rules	19	49	32

Numbers of Unsolved Problems

The relatively large proportion of unsolved problems for the "without rules" condition means that search was frequently being cut off at 80 CPU seconds. This explains why the curves for the system running without control rules become relatively flat (approaching linear) as the problems get large, rather than increasing exponentially.

[2]Due to space limitations, our discussion centers on the performance during the test phase, since the training phase (and the one time expense of learning) is assumed to be of lesser importance. It is worth mentioning, however, that the time spent learning was typically of the same order of magnitude as the time spent problem solving. However there was significant variation from problem to problem. In some cases learning took much less time than problem solving, on other occasions the opposite was true.

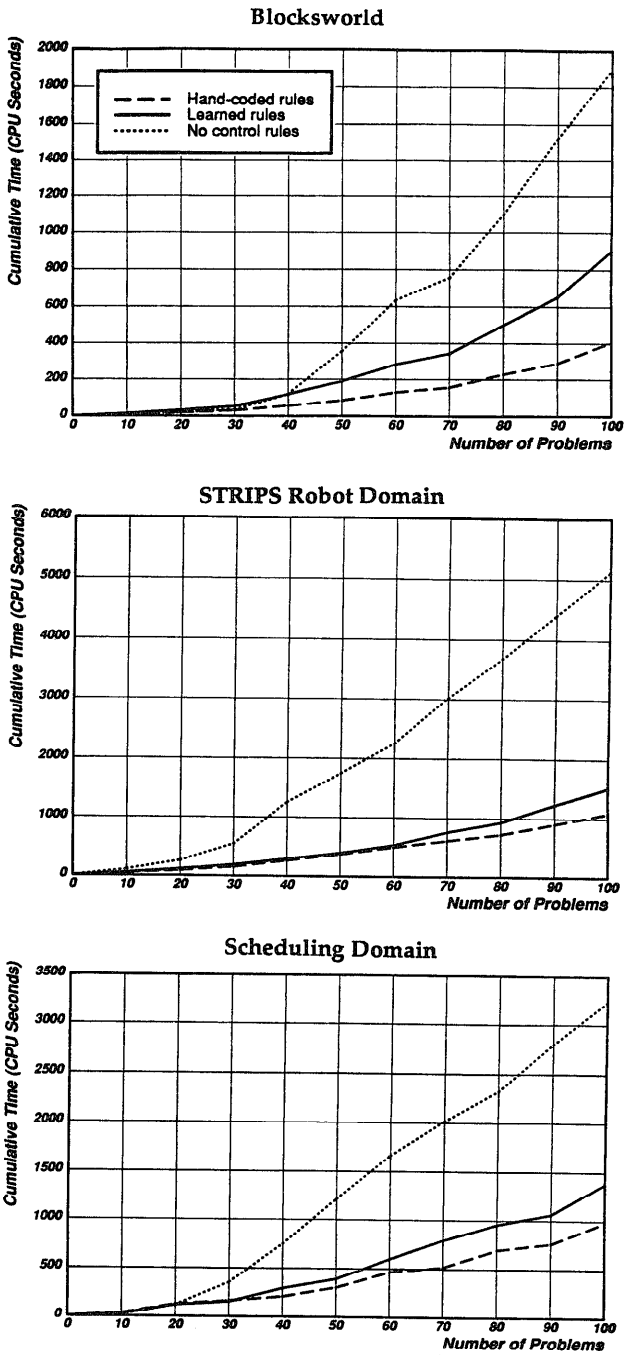

Figure 5-1: Overview of Test Phase Results

The results from all three domains show the same general trends. In each case, the system performed approximately 50-100% worse (in terms of cumulative problem solving time) with the learned rules than with the expert hand-coded rules, but much better than without control rules (although there was significant variation on the individual problems). The difference between learning and no-learning was especially dramatic in the STRIPS robot domain. Table 5-1 lists more detailed performance results from the STRIPS domain test phase. The table indicates the time (in CPU seconds) to solve

each problem without control rules, with the learned rules, and with the hand-coded rules. In addition, the table indicates the number of nodes expanded by the problem solver for each condition and next to this, in parentheses, the number of nodes on the solution path. (The number of nodes on the solution path is approximately twice the number of operators in the solution.)

