To appear in *artificial intelligence*

A Structural Theory of Explanation-Based Learning

Oren Etzioni

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
etzioni@cs.washington.edu

May 1992

Abstract

The impact of Explanation-Based Learning (EBL) on problem-solving efficiency varies greatly from one problem space to another. In fact, seemingly minute modifications to problem space encoding can drastically alter EBL's impact. For example, while PRODIGY/EBL (a state-of-the-art EBL system) significantly speeds up the PRODIGY problem solver in the Blocksworld, PRODIGY/EBL actually slows PRODIGY down in a representational variant of the Blocksworld constructed by adding a single, carefully chosen, macro-operator to the Blocksworld operator set. Although EBL has been tested experimentally, no theory has been put forth that accounts for such phenomena. This paper presents such a theory.

The theory exhibits a correspondence between a graph representation of problem spaces and the proofs used by EBL systems to generate search-control knowledge. The theory relies on this correspondence to account for the variations in EBL's impact. This account is validated by STATIC, a program that extracts EBL-style control knowledge directly from the graph representation, without using training examples. When tested on PRODIGY/EBL's benchmark tasks, STATIC was up to three times as effective as PRODIGY/EBL in speeding up PRODIGY.

1 Introduction

Controlling search is a central concern for AI. Explanation-Based Learning (EBL) is a widely-used technique for acquiring search-control knowledge [34]. The credibility of EBL was bolstered by Minton's experiments [30]. Minton tested EBL's impact on the PRODIGY problem solver in three benchmark problem spaces. In each case, PRODIGY's explanation-based learning module, PRODIGY/EBL, was able to significantly speed up PRODIGY by acquiring search-control rules.[1] To demonstrate the power of EBL, Minton ran PRODIGY, with and without PRODIGY/EBL's control rules, on one hundred randomly generated problems in each problem space. PRODIGY ran considerably faster when guided by PRODIGY/EBL's control rules. Figure 1 shows the results of applying PRODIGY/EBL to the Blocksworld.

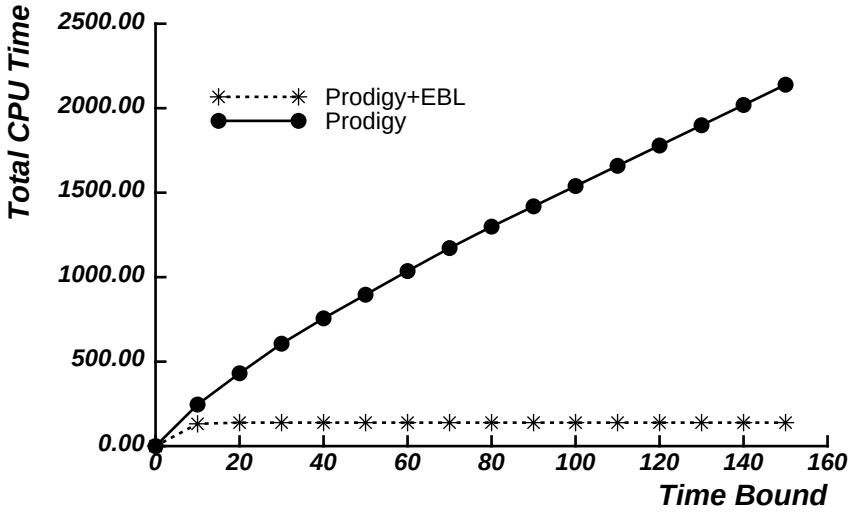

Figure 1: PRODIGY/EBL speeds up PRODIGY in the Blocksworld. A bound is placed on the CPU-time allotted to each problem. The total time to solve all the problems (Y-axis) is graphed against the time bound (X-axis). Thus, the Y-coordinate of a point on a curve represents the amount of time a system spent to solve all the problems, given the time bound in the X-coordinate. The graph is designed to show how the relative performance of the systems scales as the time bound is increased. See Section 6.2 for additional discussion of this graphical format.

1.1 Motivation

Unfortunately, EBL's impact on problem-solving efficiency varies widely from one problem space to another. In fact, seemingly minute modifications to problem space encoding can drastically alter EBL's impact on problem-solving time. The Augmented Blocksworld

[1] The paper uses the term "PRODIGY/EBL" to refer specifically to PRODIGY's EBL module, and "EBL" to refer to the general paradigm. The PRODIGY problem solver, guided by PRODIGY/EBL's control rules, is referred to as "PRODIGY+EBL" (see, for example, Figure 1).

problem space (ABworld for short) illustrates this point. The ABworld, created by adding a single, carefully chosen, macro-operator to PRODIGY's Blocksworld operator set, foiled PRODIGY/EBL. The macro-operator, displayed in Figure 2, enables PRODIGY to grasp a block that is second-from-the-top of a tower.

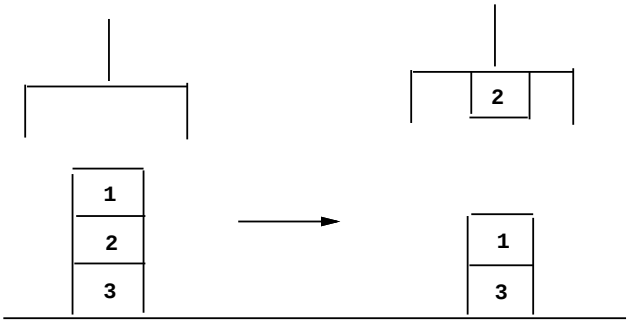

Figure 2: The GRASP-SECOND-BLOCK macro-operator used to create the ABworld.

Running PRODIGY/EBL on the ABworld (following Minton's training procedure for the Blocksworld) produced a rule set that actually slowed PRODIGY down on Minton's test problems (Figure 3). A detailed discussion of this experiment appears in Section 6.3.

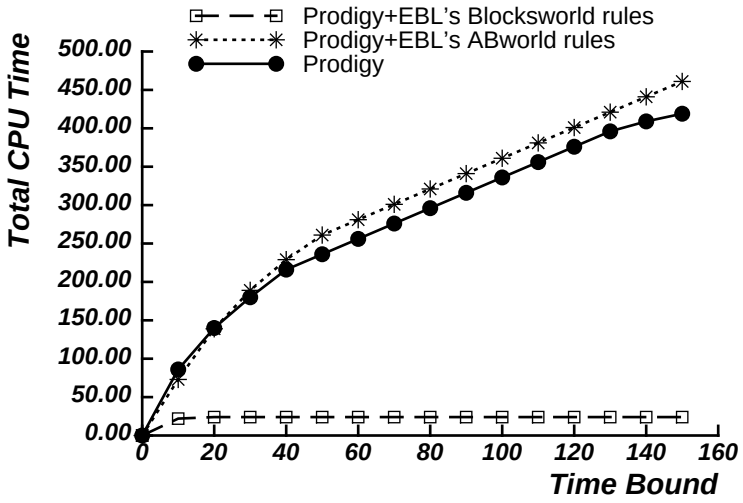

Figure 3: PRODIGY/EBL is foiled in the ABworld.

Note that controlling search is easy in the ABworld. 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 when they are used to guide PRODIGY in the ABworld (Figure 3). The added macro-operator makes *learning* to control search more difficult. A theory is needed to explain this surprising phenomenon.

Previous work (e.g., [15, 30, 38, 52]) has emphasized the role of the problem distribution encountered by the problem solver in accounting for EBL's performance. However, problem

distributions cannot explain why PRODIGY/EBL was foiled in the ABworld since *the same problems were used to train and test* PRODIGY/EBL *in both the Blocksworld and the ABworld.* The only change was the addition of a single macro-operator to the Blocksworld's operator set. Only a theory that analyzes the effectiveness of EBL in terms of problem-space characteristics can account for PRODIGY/EBL's performance in the ABworld experiment. This paper presents such a theory.

