Why PRODIGY/EBL Works

Oren Etzioni*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
etzi@cs.cmu.edu

Abstract

Explanation-Based Learning (EBL) fails to accelerate problem solving in some problem spaces. How do these problem spaces differ from the ones in Minton's experiments [1988b]? Can minute modifications to problem space encoding drastically alter EBL's performance? Will PRODIGY/EBL's success scale to real-world domains? This paper presents a formal theory of problem space structure that answers these questions. The central observation is that PRODIGY/EBL relies on finding nonrecursive explanations of PRODIGY's problem-solving behavior. The theory explains and predicts PRODIGY/EBL's performance in a wide range of problem spaces. The theory also predicts that a static program transformer, called STATIC, can match PRODIGY/EBL's performance in some cases. The paper reports on an array of experiments that confirms this prediction. STATIC matches PRODIGY/EBL's performance in each of Minton's problem spaces.

Motivation

"It is rare that one sees an AI system evaluated carefully by anyone other than its creator."
[Bobrow, 1984]

Minton [1988b] showed that Explanation-Based Learning (EBL) can significantly accelerate problem solving. PRODIGY/EBL accelerated the PRODIGY problem solver in the Blocksworld, the Stripsworld (a STRIPS-like problem space), and the Schedworld (a scheduling problem space). EBL fails to reduce PRODIGY's problem-solving time in certain problem spaces, however.

This observation is particularly tantalizing because minute modifications to problem space encoding can drastically alter EBL's efficacy. For example, adding a single, carefully chosen, macro-operator to PRODIGY's Blocksworld operator set suffices to foil PRODIGY/EBL. The macro-operator allows PRODIGY to grasp a block that is second-from-the-top of a tower. Running PRODIGY/EBL on this augmented Blocksworld (following the training procedure outlined in Minton's thesis [1988a]) produced a rule set that actually slowed PRODIGY down on Minton's test problems. A discussion of this experiment appears in [Etzioni, 1990].

Why is PRODIGY/EBL foiled in the augmented Blocksworld? This paper reports on a formal theory of problem space structure that answers this question. The failure is not due to an idiosyncrasy of PRODIGY/EBL. As explained later, the augmented Blocksworld presents a rigorous challenge to any EBL system that shares PRODIGY's problem-solving method.

Note that controlling search is easy in the augmented Blocksworld. Merely ignoring the added macro-operator yields Minton's original Blocksworld. Indeed, the rules learned by PRODIGY/EBL in Minton's Blocksworld lead to adequate performance even in the augmented Blocksworld. The added macro-operator makes *learning* to control search difficult.

Overview

Learning problem solvers such as Soar [Laird *et al.*, 1987] and prodigy [Minton *et al.*, 1989] attempt to reduce problem-solving time by acquiring control rules. To choose between alternative actions, the problem solver matches the rules against its current state and adheres to the preferences expressed by the rules that matched successfully. The preferences can prune portions of the search space by rejecting alternatives, or by suggesting promising alternatives to try first.

Utilizing control rules trades problem space search for matching. I analyze a class of problem spaces for which this tradeoff is favorable: problem spaces that yield nonrecursive explanations of problem-solving behavior. Although I focus on PRODIGY/EBL, my conclusions apply to a wide range of EBL systems. Studying prodigy has enabled me to empirically test my analysis.

*The author is supported by an AT&T Bell Labs Ph.D. Scholarship. This research is sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499.

Since the cost of matching a control rule grows exponentially with the size of that rule, acquiring control rules can fail to accelerate problem solving. This problem has been dubbed the *utility problem* [Minton, 1988b]. The match cost of a bounded-size control rule is polynomial in the problem size, however. Thus, the utility problem could be solved by bounding the size of the control rules acquired.

Unfortunately, the size of EBL's rules is not bounded. Why is that? When EBL's explanations are recursive, distinct rules are learned at every recursive depth to which the explanations are expanded. This is known as the *generalization-to-N problem* [Cheng and Carbonell, 1986, Prieditis, 1986, Shavlik and DeJong, 1985]. In the Blocksworld, for example, PRODIGY/EBL learns distinct rules for clearing the bottom block of a two-block tower, a three-block tower, and so on. This is problematic in two regards. First, after learning from examples of two, three, and four block towers, PRODIGY/EBL will fail to generalize to five-block towers; in fact, any finite rule set will fail to cover all possible Blocksworld problems. Second, the match cost of the rules formed in this manner grows exponentially with the problem size.