	No cntrl rules		Learned rules		Hand-coded rules	
Prob name	time	nodes	time	nodes	time	nodes
SW-TST5	13.6	142 (32)	8.3	32 (32)	7.2	32 (32)
SW-TST10	0.9	6 (6)	1.2	6 (6)	1.0	6 (6)
SW-TST15	1.0	8 (8)	1.6	8 (8)	1.4	8 (8)
SW-TST20	16.3	222 (14)	4.7	14 (14)	3.8	14 (14)
SW-TST25	19.5	252 (18)	4.6	18 (18)	3.7	18 (18)
SW-TST30	80.0	843 (*)	12.9	42 (42)	11.3	43 (42)
SW-TST35	25.4	291 (10)	6.9	19 (10)	8.8	40 (10)
SW-TST40	80.0	955 (*)	9.8	30 (30)	16.0	70 (30)
SW-TST45	80.0	840 (*)	20.2	52 (52)	25.3	99 (52)
SW-TST50	80.0	872 (*)	16.1	46 (46)	18.7	77 (46)
SW-TST55	31.3	332 (26)	9.3	26 (26)	8.8	34 (26)
SW-TST60	65.0	593 (50)	20.4	52 (50)	14.0	51 (50)
SW-TST65	80.0	912 (*)	14.6	38 (38)	11.8	39 (38)
SW-TST70	80.0	588 (*)	4.6	12 (12)	3.3	12 (12)
SW-TST75	80.0	742 (*)	27.7	55 (30)	26.4	81 (30)
SW-TST80	80.0	753 (*)	19.3	43 (34)	13.6	46 (34)
SW-TST85	44.8	486 (14)	6.0	14 (14)	4.2	14 (14)
SW-TST90	41.9	397 (14)	6.8	14 (14)	7.5	26 (24)
SW-TST95	80.0	712 (*)	40.4	92 (60)	13.4	40 (40)
SW-TST100	80.0	793 (*)	30.6	67 (60)	22.7	66 (60)

* No solution found within time limit

Table 5-1: Sample Data from STRIPS Domain Test Phase

Why were the hand-coded rules more effective than the learned rules? The following table is illuminating. It shows for each condition the number of nodes explored and the average time necessary to examine each node, over all 100 test problems in the STRIPS domain.

	Total nodes explored	Avg. time per node
With hand-coded rules	3,992	.274 seconds
With learned rules	4,038	.379 seconds
Without rules	52,821	.098 seconds

STRIPS Robot Domain, Performance Statistics

The data indicates that the learned rules and the hand-coded rules were of comparable power; the total number of nodes explored in each case was almost identical(approximately a factor of 13 fewer nodes than were explored without control rules). However, the use of control rules, both learned *and* hand-coded, increased the time necessary to expand each node due the extra processing time requred to match the control rules. While the extra cost was worthwhile, notice that the hand-coded rules were less expensive to use than the learned rules, accounting for their better performance. In summary, it would appear that for this domain the hand-coded rules and the learned rules encoded roughly equivilent knowledge, but the learned rules did not express the knowledge as efficiently. Presumably, this indicates that the compression process can be improved. It is difficult to pin the blame solely on the compressor, however, since the its performance is highly dependent on the initial description produced by EBS.

The results from the blocksworld were similar, in that the learned rules were approximately equivalent in coverage to the hand-coded rules, but more expensive to use. Interestingly, the results from the scheduling domain, shown below, tell a slightly different story.