1.2 Overview

The theory relies on two basic constructs: the dichotomy between recursive and nonrecursive proofs and the Problem Space Graph (PSG) representation. The dichotomy is important because, as argued below, nonrecursive proofs tend to generate effective search-control knowledge whereas recursive proofs often fail to do so. The PSG is an AND/OR graph representing all backward-chaining paths through a problem space. PSGs facilitate answering questions such as "in what ways do different subgoals interact?" and "which subgoals give rise to recursive plans?" PSGs enable us to anticipate the proofs EBL can generate, given a particular problem space encoding.

These constructs enables us to state the fundamental tenet of the structural theory.
The Structural Thesis: *there is a correspondence between the PSG representation and the proofs used by EBL systems to generate search-control knowledge. As a result, automatic PSG analysis can be used to compute the weakest preconditions of the proofs and to discriminate between recursive and nonrecursive proofs.*

The thesis is validated by STATIC, a novel computational procedure that yields effective search-control knowledge based on PSG analysis. Given a problem-space definition, STATIC generates and traverses the appropriate PSGs, systematically searching for PSG subgraphs that correspond to nonrecursive EBL proofs. STATIC outputs the control knowledge EBL would have acquired from these proofs. STATIC's performance is compared with that of PRODIGY/EBL in Section 6.2; its design is described in detail in [11].

Asymptotic Analysis EBL's impact on problem solving is a complex phenomenon that depends on a large number of factors including the problem space definition, the problem distribution, the problem solver's search method, the problem solver's matcher, EBL's target concepts, EBL's training examples, and more. The analysis in this paper abstracts away from many of these intricate details, and attempts to identify key problem space characteristics that strongly influence EBL's performance.

The analysis contrasts recursive and nonrecursive proofs and shows that, as state size increases, the size of nonrecursive proof trees is bounded whereas the size of recursive proof trees is not. It follows that the cost of matching the weakest preconditions of recursive proofs increases exponentially with state size, but only polynomially in the case of nonrecursive proofs. The analysis also shows that, unless the depth of the problem solver's recursions is bounded, no finite set of recursive proofs can achieve a polynomial-node search in the limit. However, when learning from nonrecursive proofs, EBL can achieve polynomial-time

problem solving in certain cases (see Section 5.1.2).

Heuristics Based on the asymptotic analysis summarized above, I derive two simple heuristics for EBL systems:

- Learn from nonrecursive proofs.
- Do not learn from recursive proofs.

Since these heuristics do not take problem distribution into account, it is easy to manufacture cases in which they are inappropriate (see Section 5). Nevertheless, the experiments in Section 6 demonstrate that both heuristics are useful in practice. Note that by learning from target concepts such as failure, EBL can learn from nonrecursive proofs even in highly recursive problem spaces.

The heuristics are attractive because they constitute an *a priori* basis for generating effective control knowledge. A number of researchers have developed *post hoc* mechanisms for evaluating control knowledge by measuring its effectiveness on a sample of problems [19, 20, 32, 54]. However, as argued in Section 5, these *post hoc* mechanisms are heuristic as well. In addition, generating the large samples typically required by these mechanisms is very costly.

Experimental Validation When applied to PRODIGY, the structural theory helps explain PRODIGY/EBL's success in Minton's experiments as well as several experimental results reported in [10]:

1. STATIC outperforms PRODIGY/EBL when compared using PRODIGY/EBL's benchmark tasks.
2. PRODIGY/EBL's impact degrades significantly in the ABworld relative to the Blocksworld.
3. PRODIGY/EBL significantly reduces PRODIGY's Blocksworld problem-solving time over a wide range of problem distributions.
4. PRODIGY/EBL's impact degrades sharply when learning only from success.

Organization The paper is organized as follows. Sections 2 and 3 present a complexity analysis of EBL problem solvers. Section 4 introduces the structural thesis, and Section 5 argues for the heuristics mentioned above, concluding the presentation of the theory. Section 6 describes a host of experiments performed to test the theory, and Section 7 contrasts the theory with related work analyzing EBL. Finally, Section 8 presents a critique of the structural theory and points to directions for future work.

2 Meta-Level Problem Solvers

This section describes an abstract model of meta-level problem solvers that is the basis for the analysis that follows. The model is an idealization of problem solvers such as [7], MRS [18], Soar [25], PRODIGY [34], THEO [37], and many others. The distinguishing feature of meta-level problem solvers is their ability to use domain-specific meta-level rules (called *control rules*) to guide their problem solving. Each control rule consists of applicability conditions and a recommendation. At every node in its problem-solving search, a meta-level problem solver matches the applicability conditions of its rules against its current state. When the applicability conditions of a rule are met, the meta-level problem solver abides by its recommendation. The problem solver does not subgoal on the applicability conditions of control rules. The PRODIGY problem solver, described below, is an example of a meta-level problem solver.

2.1 The PRODIGY Problem Solver

Detailed descriptions of PRODIGY appear in [30, 34, 35]. The bare essentials follow. PRODIGY is a domain-independent problem solver. Given an initial state and a goal expression, PRODIGY searches for a sequence of operators that will transform the initial state into a state that matches the goal expression. A sample PRODIGY operator appears in Table 1. PRODIGY's sole problem-solving method is a form of means-ends analysis [40]. Like STRIPS [16], PRODIGY employs operator preconditions as its differences. However, PRODIGY's operator description language is considerably more expressive, allowing universal quantification and conditional effects.

```
(UNSTACK
    (preconditions
        (and (object Block-X) (object Block-Y)
             (on Block-X Block-Y) (clear Block-X) (arm-empty)))
    (effects ((del (on Block-X Block-Y))
              (del (clear Block-X))
              (del (arm-empty))
              (add (holding Block-X))
              (add (clear Block-Y)))))
```

Table 1: The Blocksworld operator UNSTACK. Variable names are capitalized.

PRODIGY's default search strategy is depth-first search. The search is carried out by repeating the following decision cycle [31]:

1. Choose a node in the search tree. A node consists of a set of goals and a world state.

2. Choose one of the goals at that node.

3. Choose an operator that can potentially achieve the goal.

4. Choose bindings for the variables in the operator. If the instantiated operator's preconditions match the state then apply the operator and update the state, otherwise subgoal on the operator's unmatched preconditions. In either case, a new node is created.

Search-control knowledge in PRODIGY is encoded via control rules, which override PRODIGY's default behavior by specifying that particular candidates (nodes, goals, operators, or bindings) should be selected, rejected, or preferred over other candidates [31]. Alternatives that are selected are the only ones tried; alternatives that are rejected are removed from the selected set. Finally, all other things being equal, preferred alternatives are tried before other ones. PRODIGY matches control rules against its current state. If the antecedent of a control rule matches, PRODIGY abides by the recommendation in the consequent. For example, the control rule in Table 2 tells PRODIGY to reject the Blocksworld operator UNSTACK when the block to be held is not on any other block.

```
(REJECT-UNSTACK
  (if (and (current-node Node)
           (current-goal Node (holding Block-X))
           (candidate-operator Node unstack)
           (known Node (not (on Block-X Block-Y)))))
  (then (reject operator unstack)))
```

Table 2: A control rule from PRODIGY's Blocksworld.

2.2 Formal Definition

I specify the meta-level problem solvers model more precisely below. See [10] for a complete specification. A problem space is defined by a set of operators that add and delete ground literals from states.[2] Given a problem-space definition, a problem solver takes as input an initial state and a ground goal expression, and searches for an operator sequence that will map the initial state to one that matches the goal. The size of the problem solver's state is denoted by s. In the absence of control knowledge, the number of nodes expanded during search is assumed to scale exponentially with s in the worst case.

Control rules have the potential to reduce the number of nodes expanded to a polynomial or even linear function of s. Since matching each control rule has a cost, however, control

[2] A *literal* is a possibly negated atomic formula. A *ground* literal is one that contains no variables.

rules typically reduce the number of nodes searched but increase the cost of expanding each node, measured by the total number of elementary matching operations [57]. Consequently, control rules do not necessarily reduce problem-solving time.