Recursion abounds in problem spaces, even ones as simple as the Blocksworld. How is PRODIGY/EBL able to acquire effective control knowledge then? The answer is: PRODIGY/EBL *relies on finding nonrecursive explanations of* PRODIGY*'s problem solving behavior.* Nonrecursive explanations provide a natural, problem-space-specific bound on rule size. The cost of matching rules based on nonrecursive explanations is polynomial in the size of the problems encountered.

PRODIGY/EBL's ability to learn from failure allows it to find nonrecursive explanations even when PRODIGY's solutions are recursive. When PRODIGY's failure paths are recursive, PRODIGY/EBL can sometimes learn from success. When both success and failure can only be explained recursively, however, PRODIGY/EBL is foiled.

PRODIGY/EBL's explanations are not arbitrary proofs. The explanations mirror the structure of PRODIGY's problem spaces. I formalize this notion by defining the problem space graph (PSG), a compact representation of problem space structure. PRODIGY/EBL's explanations correspond to PSG subgraphs. PRODIGY/EBL finds nonrecursive explanations only when the appropriate nonrecursive subgraphs are present in the PSG. The absence of such subgraphs from the augmented Blocksworld foils PRODIGY/EBL.

A program that does not utilize training examples can form roughly the same rules as PRODIGY/EBL. The program, called STATIC, detects the PSG subgraphs that correspond to nonrecursive explanations, and extracts control rules directly from the subgraphs. STATIC matches PRODIGY/EBL's performance in Minton's problem spaces. This surprising result confirms my central claim: nonrecursive problem space structure is the primary source of PRODIGY/EBL's power.

The following section shows how recursive explanations lead to rules whose size cannot be bounded. The subsequent section defines the PSG, and formalizes the notion of problem space structure. I then demonstrate how EBL's performance can be analyzed in terms of the PSG. I describe STATIC briefly, and evaluate its performance experimentally. A discussion of generalization-to-N and other related work follows.

Recursive Explanations

This section shows that the size of PRODIGY/EBL's rules cannot be bounded due to recursive explanations. PRODIGY/EBL explains how choices made by the PRODIGY problem solver led to success, failure, or goal interaction. The explanations translate portions of PRODIGY's problem-solving trace into logical proofs. The proofs contain domain-specific axioms that correspond to PRODIGY's operators. When a trace contains recursive operator application, the corresponding proof is thus recursive. The depth of the recursion in the proof equals the depth of operator recursion in the trace. Distinct control rules are learned at each recursive depth. Moreover, larger rules are learned as the recursive depth increases.

In the Blocksworld, for example, clearing a block is a recursive operation. Clearing the bottom block of an N-block tower requires N-1 calls to the UNSTACK operator. As a result, PRODIGY/EBL forms distinct rules for the two-block tower, the three-block tower, and so on. Each rule only applies to towers of a given height. As the state size increases, and increasingly taller towers are possible, more rules are necessary for guiding PRODIGY to choose UNSTACK. Recursive explanations are not unique to learning from success. Failure and goal interaction can require recursive explanations as well.

As PRODIGY encounters problems which require deeper recursions, EBL will form successively larger rules. When the size of the problems is not bounded, the size of the rules learned from recursive explanations cannot be bounded either. In the worst case, the size of the learned rules scales linearly with the size of PRODIGY's problems. The size of rules learned from nonrecursive explanations, in contrast, is bounded by the number of predicates and operators in the problem space. Of course, the size of nonrecursive explanations can scale up as we consider increasingly larger problem spaces. The size is bounded, however, as PRODIGY encounters larger and larger problems in a fixed problem space.