	Total nodes explored	Avg. time per node
With hand-coded rules	1,970	.497 seconds
With learned rules	3,029	.454 seconds
Without rules	33,790	.096 seconds

Scheduling Domain, Performance Statistics

In this case, the learned rules were slightly *more* efficient to evaluate than the hand-coded rules, but the coverage of the learned rules was poorer, in that they did not focus the search as well as the hand-coded rules. It appears that the learned rules were less general than the hand-coded rules.

Finally, in evaluating the effectiveness of the learning mechanism, it is worth noting that the learned rules were guaranteed to be correct, whereas the hand-coded rules contained several errors. The errors were noticed only when the solutions using the learned rules and hand-coded rules were compared. (The hand-coded rules were corrected before generating the final results shown above.) Errors in the hand-coded rules were especially likely to crop up in the more complex, less-intuitive domains such as the scheduling domain. Coding control rules can be a time-consuming, tedious process even for experts; in the scheduling domain the task took approximately eight hours.

5.1. Evaluating the System's Components

Analysis of the three most significant aspects of the PRODIGY/EBL system (the use of multiple meta-level target concepts, compression analysis, and utility evaluation) reveal that they each contribute substantially to the overall effectiveness of the learning process.

A rough indication of the relative utility of the various target concepts can be gained by comparing the relative number of rules learned from each target concept that were empirically found to be useful, as shown below.

	Blocksworld	STRIPS domain	Scheduling
Succeeds	1	6	6
Fails	11	12	10
Sole-alternative	4	6	4
Goal-interaction	3	6	17

Breakdown of Rules Learned by Target Concept Type

To measure the contribution of compression, the blocksworld training phase and test phase problems were re-run with parts of the compressor "turned off". As previously described, compression occurs in three successive stages: partial evaluation, domain-independent transformations, and domain-specific transformations. Unfortunately, the first stage, partial evaluation, cannot be turned off because it is interleaved with (and necessary for) the EBS explanation process. With stages two and three turned off during learning, performance dropped to a point where it was slightly worse than without learning. With only the third stage of compression turned off, the system performed 40% worse in the test phase (but better than without learning). Compression has such a significant effect because the explanations generated by PRODIGY can be verbose and repetitive, just as one might expect machine-generated proofs to be. This is especially true when learning from failures and goal-interactions, since entire subtrees may have to be explained.

The contribution of utility evaluation is summarized by the table below. It lists the total number of training example/target concept pairs that were explained via the EBS algorithm in the training phase for each domain. It also indicates how many of the resulting control rules were initially estimated to be useful, and the number of these control rules which passed empirical utility validation.

	Blocks domain	STRIPS domain	Scheduling domain
Examples analyzed	328	588	946
Cntrl rules estimated useful	69	130	243
Cntrl rules found to be useful	19	30	37

To test the contribution of empirical utility validation, the blocksworld experiments were re-run without the validation step, so that the system saved all rules estimated to be useful. Performance was approximately sixty-five percent poorer than with the original learned rules. (When the utility estimation step was also left out, performance deteriorated so rapidly that the full test phase could not be completed due to resulting technical problems.)

6. Comparison with Macro-Operators

The learning system was also compared against traditional EBL macro-operator techniques. Both selective [7] and unselective [3] macro-operator learning techniques were tested. It was found that the control rules produced by PRODIGY's EBL component were, in general, considerably more effective in improving problem solving performance than the learned macro-operators. In some cases, macro-operator learning resulted in overall performance degradation, as predicted. Surprisingly however, in the blocksworld, selective macro-operator learning was as effective as PRODIGY'S EBL method, although the solutions found using macro-operators were almost fifty percent longer (and thus extremely suboptimal). A close analysis of the results revealed the reason. Normally, the PRODIGY planner backtracks whenever a state-cycle occurs (e.g. the planner puts block A on block B, and then removes block A from block B). This helps the problem solver avoid suboptimal solutions. In the blocksworld, where state-cycles are frequent, significant backtracking may be required before a cycle-free solution is found. When using macro-operators the system "jumped" over intermediate states where cycles would have normally been detected. In effect, the macro-operators caused the system to trade solution quality for search time.