When does a set of control rules τ reduce problem-solving time on an individual problem? This question can be answered simply using the following notation:

- η: the number of nodes expanded during unguided problem solving.
- η_τ: the number of nodes expanded when problem solving is guided by τ.
- κ: the fixed cost of expanding a node without matching any control rules.
- μ_τ: the average cost, per node, of matching τ.

The cost of solving a problem without control rules is $\kappa\eta$. The cost of solving the problem using the rule set τ is $(\kappa + \mu_\tau)\eta_\tau$. Clearly, τ reduces problem-solving time if and only if:

$$(\kappa + \mu_\tau)\eta_\tau < \kappa\eta \tag{1}$$

2.3 The Utility Problem

The above inequality applies to individual problems. However, control knowledge is usually provided for an entire problem space. When is control knowledge effective over the entire space? The most widely-used notion of effectiveness in the literature is *average speedup*. A set of control rules is said to be effective if it speeds up problem solving, on average, on a population of problems (e.g. [21, 32, 52]). Average speedup is relatively easy to test experimentally; measuring problem-solving time with and without a set of control rules, on a large, randomly generated sample of problems, indicates whether the set achieves average speedup or not [13]. Unfortunately, average speedup is distribution-specific—a rule set may be effective on one problem distribution and ineffective on another. Furthermore, average speedup is a weak notion. Problem solving may remain intractable, due to its exponential nature, despite a sizable average speedup.

We can replace "average speedup" with "achieving polynomial-time problem solving" as the criterion for the effectiveness of a rule set. Polynomial-time problem solving is distribution-free, and achieving polynomial-time problem solving is generally taken to guarantee tractability ("polynomial" may be replaced with "low-order polynomial" if necessary). Moreover, any rule set that achieves polynomial-time problem solving will also achieve average speedup on sufficiently difficult problems because, for any pair of polynomial and exponential functions, there is some point after which the exponential function is always larger. Thus, the exponential cost of default problem solving is invariably greater than the cost of polynomial-time problem solving for sufficiently difficult problems.

Note that the terms "polynomial" and "exponential" in the above paragraph refer to the computational complexity of the problem solver's running time, not to the inherent complexity class of the problem (e.g., NP, PSPACE, etc.). The complexity class of any

problem is *fixed* given the problem definition and a particular computational model. However, the complexity of different *methods* or algorithms for solving the problem varies. The goal of speedup learning, as formulated above, is to automatically transform the problem solver's default search behavior into a polynomial-time algorithm for a given problem. Clearly, this can only be done for problems in the complexity class P (i.e., problems solvable in polynomial time). Still, automatically generating polynomial-time algorithms, for a wide range of problems in P, has distinct advantages over requiring a human programmer to derive the algorithms by hand.

Although the polynomial-time criterion is stronger and easier to analyze (cf. [39]), it is worst-case and asymptotic. Furthermore, average speedup may be attainable in cases where polynomial-time problem solving is not. Consequently, this paper will consider both criteria explicitly.

2.4 The Cost of Matching Control Rules

This section shows that the cost of matching a conjunctive logical expression against a problem solver's state, using standard match algorithms, is exponential in the expression's length [57].[3] This observation is invoked repeatedly in the analysis that follows.

Consider the tree generated by the matching process. A node in the tree represents an unbound variable in the expression being matched, and a branch in the tree represents a potential binding for the variable. A path from the root of the tree to a leaf represents a variable substitution under which the expression is matched (Figure 4).

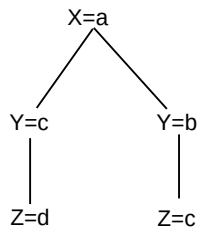

Figure 4: A match tree showing how the expression `p(X,Y),r(Y,Z)` might be matched against the state `p(a,c),p(a,b),r(c,d),r(b,c)` .

Thus, the tree's depth is the number of unbound variables, and the tree's branching factor is the number of potential bindings for each variable. When an expression containing v unbound variables is matched against a state of size s, the number of nodes in the tree (and hence the work done by the matcher) is $O(s^{v+1})$. Since the number of unbound variables in an expression is bounded by the expression's length, match time is said to be exponential in the expression's length. The exponential cost of matching cannot be overcome by merely improving matching algorithms, because the problem of matching arbitrary conjunctive expressions is intrinsically difficult. To see this note that the problem of subgraph isomorphism,

[3]This match cannot be performed via unification (which is linear-time) because unification is not defined for conjunctive expressions. See [22] for a precise definition of unification.

which is known to be NP-hard [17, page 202], can be reduced to the problem of matching a conjunctive expression [36, page 184].

This observation is important because, although the cost of matching a rule against a state of size s is exponential in the rule's length, match cost is *polynomial* in s for any rule of bounded length. If the rule's length is bounded by k, its match cost is $O(s^{k+1})$ which is polynomial in s. This observation holds for any finite set of control rules. Consequently, we have the following:

Proposition 1 *Any finite set of control rules, which reduces the number of nodes expanded by a problem solver from an exponential in s to a polynomial in s, will achieve polynomial-time problem solving (and average speedup for sufficiently large values of s).*

Note that this observation holds only when the problem solver is able to reach a point where no further learning is necessary. Thus, the number and size of the control rules does *not* scale with s. As argued in Section 3.3.3, this favorable situation can occur when the problem solver is learning control rules based on nonrecursive proofs. If, on the other hand, the problem solver is forced to continue learning additional control rules as the state size s increases, then it will not converge to a finite set of control rules in the limit. As explained in Section 3.3.1, this situation occurs when the problem solver is forced to learn from unbounded recursive proofs.

3 EBL Problem Solvers

This section considers EBL problem solvers, a restricted class of meta-level problem solvers that acquires control rules via EBL. The section presents the EBG framework, which defines standard EBL terminology, and describes the relationship between EBL's proofs and the length of EBL's control rules, a prerequisite to the analysis in Section 3.3.

3.1 The EBG Framework

Mitchell, Keller and Kedar-Cabelli [38] describe a model of EBL, called Explanation-Based Generalization (EBG) that articulates many of the aspects common to various EBL systems (see also [8]). Two of the major contributions of the EBG model are the identification of explanations with proofs, giving a precise meaning to the term "explanation," and the clear specification of the inputs and output of EBL shown in Table 3. The inputs consist of a target concept, a theory for constructing explanations, a training example, and an operationality criterion. The output is a sufficient condition for recognizing the target concept. The operationality criterion is intended to ensure that the sufficient condition can be used to recognize instances of the concept efficiently.

According to the model, EBL systems prove that the training example is an instance of the target concept and output the *weakest precondition* of the proof. The weakest precondition of a proof is the (weakest) sufficient condition under which the proof succeeds. Section 3.2, below, defines this notion more precisely (cf. [9, 29]).

Given:

- *Target Concept Definition:* A concept definition describing the concept to be learned. (It is assumed that this concept definition fails to satisfy the Operationality Criterion.)
- *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 concept definitions, specifying the form in which the learned concept definition must be expressed.

Determine:

- A generalization of the training example that is a sufficient concept description for the target concept and that satisfies the operationality criterion.

Table 3: Mitchell *et al.*'s specification of EBG.

We can use EBG terminology to characterize the control rules acquired by EBL problem solvers. Consider the target concept "success." A training example is a portion of the problem solver's trace exemplifying the success of a particular candidate (e.g. an operator). Based on the training example, EBL proves that the operator invariably succeeds under certain conditions. The applicability conditions of the resulting control rule are the weakest precondition of that proof. The rule's recommendation is a function of what is being proved. In PRODIGY/EBL, for example, proving that an operator succeeds yields an operator preference rule; proving that a binding fails yields binding rejection rule, etc.