Problem Space Graphs

PRODIGY/EBL's explanations mirror the structure of PRODIGY's problem space. Successfully clearing a block has a recursive explanation *because* plans that achieve the `clear` goal are recursive. Thus, whether

the size of PRODIGY/EBL's rules is bounded depends on where recursion occurs in the problem space. I formalize this notion using PSGs. A PSG is an AND/OR graph which represents the goal/subgoal relationships in a problem space. The PSG is independent of any state information; it is derived from the problem space definition. EBL's explanations correspond to PSG subgraphs. Since PSGs make recursion explicit, they facilitate finding nonrecursive explanations. This section introduces PSGs, and describes their relation to EBL's explanations.

A PSG consists of disjoint subgraphs each of which is rooted in a distinct goal literal. Each subgraph is derived by symbolically back chaining on the problem space's operators from the root. The root literal is connected via OR-links to all the operators that achieve the literal, and each operator is connected via AND-links to its preconditions. Thus, the PSG nodes are an alternating sequence of (sub)goals and operators; the PSG edges are either AND-links or OR-links. Figure 1 depicts the Blocksworld PSG subgraph rooted in `(holding block)`. The graph is directed and acyclic. Two operators that share a precondition have AND-links to the same node, so the graph is not a tree.

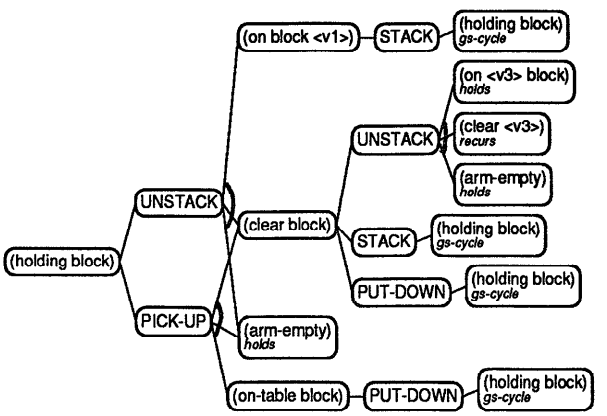

Figure 1: The `holding` PSG Subgraph.

An operator achieves a goal if the operator's effects successfully unify with the goal. This unification imposes codesignation constraints between the arguments to the goal literal, and the operator's variables. The operator's preconditions are partially instantiated to reflect these constraints. For example, since the goal is `(holding block)`, UNSTACK's first precondition is `(on block <v1>)`.

The graph terminates in one of four cases: A leaf literal (labeled *gs-cycle*) already appears on a path from the leaf to the root. This is known as a goal-stack cycle. A literal's predicate (labeled *recurs*) recurs on a path to the root with no goal-stack cycle. No operators unify with a literal (this never happens in the Blocksworld). Finally, a literal (labeled *holds*) is known to be true given the current goal stack. This fact can be inferred from constraints on legal states, and the observation that literals on the goal stack are not true in the current state. For example, PRODIGY is either holding a block or its arm is empty. Hence, if the current goal is `(holding block)`, then `arm-empty` must be true.[1]

The PSG and EBL's Proofs

The notion of representing programs (or problem spaces) as graphs is well-known in computer science (see, for example, [Kowalski, 1974]). It is surprising to note, however, that PRODIGY/EBL's proofs correspond to PSG subgraphs.

PRODIGY/EBL's failure proofs, for example, have the following flavor: an operator cannot be executed because one of its preconditions cannot be achieved. A precondition cannot be achieved because all of the operators that could potentially achieve it cannot be executed and so on. Each proof "explains" a failure by an alternating series of existentially and universally quantified statements about the relevant preconditions and operators. Success proofs have the same nature, but the quantifiers are reversed. These alternating sequences correspond to the alternating sequences of operator and precondition nodes in the PSG.

For example, UNSTACK fails because one of its preconditions, `(on block <v1>)`, cannot be achieved in the context of `(holding block)`. `on` cannot be achieved because all of the relevant operators (STACK is the only one) cannot be executed. STACK cannot be executed because one of its preconditions, `(holding block)`, results in a goal-stack cycle. This proof mirrors the subgraph in Figure 2. The example illustrates the correspondence between PSG subgraphs and PRODIGY/EBL's explanations. A formal exposition of this correspondence appears in [Etzioni, 1990].

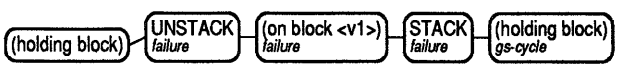

Figure 2: The Failure Subgraph for UNSTACK.