7. Discussion

I have argued that an EBL system must be sensitive to both the costs and benefits of knowledge if it is to have a positive influence on performance. The empirical results reported here demonstrate that, for the three domains investigated, the PRODIGY/EBL system can improve problem solving performance over large sets of problems. Moreover, the experiments show that without the mechanisms used by PRODIGY, EBL may not be useful.

It is worth noting, however, that experimental evidence has its limitations. For example, the performance of a problem solver and/or learning system depends greatly on the domain and its specification. Theoretically speaking, there is no such thing as an "average domain", any more than there is an "average integer" or "average program" (all of which are unbounded sets). So, while I have attempted to investigate varied domains, it is apparent that better methods for characterizing and comparing domains must be developed. (Initial steps towards describing the types of domains for which PRODIGY's EBL method is useful are described in [9].) Furthermore, this research also illustrates the difficulty of comparing learning methods, as evidenced by the subtle "optimality vs. search time" effect encountered when comparing the PRODIGY/EBL method with macro-operator learning.

There are clearly difficulties in comparing, replicating, and evaluating empirical studies in machine learning. However, the fact that the utility problem has been largely unacknowledged until recently illustrates the importance of carrying out such studies, in spite of such difficulties. Careful empirical research can offer valuable insight into complex problems, complementing our theoretical analyses.

Acknowledgements

I would like to thank the members of the PRODIGY group, especially Craig Knoblock, Yolanda Gil, and Jaime Carbonell, for their help in carrying out the experiments reported here. Thanks also to Pat Langley and Jack Mostow for suggesting these experiments, and to Bernadette Kowalski for her help revising this paper.

References

1. Brown, F.M. "An Experimental Logic Based on the Fundamental Deduction Principal". *Artificial Intelligence 30*, 2 (1986).

2. DeJong,G.F. and Mooney, R. "Explanation-Based Learning: An Alternative View". *Machine Learning 1*, 2 (1986).

3. Fikes, R., Hart, P. and Nilsson, N. "Learning and executing generalized robot plans". *Artificial Intelligence* 3, 4 (1972).

4. Garey, M.R. and Johnson, D.S.. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

5. Kahn, K. "Partial Evaluation as an Example of the Relationship between Programming Methodology and AI". *AI Magazine 5*, 1 (1984).

6. Keller, R. M. Defining Operationality for Explanation-Based Learning. Proceedings of AAAI-87, Seattle, Washington, 1987.

7. Minton, S. Selectively Generalizing Plans for Problem Solving. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, CA, 1985.

8. Minton, S. and Carbonell, J.G. Strategies for Learning Search Control Rules: An Explanation-Based Approach. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, ITALY, 1987.

9. Minton, S.. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Kluwer Academic Publishers, 1988. Also appears as tech. report CMU-CS-88-133, Computer Science Dept., Carnegie-Mellon Univ.

10. Mitchell, T., Utgoff, P. and Banerji, R. Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics. In *Machine Learning: An Artificial Intelligence Approach*, Carbonell, J., Michalski, R. and Mitchell, T., Eds., Tioga Publishing Co., 1983.

11. Mitchell, T., Keller, R. and Kedar-Cabelli, S. "Explanation-Based Generalization: A Unifying View". *Machine Learning 1*, 1 (1986).

12. Prieditis, A.E. and Mostow, J. PROLEARN: Toward a Prolog Interpreter that Learns. Proceedings of AAAI-87, Seattle, WA., 1987.

13. Rosenbloom, P.S. and Laird, J.E. Mapping explanation-based generalization onto SOAR. Proceedings of AAAI-86, Philadelphia, PA, 1986.

14. Silver, B. Precondition Analysis. In *Machine Learning: An Artificial Intelligence Approah, Vol II*, Michalski, R.S., Carbonell, J.G. and Mitchell, T.M., Eds., Morgan Kaufmann, 1986.

15. Tambe, M. and Newell, A. Some Chunks are Expensive. Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, MI, 1988.