3.2 Proof Trees

This section defines the notion of a weakest precondition (WP), and shows how WP length is determined by proof-tree size. The section then analyzes the relationship between proof-tree size and recursive proofs and shows that, given any finite rule set, the size of a nonrecursive proof tree is bounded whereas the size of a recursive proof tree is not.

For simplicity, I restrict the discussion to Horn Clauses. The analysis is easily extended to more expressive languages. I define a proof to be the instantiation of a proof tree. A proof tree is an AND-tree rooted in the literal derived by the proof. Each internal node in the tree is derived by some Horn Clause whose consequent or "head" unifies with the literal at the node. The variable substitutions from the unification are propagated to the rule's antecedent conditions or "body," which is denoted by a collection of nodes connected by an AND-arc. Note that a proof tree, as defined here, is only partially instantiated. An illustrative proof tree appears in Figure 5. The conjunction of the tree's leaves constitutes the proof tree's

WP.

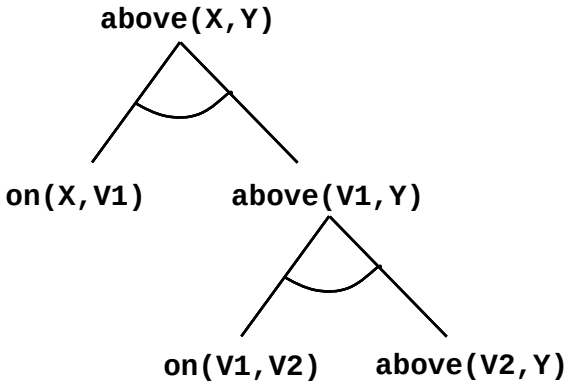

Figure 5: A recursive proof tree.

WP length is the number of literals in a proof's WP. WP length can be characterized in terms of the corresponding proof tree: it is the number of leaf nodes in the tree. It follows that, in the worst case, WP length is:

- polynomial in the proof-tree's branching factor.
- exponential in the proof-tree's depth.

These observations are of interest because, as shown in Section 2.4, the cost of matching a logical expression depends on its length. Since EBL problem solvers match the WP of EBL's proofs, the cost of matching EBL's control rules depends on WP length. The following sections relate WP length to the recursive nature of the proof.

A proof is said to be *recursive* when it is derived from a recursive proof schema. A proof schema is said to be recursive when it yields proofs that derive one literal from a second literal that unifies with, but is not identical to, the first. For example, the inference rule:

```
above(X,Y) :- on(X,V1), above(V1,Y).
```

can be used to define a recursive proof schema in which `above(X,Y)` is derived from `above(V1,Y)`, `above(V1,Y)` can be derived from another application of the inference rule and so on. Of course, recursive derivations need not be immediate in general. The *recursive depth* of a proof is the maximal depth to which any recursion (there may be several) is expanded or "unfolded" in the proof. The recursive depth of the proof in Figure 5 is two. A recursive proof is said to be *nontrivial* when the body of its recursive inference rule contains more than one literal. For example, the inference rule `p(X) :- p(Y)` is trivial, but the `above` inference rule is not.

When each nontrivial recursive call replaces a proof tree leaf by two leaves, WP length increases linearly with recursive depth. In Figure 5, for example, `above(X,Y)` is replaced by `on(X,V1), above(V1,Y)`. When the recursion is carried a step further, `above(V1,Y)` is replaced by `on(V1,V2), above(V2,Y)` and so on. When the number of recursive calls

increases with each recursive expansion, as in the inference rule:

```
above(X,Y) :- above(X,Z), above(Z,Y).
```

WP length increases exponentially with recursive depth. In general, we have the following:

Proposition 2 *The WP length of a nontrivial recursive proof increases (at least) linearly with the proof's recursive depth.*

This proposition suggests that recursive proofs expanded to nontrivial depths will result in lengthy WPs. This suggestion is corroborated by the empirical results in Section 6.

A proof is said to be *nonrecursive* when it is not generated by a recursive proof schema. Given a finite set of inference rules, the depth of an arbitrary nonrecursive proof tree is bounded by the number of literals in the rule set that do not unify with each other. Since the branching factor of a proof tree is bounded by the maximal number of antecedent conditions to an inference rule in the theory, it follows that the size and WP length of nonrecursive proof trees are bounded as well. Thus, we have the following:

Proposition 3 *Given a finite rule set, the WP length of a nonrecursive proof tree is bounded.*

Note that this result applies to nonrecursive proof *trees*, where a tree is defined to be rooted in a single literal. Naturally, if the number of literals at the root is unbounded so is the size of the proof.

3.3 A Complexity Analysis of EBL Problem Solvers

The previous section described EBL's role in meta-level problem solvers and showed that whereas the WP length of arbitrary nonrecursive proof trees is bounded, in any finite rule set, the WP length of recursive proofs is not. This section considers the implications of this observation for the effectiveness of EBL. Note that although learning from recursive *proofs* is shown to be problematic in many cases, learning in recursive *problem spaces* is not.[4] Utilizing a variety of target concepts (e.g. failure) facilitates learning from nonrecursive proofs even in recursive problem spaces. The following discussion does *not* suggest, therefore, that EBL is ineffective when confronted with recursive problem spaces.

3.3.1 Recursive Proofs

Since the cost of matching a conjunctive expression is exponential in its length (Section 2.4), and since WP length increases (at least) linearly with recursive depth (Proposition 2), it follows that the cost of matching the WP of a recursive proof tree is exponential (or worse) in the tree's recursive depth. In many cases, recursive depth increases with state size. Consider,

[4] A problem space is said to be recursive when there are problems in the space whose solution necessitates recursive plans. A recursive plan is a plan generated by a recursive plan schema. A plan schema is said to be recursive when it generates plans where, in order to achieve one literal, the plan achieves a second literal that unifies with, but is not identical to, the first.

for example, a Blocksworld state that consists entirely of a tower of blocks. Proving that the bottom block of the tower can be cleared is recursive and the depth of the recursion is equal to the height of the tower or, equivalently, to the state size. In general, we have the following:

Proposition 4 *When the recursive depth of the proof trees generated by EBL increases linearly with the state size s, the cost of matching the weakest preconditions of EBL's proofs increases exponentially (or worse) with s.*

Thus, as the problem solver encounters larger and larger problems, and correspondingly deeper recursions, it is forced to acquire control rules with successively longer antecedents, and the match cost of the antecedents increases exponentially. Clearly, this result refers to worst-case match cost. If the predicates in a proof tree are unique attributes, for example, the cost of matching the proof tree's WP can be linear in its recursive depth.[5]

Even in this case, however, rules learned from unique-attribute recursive proofs can result in exponential match cost due to insufficient reduction in the number of nodes expanded. To see this consider Equation 1 (Section 2.2). The equation demonstrates that the total overhead due to learning is the overhead of matching control rules, at a given node, multiplied by the number of nodes expanded. If the number of nodes expanded after learning, η_τ, remains exponential in s, then the total overhead of matching τ is exponential in s, even when the per-node match cost of τ is *constant*, let alone linear in rule length. Thus, we have the following:

Proposition 5 *When η_τ is exponential in s, the total overhead of matching τ is exponential in s.*[6]

Unique-attribute rules are particularly susceptible to the problem of insufficient pruning because recoding a domain theory into unique attributes, to reduce the per-node match cost of learned rules, can lead EBL to produce highly-specific rules [56]).

Another problem with recursive proofs is *recursion-depth-specificity*, the WP of a recursive proof only matches states that support the very same recursive expansion the WP was learned from. As a result, when EBL's proofs are recursive, distinct rules are learned at every recursive depth to which the proofs are expanded [4, 43, 48]. For example, PRODIGY/EBL learns distinct Blocksworld control rules for clearing the bottom block of a two-block tower, a three-block tower, and so on. Thus, 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 set of control rules learned from recursive proofs will fail to cover all possible Blocksworld problems.