Recursion in the PSG occurs at the nodes labeled *recurs*. An explanation is recursive only if one of the nodes in its corresponding PSG subgraph is labeled *recurs*. Thus, the PSG provides an easy means of avoiding recursive proofs.

The only source of recursion in the `holding` PSG is the application of UNSTACK to achieve `clear`. Proving that an operator can achieve `(holding block)` requires demonstrating that `block` can be cleared—a recursive proof. Hence, in the Blocksworld, learning from

[1] This statement and the PSG's depiction are slightly simplified. Since the `holding` goal is satisfied only when a particular block is held, `(arm-empty)` will be false when PRODIGY is grasping a different block. Expanding the `(arm-empty)` node one step further, however, reaches the precondition `(holding <v2>)` which *is* guaranteed to be true in this context.

success results in recursive explanations and overly-specific rules.

Proving that an operator will *fail* to achieve `holding`, however, does not require mentioning `clear` at all. PICK-UP fails if `(on-table block)` is not true; UNSTACK fails if `(on block <v1>)` is not true. As the PSG indicates, a goal-stack cycle results in either case. This occurs in Blocksworld problems of any size. Thus, EBL is able to form general rules by analyzing failures. In fact, five failure proofs yield all the control rules necessary to make appropriate operator and bindings choices. The failure proofs are nonrecursive, yielding rules that are compact, general, and cheap-to-match.

The Blocksworld example illustrates an important point. The nature of PRODIGY/EBL's rules depends both on the problem space and on the target concept employed to acquire the rules. Proofs based on distinct target concepts pick out distinct PSG subgraphs. In the example, the success subgraphs are recursive whereas the failure subgraphs are not. This observation suggests a criterion for choosing which target concept to utilize in learning control knowledge: *learn from the target concept that yields a nonrecursive proof.* STATIC, described in the following section, employs this criterion to choose which control rules to form.

PRODIGY/EBL does not employ this criterion. In the case of `holding`, for example, it learns from both success and failure. PRODIGY/EBL relies on its utility analysis mechanism to weed out ineffective rules.

Predicting EBL's Performance

In the Blocksworld, nonrecursive explanations can be found for most problem-solving phenomena. When explaining the success of an operator is recursive, explaining the failure of its sibling operators is not. Consequently, given a sufficient number of examples, PRODIGY/EBL is able to accelerate PRODIGY significantly on Blocksworld problems.

In the augmented Blocksworld, by way of contrast, recursion occurs both on the route to failure and to success. Consider, for example, choosing UNSTACK to achieve the goal `(holding block)` when the block is on the table. This path is doomed to failure in both Blocksworlds. In the Blocksworld, the failure can be explained nonrecursively since trying to achieve `on` leads to a goal-stack cycle immediately (Figure 2).

In the augmented Blocksworld, in contrast, `on` can be achieved by the macro-operator added to PRODIGY's operator set. The macro-operator allows PRODIGY to grasp the block that's second-from-the-top of a tower. The top block lands on the block that is third-from-the-top as a side effect. Thus, PRODIGY has another means of achieving `on` which does not immediately cause a goal-stack cycle. Since prodigy explores the recursive path that begins with the macro-operator before failing, PRODIGY/EBL is forced to analyze that path in order to explain PRODIGY's failure. Consequently, explaining the failure of UNSTACK in the augmented Blocksworld is recursive.

This example illustrates how adding a macro-operator to the Blocksworld led to PRODIGY/EBL's lackluster performance. Problem spaces that yield only recursive explanations, relative to the learner's target concept and the problem solver's search method, will challenge PRODIGY/EBL, Soar, and a wide range of EBL systems.

Whether a problem space yields recursive or nonrecursive explanations can be determined by analyzing the problem space's PSG. EBL will accelerate problem solving when the PSG yields nonrecursive explanations.[2] Hence, analyzing PSGs is a powerful methodology for predicting and explaining EBL's performance. I return to this idea in the conclusion of the paper.