As a result, when learning from recursive proofs, EBL is particularly sensitive to both problem distribution and choice of training examples. To make this observation more precise

[5] Roughly, unique attributes (also known as *determinate literals*) are predicates whose arguments have unique bindings thereby restricting the branching factor in their match tree to one [56]. The cost of matching a unique-attribute rule scales only linearly with rule length.

[6] Paul Rosenbloom notes that even though the total overhead of τ is exponential in s, the *ratio* $(\kappa + \mu_\tau)\eta_\tau/\kappa\eta$ is constant, so long as μ_τ is constant. See [12] for additional discussion of this issue.

I define the recursive depth of a problem, relative to a given EBL problem solver, to be the maximum of the recursive depths of the proofs EBL has to learn from in order to eliminate backtracking on that problem. Suppose EBL learns from recursive proofs whose maximal depth is ρ, and subsequently encounters a problem whose recursive depth is ρ', where $\rho' > \rho$. The problem solver's search will be unguided until it reduces its problem into subproblems whose recursive depth is ρ. If we assume that the number of nodes expanded by the problem solver is exponential in $\rho' - \rho$ in the worst case then, by Proposition 5, we have that the total match cost overhead of EBL's control rules is exponential in $\rho' - \rho$. It follows that, unless $\rho' - \rho$ is bounded, EBL cannot achieve polynomial-time problem solving by learning from recursive proofs. This observation applies to unique-attribute proofs as well.

3.3.2 Generalization-to-N

In some cases, the recursion-depth-specificity of EBL's control rules can be overcome using a technique known as *generalization-to-N* [4, 5, 27, 43, 48, 47, 52]. Generalization-to-N algorithms analyze the recursive expansion in their training example, induce a general representation of the recursion's structure (e.g. a finite automaton or a context-free grammar), and construct a new recursion based on this representation. This recursion is expanded "at run-time," when the rule produced by generalization-to-N is matched.[7]

Thinking of a recursive proof as a string where each inference rule is a symbol helps to understand how generalization-to-N constructs new recursions. In essence, generalization-to-N heuristically detects repeated substrings in the proof and generates rules that allow these substrings to repeat an arbitrary number of times. For example, the technique can generalize from $aaab$ to a^*b or from $abcabc$ to $(abc)^*$. However, the formalism used to represent recursion imposes limitations on the process. If the formalism is finite automata, for example, then generalization-to-N cannot generalize from $aabb$ to the expression a^nb^n, which cannot be represented as a finite automaton. Furthermore, many strings are ambiguous. The string $abaaba$ is plausibly generalized to $(aba)^*$ or to aba^*ba. In effect, generalization-to-N imposes an inductive bias on the generalization of recursions, based on its heuristics for detecting and generalizing repeated substrings, which can lead it to overgeneralize (cf. [3]).

Thus, although generalization-to-N addresses the recursion-depth-specificity of EBL's control knowledge, it will not always generalize a recursion appropriately or adequately. Furthermore, the technique yields rules that expand recursions at run time. The match cost of this expansion is still exponential in its depth in the worst case.

3.3.3 Nonrecursive Proofs

Nonrecursive proofs avoid both the unbounded WP length and the recursion-depth-specificity associated with recursive proofs. As Proposition 3 shows, the size of nonrecursive proofs is

[7] For this reason, generalization-to-N is outside the scope of the EBL problem solvers model employed here. Indeed, EBL problem solvers such as Soar and PRODIGY do not use generalization-to-N. Nevertheless, the above discussion shows that generalization-to-N does not solve all of the problems associated with recursive proofs.

bounded in any finite rule set. Unlike recursive proof trees, the WP length of nonrecursive proofs does not scale with the state size s. By the complexity argument in Section 2.4, it follows that the match cost of a nonrecursive control rule (i.e., a control rule derived from a nonrecursive proof) is polynomial in s.

Proposition 6 *Given a finite rule set, the cost of matching the weakest precondition of an arbitrary nonrecursive proof tree against a state of size s is polynomial in s.*

Since the WP length of nonrecursive proof trees is bounded, given any finite rule set, it follows that the number of distinct nonrecursive proof trees is also bounded. Furthermore, by Proposition 6, the cost of matching the WPs of an arbitrary set of nonrecursive control rules is polynomial in s. Thus, if EBL acquires a set of nonrecursive control rules that reduces the number of nodes expanded by the problem solver to a polynomial in s then, by Proposition 1, EBL has achieved polynomial-time problem solving. Note that, unless the depth of the problem solver's recursions is bounded, EBL cannot achieve polynomial-time problem solving by learning from recursive proofs.

The depth of nonrecursive proof trees is not only bounded but, often, quite small. Consider, for example, rule sets that contain no constants (e.g. the Blocksworld problem space definition). The depth of a nonrecursive proof tree, in such a rule set, is bounded by the number of predicates in the set. Even this bound is tight only when every predicate in the set participates in some nonrecursive inference chain. In practice, the depth of nonrecursive proof trees is even smaller, resulting in highly compact WPs. Thus, often, the per-node cost of matching these WPs is not merely an arbitrary polynomial, as guaranteed by Proposition 6, but a low-order one.

3.4 Conclusion

This section presented an idealized model of EBL problem solvers, and used a complexity analysis to draw a dichotomy between recursive and nonrecursive proofs. The weakest preconditions (WPs) of nonrecursive proofs have bounded length and polynomial per-node match cost in any finite rule set. The number of distinct literals in a rule set is a natural bound on the maximal WP length for nonrecursive proofs in that rule set. In contrast, the WPs of recursive proofs have unbounded length and exponential match cost. Furthermore, recursive proofs are recursion-depth-specific whereas nonrecursive proofs are not. It follows that, unless recursive depth is bounded, EBL can only achieve polynomial-time problem solving when learning from nonrecursive proofs.

4 The Structural Thesis

The previous section drew a dichotomy between recursive and nonrecursive proofs. This section demonstrates that we can discriminate between recursive and nonrecursive proofs using Problem Space Graphs (PSGs), a graph representation of problem spaces. The basic tenet of this section is the following.

The Structural Thesis: *there is a correspondence between PSGs and the proofs used by EBL systems to generate search-control knowledge. As a result, automatic PSG analysis can be used to compute the weakest preconditions of the proofs and to discriminate between recursive and nonrecursive proofs.*

The section is organized as follows. Section 4.1 defines Problem Space Graphs (PSGs). Section 4.2 illustrates how a PRODIGY/EBL proof can be translated into a PSG subgraph, explains how to discriminate between recursive and nonrecursive proofs, and how to compute the weakest precondition of a proof based on its PSG correlate. Finally, Section 4.3 considers the scope of the thesis, and its limitations. The motivation for focusing on PRODIGY/EBL is two-fold. First, PRODIGY/EBL is a state-of-the-art EBL system that is interesting in and of itself. Second, since PRODIGY and its EBL module are well-documented and publicly available, experiments using PRODIGY (see Section 6) can be verified and replicated by other researchers. For the sake of generality, the analysis abstracts away from PRODIGY/EBL's singular features such as compression analysis, empirical utility evaluation, example selection heuristics, etc.

4.1 Problem Space Graphs (PSGs)

The PSG represents all possible paths in a backward-chaining search through a problem space. The PSG is derived from the problem space definition via partial evaluation or symbolic execution.[8] The PSG is precisely specified below.

4.1.1 Specifying the PSG

A problem space can be represented as a set of disjoint graphs (or PSGs), each rooted in a distinct achievable literal. A depiction of the PSG rooted in the Blocksworld literal `(holding V)` appears in Figure 6. The PSG is a directed, acyclic AND/OR graph. The root literal is connected, via OR-links, to the operators that match it. The operators are partially instantiated via the variable substitution from the match. Each operator is connected, via AND-links, to its partially instantiated preconditions. Each precondition is connected to the operators that match it, and so on. Thus, the PSG's nodes are an alternating sequence of (sub)goals and operators, and the PSG's edges are an alternating sequence of AND-links and OR-links.

When two operators that have a common goal share a precondition, a single node designates that precondition in the PSG. Both operators are linked to the node via AND-links, so the graph is not a tree in general. In Figure 6, for example, both pick-up(V) and UNSTACK(V,V2) have `(clear V)` as a precondition. Consequently, both operators have AND-links to that node.

As described thus far, recursion would result in infinite PSGs. In fact, the PSG has well-defined termination conditions under which PSGs are provably finite [11]. To state these

[8] The PSG should not be confused with a state space graph. The state space graph's nodes are states, and its edges are instantiated operators, whereas PSG nodes are literals or operators, and PSG edges are AND/OR links.

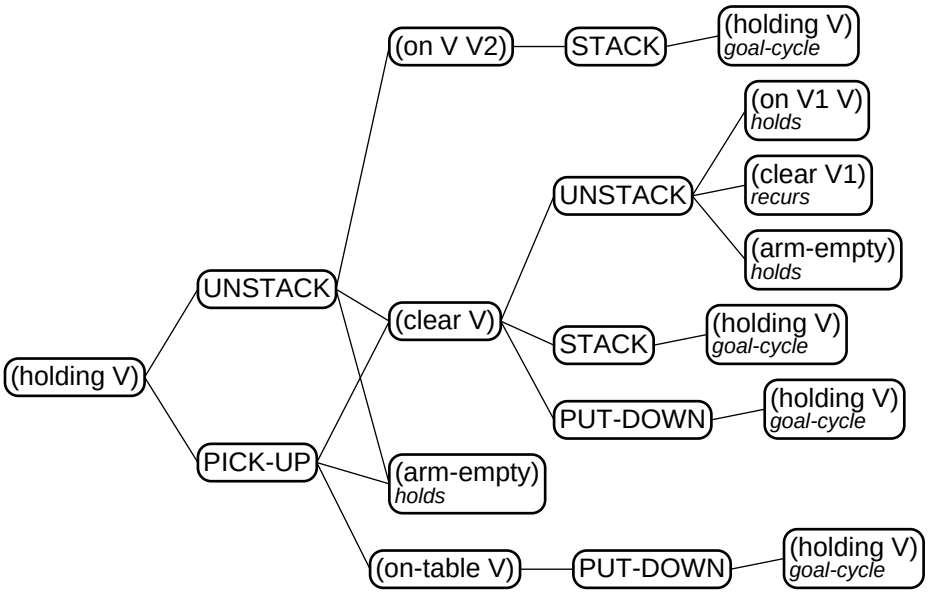

Figure 6: The `holding` PSG.

conditions precisely I introduce the notion of a literal's *ancestors*, which are the set of literals on the unique path from the literal to the root of the PSG. The ancestors of a PSG node n represent the set of subgoals PRODIGY would generate before subgoaling on the literal at n. PSG expansion is terminated at n if and only if:

- No operators can achieve the literal at n.
- The literal is identical to one of its ancestor literals (I refer to this condition as a *goal cycle*).
- The literal unifies with, but is not identical to, one of its ancestors (I say that the literal *recurs*).
- The literal is necessarily satisfied, given that its ancestors are subgoals waiting to be achieved. In the `holding` PSG, for example, `(on V1 V)` is labeled *holds* because, to reach it, PRODIGY subgoals on `(holding V)` and `(clear V)` and, if a block is neither clear nor held, then there must be some block on it.[9]

An algorithm that derives PSGs from problem space definitions appears in [11].

The semantics of PSG nodes can be defined relative to PRODIGY's problem solving. A literal node appears in the PSG if and only if there is a world state in which the problem

[9]The `(arm-empty)` nodes are labeled *holds* to keep the PSG's depiction compact. The label is justified by noting that PUT-DOWN can invariably achieve `(arm-empty)`.

solver subgoals on an instance of that literal in trying to achieve its current goal. An operator node appears in the PSG if and only if there is a world state in which PRODIGY will back-chain on an instance of the operator in trying to achieve its current goal. A literal node has an OR-link to an operator node if and only if the operator matches the literal. An operator has an AND-link to a literal if and only if the literal is a precondition to the operator.

4.2 The Correspondence of PSGs to EBL's Proofs

The notion of representing programs (or problem spaces) as graphs is well-known in computer science (see, for example, [24, 58]). It is interesting to note, however, that PRODIGY/EBL's proofs correspond to PSG subgraphs.[10]

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 are analogous, 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 V V2)`, cannot be achieved in the context of `(holding V)`. `On` cannot be achieved because all of the relevant operators (STACK is the only one) cannot be executed. STACK cannot be executed because trying to achieve one of its preconditions, `(holding V)`, results in a goal cycle. That is, the problem solver subgoals on `(holding V)` in the process of trying to achieve `(holding V)`, which leads to an infinite loop. The PSG subgraph in Figure 7 is a representation of this proof. The weakest precondition of the proof can be determined by analyzing the subgraph. In this case, the goal cycle occurs when the goal is `(holding V)` and the precondition `(on V V2)` is not true in the current state, leading PRODIGY to back-chain on the operator STACK in order to satisfy the precondition. This proof yields the rejection rule that appears in Table 2, Section 2.1.

Figure 7: The PSG subgraph corresponding to the proof that UNSTACK fails to achieve `holding`.

The example illustrates the general correspondence between PSG subgraphs and PRODIGY/EBL's proofs. This relationship is not surprising if we consider that, by construction, the PSG explicitly represents the failure and success of different problem-solving paths which is precisely the subject matter of EBL's proofs. See [10] for a more detailed description of the mapping between PSGs and EBL's proofs.

[10] The recursive expansions in PRODIGY/EBL's recursive proofs are represented as leaf nodes.

The correspondence described above facilitates traversing the PSG searching for subgraphs corresponding to nonrecursive proofs (i.e. subgraphs none of whose leaves are labeled *recurs*) and deriving control rules based on these subgraphs. That is precisely what the STATIC program does. A complete description of STATIC's algorithms appears in [11].

4.3 Scope of the Structural Thesis

Although the structural thesis applies to many existing EBL problem solvers (e.g., [27, 1, 52]) its scope is limited for a number of reasons. First, although most commonly-used target concepts (success, failure, etc.) map naturally to PSG subgraphs, other target concepts (e.g., state cycles) do not. Second, PSGs have only been developed for backward-chaining problem solvers. Developing a PSG representation for forward-chaining problem solvers is a topic for future research. Finally, consider a domain theory that augments and transcends the problem space definition. For example, consider a domain theory that contains inference rules which can assess the outcome of a complex problem space recursion using a simple reasoning process.[11] When the domain theory and the problem space are divorced in this manner, the structural thesis is false. However, a sibling thesis may be formulated linking a graph representation of the domain theory to EBL's proofs. Ultimately, determining the precise scope of the thesis is an empirical enterprise. Further analysis of a variety of EBL problem solvers, domain theories, and target concepts is required to determine whether the thesis applies or not.

5 Choosing What Proofs to Learn From

An EBL system can have access to a wide variety of target concepts, and there are many different ways to prove that a given training example is subsumed by a particular target concept. Each such proof yields a different control rule, some of which are considerably more useful than others. When analyzing a training example, EBL merely computes the weakest precondition of a *particular* proof. It is by no means guaranteed to find "the best" control rule, or even an effective one [14]. Indeed, EBL frequently derives ineffective control knowledge in practice.

Based on the complexity analysis in Section 3, this section presents heuristics that help EBL to overcome this problem by indicating which proofs EBL should learn from. The complexity analysis does not prove that the heuristics are appropriate in every case. In fact, it is easy to manufacture cases in which the heuristics are inappropriate; I discuss such cases in Sections 5.2 and 5.1. Nevertheless, the experiments in Section 6 show that the heuristics are valuable in practice.