Static Analysis of PSGs

Thus far I have argued that PRODIGY/EBL exploits nonrecursive proofs to extract cheap but effective control rules, and that the existence of these proofs is a function of both problem space structure and PRODIGY/EBL's "proof strategies" or target concepts. Furthermore, I have shown that nonrecursive proofs correspond to nonrecursive PSG subgraphs.

It follows that we can extract effective control rules directly from the PSG. To test this prediction I wrote the STATIC program which forms control rules by applying a select subset of PRODIGY/EBL's proof strategies directly to the PSG. Given a problem space encoded as a set of operators and a set of constraints on legal states, STATIC constructs the PSG corresponding to that problem space. The PSG subgraph in Figure 1 depicts an instance of STATIC's PSG data structure. STATIC embarks on a depth-first search for nonrecursive PSG subgraphs, and extracts control rules from the nonrecursive subgraphs it finds. STATIC only forms control rules when the nonrecursive learning criterion, articulated earlier, is met. No examples, rule compression, or utility analysis are used. A sketch of STATIC's algorithm for analyzing failure appears in Table 1. STATIC's algorithm for analyzing success is analogous.

To illustrate the algorithm's execution consider the failure subgraph in Figure 2. STATIC computes the **fc**, or failure condition, and **label** fields of each node in postorder. STACK, at the right of the subgraph, is labeled *failure*, and its **fc** is ogs(`(holding block)`). That is, STACK(block,<v1>) will fail when `(holding block)` is on PRODIGY's goal stack. The **fc** of `(on block <v1>)` is the same. UNSTACK's only failed precondition is `(on block <v1>)`. UNSTACK is labeled *failure* and its **fc** is ogs(`(holding block)`) $\wedge\neg$(on

[2] Of course, in cases where different bounds on the cost of matching EBL's rules can be enforced, EBL will accelerate problem solving as well.

Input: operator set + constraints on legal states.
Output: Operator rejection rules.

;;; ops(lit) refers to the operators that achieve lit.
;;; fc(x) refers to the conditions under which x fails.
Main loop:
for lit in goal-lits **do**
    Create PSG subgraph rooted in lit (e.g., Figure 1).
    for o in ops(lit) **do** LABEL-OP(o).
    Find operators labeled *failure* and form rejection
        rules whose antecedents are the operators' fcs.

;;;failure-labels: {failure, gs-cycle, unachievable}.
;;;Additional labels: {success, holds, recurs}.
;;; $p \in$ fail-precs(o) iff $p \in precs(o) \wedge label(p) \in$ failure-labels.
;;; $ogs(p)$ is true when p is on prodigy's goal-stack.
procedure LABEL-OP(o)
for unlabeled lit in precs(o) **do** LABEL-LIT(lit).
if ¬empty(fail-precs(o)) **then** {label(o) ← *failure*;
  if $\exists p \in$ fail-precs(o) s.t. label(p)=*gs-cycle* **then** fc(o) ← ogs(p).
  else for p in fail-precs(o) **do** fc(o) ← fc(o) $\vee(fc(p) \wedge \neg p)$.}
else if $\exists p \in precs(o)$ s.t. label(p)=*recurs* **then** label(o) ← *recurs*.
    else label(o) ← *success*.

;;;LABEL-LIT(lit) is never called when ops(lit)=NIL.
;;;$o \in$ fail-ops(lit) iff $o \in ops(lit) \wedge label(o) = failure$.
procedure LABEL-LIT(lit)
for o in ops(lit) **do** LABEL-OP(o).
if $\exists o \in ops(lit)$ s.t. label(o)=*success* **then** label(lit) ← *success*.
else if $\exists o \in ops(lit)$ s.t. label(o)=*recurs* **then** label(lit) ← *recurs*.
    else {label(lit) ← *failure*;
        for o in fail-ops(lit) **do** fc(lit) ← fc(lit) ∧ fc(o).}

Table 1: STATIC's algorithm for analyzing failure.[3]

`block <v1>`). Thus, STATIC forms a rule that rejects UNSTACK when PRODIGY's goal is `(holding block)` and `(on block <v1>)` is false. An analogous rule rejects PICK-UP when `(on-table block)` is false. Since both UNSTACK and PICK-UP require achieving `(clear block)`, a precondition that is achieved recursively, no rules are learned from success.