The heuristics are attractive because they constitute an *a priori* basis for generating effective control knowledge. A number of researchers have developed *post hoc* mechanisms for

[11] Imagine a Blocksworld domain theory that specified the Cartesian coordinates of each block. Determining whether one block is above another, while recursive in the problem space, merely requires comparing X-coordinates in the domain theory.

evaluating control knowledge by measuring its effectiveness on a sample of problems [19, 20, 32, 54]. The advantage of *post hoc* approaches is that they "consider" problem distribution, whereas *a priori* approaches do not. Sample-based approaches have several disadvantages, though. First, the complex interactions between different control rules mean that individual control rules cannot be tested in isolation. As a result, the *post hoc* mechanisms rely on hill climbing to incrementally (and heuristically) evaluate sets of control rules. Second, the mechanisms' performance is sensitive to the composition and ordering of their sample. With some probability, a randomly-generated sample will not accurately represent the population from which it is drawn. When this is the case, the performance of a sample-based mechanism is arbitrarily bad. Large sample sizes ensure that the probability of this event is low but, to provide such guarantees, the mechanisms proposed in [19, 20, 54] typically require their problem solver to solve literally thousands of problems. Even the *ad hoc* rule-evaluation mechanism used by PRODIGY/EBL [32] accounts for a significant fraction of PRODIGY/EBL's learning time [10, chapter 8]. Finally, sample-based rule selection is distribution-dependent. If the problem distribution changes, the rule selection algorithm has to be invoked again, on a fresh sample, to ensure that an appropriate rule set is chosen. Thus, as recognized by [19], *a priori* heuristics for generating effective control rules have distinct advantages.

5.1 The Nonrecursive Heuristic

The nonrecursive heuristic can be stated as follows:

```
Learn from nonrecursive proofs.
```

As pointed out earlier, control knowledge represents a tradeoff between reduced problem-space search and increased match cost. Proposition 6 suggests that this tradeoff may be favorable when learning from nonrecursive proofs. In general, learning from nonrecursive proofs is *not* guaranteed to be effective for several reasons. The analysis in Section 3.3 presupposes an exponential search. If unguided search is polynomial in s, for example, then the nonrecursive heuristic can easily result in a significant average slowdown. In addition, although the per-node match cost of nonrecursive control rules increases only polynomially with state size s, their total match cost overhead can still be exponential when the number of nodes expanded by the problem solver, after learning, is exponential (Proposition 5). Finally, due to the *a priori* nature of the nonrecursive heuristic, it is easy to define problem distributions where nonrecursive control rules are rarely applicable and thus ineffective. The experiments reported in [19, 42] demonstrate this point.

Despite the caveats enumerated above, the experiments described in Section 6 demonstrate the value of the nonrecursive heuristic in practice. Additional work is required to precisely identify the classes of problems in which it is effective. Augmenting the heuristic with a variety of example-based and sample-based approaches as suggested in [10] and investigated in [19, 42] is a worthwhile direction for future work.

5.1.1 Bounded-Depth Recursion

Bounded-depth recursive proofs are distinct from nonrecursive proofs. When the size of a recursive proof is bounded, the analysis used to derive Proposition 6 applies, and the cost of matching the proof's WP is polynomial in the state size s. One might be tempted to conclude that a nonrecursive proof is merely "a recursive proof unfolded to depth zero." This is *not* the case. Consider, for example, the PSG displayed in Section 4.1.1. The subgraph containing the root, `(holding V)`, the operator UNSTACK, and UNSTACK's preconditions, corresponds to a depth-zero recursive proof that UNSTACK can achieve `(holding V)`. In contrast to the WP of a nonrecursive proof, this proof's WP is recursion-depth-specific; it does not apply when `V` is covered by one or more blocks. Each recursive invocation of UNSTACK, aimed at achieving `(clear V)`, results in a deeper recursive proof with a longer recursion-depth-specific WP.

In contrast, the PSG subgraph displayed in Figure 7 corresponds to a "true" nonrecursive proof. The proof is not derived from a recursive proof schema; the same proof can be used to explain the failure of UNSTACK regardless of the number of blocks in the problem, the recursive depth of the problem solver's plan, etc. See Figure 8 (Section 6.1.3) for another example of a nonrecursive proof.

5.1.2 Fortuitous Recursion

The effectiveness of the nonrecursive heuristic is enhanced by a structural feature of problem spaces I refer to as *fortuitous recursion*. I introduce this idea using a Blocksworld example. Consider a plan to achieve `(holding block-1)` when `block-1` is at the bottom of a three-block tower: `block-3, block-2, block-1`. In order to grasp `block-1`, the problem solver has to achieve the subgoal `(clear block-1)`. To achieve `(clear block-1)`, it has to achieve `(clear block-2)` and so on. The plan is generated by expanding a recursive plan schema. The depth of the expansion is determined by the height of the block tower.

Proving that a recursive plan succeeds gives rise to a recursive proof. However, in some cases, we can prove that a recursive plan fails using a nonrecursive proof. Consider achieving `(holding block-1)` using the UNSTACK operator which is shown in Table 1 in Section 2.1. A nonrecursive proof shows that the plan fails, for our three-block tower, because using UNSTACK requires subgoaling on its `(on block-1 block-Y)` precondition, which leads to a goal cycle. The proof is described in Section 4.2, and its PSG representation appears in Figure 7 in that section.

The proof yields a sufficient condition for the failure of UNSTACK that is the basis of the operator rejection rule in Table 2, Section 2.1. The rule rejects UNSTACK, when its antecedent is matched, but does not necessarily guarantee that UNSTACK will achieve its goal when the rule's antecedent is *not* matched. When the antecedent is not matched (i.e. `(on block-1 block-Y)` holds in the current state), the success of UNSTACK depends on whether its other preconditions, `(clear block-1)` and `(arm-empty)`, can be achieved.

In fact, it turns out that `(arm-empty)` is easily achieved by putting down any block that is being held and that, invariably, there is some recursive plan that will succeed in achieving `(clear block-1)`. The Blocksworld turns out to have a fortuitous property: whenever `(on`

`block-X block-Y)` is matched, UNSTACK will achieve `(holding block-X)`. Thus, the rule for rejecting UNSTACK defines a condition that turns out to be both necessary and sufficient for failure; if the applicability conditions of the rejection rule in Table 2 are not matched, plans to achieve `holding` via UNSTACK *are* guaranteed to succeed. This property is not guaranteed by EBL. It is a structural feature of the Blocksworld.

The above example shows how a nonrecursive proof of failure can yield a necessary and sufficient condition for the failure (or, equivalently, success) of an operator. In this example, EBL need not form a recursive proof in order to eliminate backtracking. The essential feature of the example is that the success of a nonrecursive portion of a recursive plan (namely, `(on block-X block-Y)`) guarantees the success of the entire plan. When this is the case for all plans in which the recursion appears, the recursion is said to be "fortuitous." In the Blocksworld, the recursion required to clear a block is fortuitous.

Clearly, fortuitous recursion enhances the effectiveness of nonrecursive proofs of failure. When all problem-space recursions are fortuitous, and EBL is trained appropriately, EBL can eliminate backtracking by learning solely from nonrecursive proofs. Given a finite rule set, the number of distinct nonrecursive proofs is finite. Thus, by Proposition 6 in Section 3.3.3, it follows that EBL is guaranteed to achieve polynomial-time problem solving and average speedup in this case.

5.2 The Recursive Heuristic

The recursive heuristic is a negative heuristic that complements the nonrecursive one:

```
Do not learn from recursive proofs.
```

Proposition 4 shows that, in the limit, learning from recursive proofs trades an exponential search for an exponential match. Furthermore, as argued in Section 3.3.1, for any fixed set of recursive control rules there are problem distributions on which the set results in exponential overhead and average slowdown of the problem solver.