To acquire goal-ordering rules STATIC considers all possible goal pairs. For each pair, STATIC checks whether achieving one goal necessarily clobbers the other goal. If so, STATIC tells PRODIGY to achieve the clobbering goal first. The clobbering can be avoided in this manner. For example, achieving `(on block1 block2)` clobbers `(holding block3)` because `(holding block1)` is a necessary precondition of `(on block1 block2)` and only one block can be held at any given time. Consequently, STATIC informs PRODIGY that on should be achieved before **holding**. Schoppers [Schoppers, 1989] reports on a related technique for detecting goal conflicts. See [Etzioni, 1990] for a comparison, and a complete discussion of STATIC.

[3]The $\vee, \wedge$ operators are not applied to their arguments. Thus, logical formulae are incrementally constructed by the above procedures. Treatment of failures due to inconsistent variable bindings is omitted for brevity.

STATIC matches PRODIGY/EBL's performance in Minton's problem spaces as shown in Figure 3. The figure depicts cumulative running times for PRODIGY given STATIC's rules, PRODIGY/EBL's rules, and rules written by human experts. EBL's rules, the human rules, the randomly generated problem sets, and the cumulative graphs format are taken from Minton's experiments. The time to solve each problem was limited to 150 CPU seconds. Several Schedworld problems could not be solved within this time limit using one or more of the rule sets. All such problems were excluded from the Schedworld graphs. STATIC outperformed PRODIGY/EBL slightly in all three problem spaces. However, all the rule sets accelerated PRODIGY significantly.

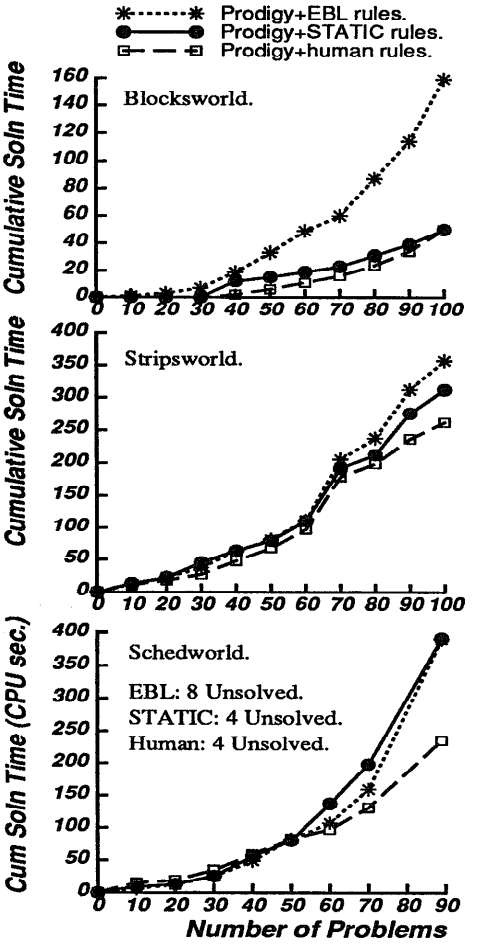

Figure 3: STATIC's Performance in the Blocksworld, the Stripsworld, and the Schedworld.

Since STATIC is biased to only form rules based on nonrecursive PSG subgraphs, it does *not* necessarily form a superset of PRODIGY/EBL's rules. On Minton's problem spaces, for example, STATIC acquired only

one rule based on analyzing successful paths whereas PRODIGY/EBL formed thirteen. STATIC does rely on "success rules" in other problem spaces, however. PRODIGY/EBL's rule sets contained a variety of overly-specific rules that STATIC did not form. The overly-specific rules were formed when PRODIGY/EBL retained features of its training examples, such as recursion depth, unnecessarily. Finally, STATIC formed useful rules that PRODIGY/EBL missed, when PRODIGY/EBL did not encounter the requisite examples in its training sequences.

Prieditis [1988] and van Harmelen & Bundy [1988] have pointed to the affinity between partial evaluation and EBL. Indeed, STATIC constructs the PSG by partially evaluating PRODIGY's problem space definition. However, the control rules acquired modify the order in which PRODIGY considers operators and subgoals. These reordering transformations are not part of the usual arsenal of partial evaluators. Thus, STATIC and EBL's impact on PRODIGY is more appropriately viewed as program transformation.

Caveats

STATIC will not match PRODIGY/EBL's performance in every case. PRODIGY/EBL can exploit the distribution of problems encountered by PRODIGY whereas STATIC cannot. STATIC is particularly sensitive to the constraints on legal states that define the PSG. If some of the constraints are omitted, STATIC will analyze various impossible paths and form a host of unnecessary rules. Finally, STATIC only utilizes a select subset of PRODIGY/EBL's proof strategies. For example, STATIC only searches for pairwise goal interactions, and will overlook more complex ones. Although learning from complex goal interactions can be useful, analyzing such interactions often leads PRODIGY/EBL to form highly specific rules that are expensive to match. STATIC's scope is restricted (relative to PRODIGY/EBL's) for two reasons: to avoid potentially expensive rules, and to keep the static analysis tractable.

STATIC is not meant as an alternative to EBL. A hybrid algorithm consisting of some static preprocessing followed by EBL may well prove most effective. STATIC appears in this paper merely to empirically demonstrate the utility and importance of nonrecursive PSG subgraphs for acquiring control knowledge.

Related Work

Formal analyses of EBL have been reported by [Cohen, 1989, Greiner and Likuski, 1989, Mahadevan *et al.*, 1988]. None of these analyses point to nonrecursive explanations as EBL's primary source of power.

Subermanian & Feldman [1990] show that for horn-clause theories in which recursion is structural and the cost of choosing which rule to apply is bounded, learning from recursive explanations is unlikely to reduce theorem-proving time. They do not consider learning from failure and goal interaction, representing problem spaces as graphs, or statically deriving nonrecursive explanations. Although a simple complexity analysis suffices to argue against learning from recursive explanations via EBL (the overhead of utilizing rules learned from such explanations is exponential in the depth of the recursions encountered), Subermanian & Feldman develop an elaborate cost model of horn-clause theorem proving. This detailed model is motivated by Subermanian & Feldman's broader project which seeks to "quantitatively estimate" the cost of inference.

Remarkably, despite our focus on different EBL algorithms and different problem spaces, we reached similar conclusions. This provides further confirmation of the ideas in both papers.

Research on generalization-to-N (e.g., [Cohen, 1988, Shavlik, 1990, Shell and Carbonell, 1989]) has also identified recursion as a problem for EBL. The focus of that body of work, however, is on forming general rules based on recursive explanations. Shavlik's program, for example, learns horn-clauses that retain recursive calls in their antecedents. Thus, a particular recursive depth is not captured by the antecedent. Unfortunately, the cost of utilizing such rules can increase exponentially with the depth to which the recursive calls are expanded at run-time.

Letovsky's RECEBG algorithm [Letovsky, 1990] is able to expand some recursive calls at compile-time without forming overly-specific rules. RECEBG expands recursive calls when it can prove, using structural induction, that the recursion will terminate.

Conclusion

Utilizing control rules trades problem space search for matching. I have analyzed a class of recursive problem spaces for which this tradeoff is favorable: problem spaces that yield nonrecursive explanations of problem-solving behavior. Such explanations yield bounded-size control rules. The match cost of bounded-size control rules is polynomial in the size of the problem solver's input.

I formalized the notion of nonrecursive explanations in terms of the problem space graph (PSG), a compact representation of problem space structure. PRODIGY/EBL's nonrecursive explanations correspond to nonrecursive PSG subgraphs. EBL's performance can be analyzed using the PSG.

I demonstrated the practical import of this analysis via two experiments. First, I showed that PRODIGY/EBL's performance degrades in the augmented Blocksworld, a problem space robbed of its nonrecursive PSG subgraphs. Second, I showed that a program that extracts nonrecursive explanations directly from the PSG matches PRODIGY/EBL's performance on Minton's problem spaces. Both experiments lend credence to the claim that PRODIGY/EBL's primary source of power is nonrecursive problem space structure.

The theory developed in this paper addresses the

questions posed at the outset. PRODIGY/EBL's efficacy in "real-world domains" depends on the structure of their PSGs. Its success will scale to considerably larger and more complex domains whose encodings yield the appropriate nonrecursive PSG subgraphs. Minor modifications to the problem space's representation can result in major changes to its PSG, however. Consequently, PRODIGY/EBL's performance can vary as seemingly minor changes are made to the problem space representation. Analyzing problem spaces in terms of PSG structure enables us to understand this behavior.

The problem studied in this paper transcends PRODIGY and EBL. Currently, most learning programs are halted after relatively short running times (compared with a human's life-time of learning) lest they swamp their performance systems with massive amounts of knowledge. Any system that accumulates knowledge over time must come to terms with the cost of utilizing that knowledge. Thus, it is not sufficient to have a powerful learning algorithm. The algorithm must be appropriately embedded in a performance system. This raises a host of challenging technical questions that research on memory organization, selective learning, and bounded match cost has to address. My work is an instance of such research.

Acknowledgments

I would like to thank William Cohen, Scott Dietzen, Haym Hirsh, Daniel Kahn, Jack Mostow, David Steier, Prasad Tadepalli, and Raúl Valdés-Pérez for comments on previous drafts. The following made invaluable contributions to the ideas herein: Craig Knoblock, Stan Letovsky, Tom Mitchell, Steve Minton, and Paul Rosenbloom. Special thanks go to Steve Minton—whose thesis work made studying PRODIGY/EBL possible.

References

[Bobrow, 1984] Daniel G. Bobrow. Retrospectives: A note from the editor. *Artificial Intelligence*, 23, 1984.

[Cheng and Carbonell, 1986] P. Cheng and J. G. Carbonell. The FERMI system: Inducing iterative macro-operators from experience. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 490–495, 1986.

[Cohen, 1988] William W. Cohen. Generalizing number and learning from multiple examples in explanation based learning. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 256–269, 1988.

[Cohen, 1989] William W. Cohen. Solution path caching mechanisms which proveably improve performance. Technical Report DCS-TR-254, Rutgers University, 1989.

[Etzioni, 1990] Oren Etzioni. *A Structural Theory of Search Control.* PhD thesis, Carnegie Mellon University, 1990. In preparation.

[Greiner and Likuski, 1989] Russell Greiner and Joseph Likuski. Incorporating redundant learned rules: A preliminary formal analysis of EBL. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[Kowalski, 1974] Robert A. Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, 22:572–595, 1974.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Letovsky, 1990] Stanley Letovsky. Operationality criteria for recursive predicates. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[Mahadevan *et al.*, 1988] Sridhar Mahadevan, B. K. Natarajan, and Prasad Tadepalli. A framework for learning as improving problem-solving performance. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*, 1988.

[Minton *et al.*, 1989] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989. Available as techincal report CMU-CS-89-103.

[Minton, 1988a] Steven Minton. *Learning Effective Search Control Knolwedge: An Explanation-Based Approach.* PhD thesis, Carnegie Mellon University, 1988. Available as technical report CMU-CS-88-133.

[Minton, 1988b] Steven Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence.* Morgan Kaufmann, 1988.

[Prieditis, 1986] A. E. Prieditis. Discovery of algorithms from weak methods. In *Proceedings of the international meeting on advances in learning*, pages 37–52, 1986.

[Prieditis, 1988] A. E. Prieditis. Environment-guided program transformation. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*, 1988.

[Schoppers, 1989] Marcel Joachim Schoppers. *Representation and Automatic Synthesis of Reaction Plans.* PhD thesis, University of Illinois at Urbana-Champaign, 1989. Available as technical report UIUCDCS-R-89-1546.

[Shavlik and DeJong, 1985] Jude W. Shavlik and G. F. DeJong. Building a computer model of classical mechanics. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pages 351–355, 1985.

[Shavlik, 1990] Jude W. Shavlik. Acquiring recursive concepts and iterative concepts with explanation-based learning. *Machine Learning*, 5(1), 1990. To appear.

[Shell and Carbonell, 1989] Peter Shell and Jaime G. Carbonell. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989. .

[Subermanian and Feldman, 1990] Devika Subermanian and Ronen Feldman. The utility of ebl in recursive domain theories. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[van Harmelen and Bundy, 1988] Frank van Harmelen and Alan Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36, 1988. Research Note.