Again, it is easy to manufacture cases in which the recursive heuristic is inappropriate, and learning from recursive proofs is actually beneficial. Essentially, when the depth of a certain recursion is bounded in a given problem distribution, then the recursive depth of proofs that unfold this recursion (and no others) is bounded, and the asymptotic analysis of nonrecursive proofs in Section 3.3.3 applies to these "bounded-depth" recursive proofs. In general, it is impossible to anticipate *a priori* that a recursion will have bounded depth in the absence of precise *a priori* knowledge of the problem distribution.[12] Furthermore, bounds on the depth of recursive unfolding can change with the problem distribution whereas the bound on nonrecursive unfolding is fixed in any given problem space. Thus, the recursive heuristic remains the best *a priori* heuristic we can formulate.

The recursive heuristic is particularly important because of the practice of training EBL on relatively simple problems that exhibit shallow recursions. As pointed out in [14], this practice often helps EBL find simpler and more compact explanations of its problem solver's

[12] See [27, 46] for exceptions in certain special cases.

behavior. However, in the case of recursive proofs, the practice can lead to a disparity between the shallow recursions in training examples and deep recursions on test cases. As Section 3.3.1 indicates, such disparity leads to exponential overhead due to EBL.

5.2.1 The Recursive Heuristic In Practice

Experimental inquiries have shown that control knowledge learned from recursive proofs can be effective when a tight bound is imposed on recursive depth in domains such as the Eightpuzzle [26, 55], or on highly skewed problem distributions [47, 56], but is ineffective in many other cases [10], even in the presence of generalization-to-N [52]. Determining exactly how skewed the distribution of problems has to be for a problem solver to benefit from recursive EBL proofs depends on the specific search space, problem solver, and matcher involved. [52] estimated this parameter empirically using recursive macro rules learned by BAGGER2 [47] in a circuit synthesis domain. They report that "unless the percentage of problems in the future queries that are solvable by the macro rules alone exceeds 60 percent for input expressions of arity 3..and 80 percent for arity 5, BAGGER2 style macro rules are not useful."

The above discussion suggests that, due to insufficient pruning, EBL will fail to achieve average speedup when learning from recursive proofs on sufficiently uniform problem distributions. Additional work is required to precisely characterize "sufficiently uniform" distributions and to formally analyze the relationship between distributional skew and the effectiveness of EBL when learning from recursive proofs.

5.3 Complementary Target Concepts

Learning from the success of a recursive plan yields a recursive proof. Although I have argued against learning from recursive proofs, I have also argued that EBL can be effective in recursive problem spaces. Indeed, most of the plans in PRODIGY/EBL's benchmark problem spaces are recursive, yet PRODIGY/EBL is quite effective in these problem spaces [30]. This paradox is resolved by observing that PRODIGY/EBL learns from failure and goal interaction in addition to learning from success. As the Blocksworld example in Section 5.1.2 illustrates, proving that a plan will fail can be nonrecursive even when the plan itself is recursive. In fact, five failure proofs yield all the control rules necessary to make appropriate operator and bindings choices on any Blocksworld problem. The failure proofs are nonrecursive, yielding rules that are compact, general, and cheap-to-match.

I refer, informally, to target concepts (e.g. failure) whose proofs are not direct translations of the problem solver's plans as complementary target concepts. *When recursive plans abound, complementary target concepts are EBL's primary means of finding nonrecursive proofs.* It follows that such target concepts will enhance EBL's effectiveness in many cases.

Practical considerations suggest another argument for learning from complementary target concepts. Even after analyzing a large number of training examples, EBL's coverage is often incomplete in practice—it is unable to guide the problem solver on every control decision in every problem. As a result, the problem solver frequently departs from the path

to a solution. When backtracking chronologically (as in PRODIGY or Prolog), the problem solver exhausts all the available alternatives at a node before backtracking from the node. Thus, in the absence of control rules that curtail backtracking, departing from a solution path often results in extensive backtracking. Control rules learned from success ("success rules") attempt to keep the problem solver on a solution path. Unfortunately, success rules will never fire at a node that is not on any solution path relative to the current goal, because the state at such a node can never match the weakest preconditions of a successful plan. Thus, once the problem solver has diverged from a solution path, the only control rules that can curtail backtracking are control rules learned from complementary target concepts. It follows that complementary target concepts are particularly useful.

Most existing EBL systems (e.g., [5, 47, 52]) and partial evaluators (e.g., [60, 46, 27]) do not use complementary target concepts. Thus, as Minton [30] argues, PRODIGY/EBL's use of multiple (and, in particular, complementary) target concepts is an important extension of the EBL method. The structural theory supports and elaborates Minton's contention, yielding an operational recommendation to the designers of EBL problem solvers: *use complementary target concepts.*

5.4 Conclusion

This section described two *a priori* heuristics for choosing what proofs to learn from in EBL. The heuristics yield a simple criterion for proof choice: learn only from nonrecursive proofs. To increase the number of nonrecursive proofs available to EBL, I recommend developing a host of *complementary target concepts*, concepts (e.g. failure in PRODIGY/EBL) whose proofs do not mirror the problem solver's plans. I describe a number of cases in which this criterion is imperfect, and others have appeared in the literature [19, 42]. The following section argues that, though imperfect, the heuristics formulated above are valuable in practice.

6 Experimental Validation

This section reports on an array of experimental results that are explained, informally, using the structural theory. In particular, the theory explains PRODIGY/EBL's success in Minton's experiments [30] as well as the experimental results listed in Section 1.2.

6.1 Explaining PRODIGY/EBL's Success

> *"It is rare that one sees an AI system evaluated carefully by anyone other than its creator."*[2]

To demonstrate the explanatory power of the structural theory, this section utilizes it to explain PRODIGY/EBL's success in Minton's experiments. Each of PRODIGY/EBL's benchmark problem spaces exhibits several structural features that facilitate PRODIGY/EBL's success. The section discusses these features abstractly, and explains how the features account

for PRODIGY/EBL's success in each space. Table 4, at the end of the section, summarizes the explanation using the terms defined below.

Problem Distribution Minton's problem distributions play an important role in explaining his results. First, the problem distributions determine the recursive depths of PRODIGY/EBL's proofs. Second, the distributions determine the aspects of each problem space that PRODIGY is exposed to. Each of the problem spaces contains some recursive structure that PRODIGY/EBL is unable to learn about effectively (see Section 6.1.1 for an example). PRODIGY/EBL's success in Minton's experiments is due, in part, to the paucity of problems that exercise this recursive structure.

Short Nonrecursive Proofs of Failure. Failure, in PRODIGY/EBL's problem spaces, is often explainable by nonrecursive proofs. As it turns out, the depth and the branching factor of PRODIGY/EBL's nonrecursive failure proofs are small, yielding compact control rules whose applicability conditions contain no more than two or three literals in many cases. As a result, control rules learned from failure are both general and cheap-to-match making the tradeoff between increased match cost and search reduction favorable even for relatively small problems; PRODIGY/EBL relies heavily on learning from failure.

Fortuitous Recursions Many of the recursions in PRODIGY/EBL's problem spaces are fortuitous (see Section 5.1.2) which means that plans containing the recursions will succeed if the nonrecursive portions of the plans succeed. This facet of the problem spaces makes learning from nonrecursive failure proofs particularly potent.

Nonrecursively Serializable Subgoals A set of subgoals is said to be *serializable* if there exists an ordering of the subgoals such that, if the subgoals are achieved in that order, once a subgoal is satisfied it need never be violated in order to achieve the remaining subgoals [23, page 39]. A problem is said to be serializable when the subgoals that comprise the problem's goal conjunction are serializable. Since the appropriate goal ordering may depend on the initial state, serializing different problems may require distinct goal orderings. PRODIGY's goal ordering control rules enable it to serialize its goals on a variety of different problems. A problem is said to be *nonrecursively serializable* if control rules that serialize its goals can be learned from nonrecursive proofs of goal interactions. Most of the problems in Minton's experiments are nonrecursively serializable, which enhances the coverage and reduces the match cost of PRODIGY/EBL's goal-ordering rules.

Using the terms introduced above, I now consider each of PRODIGY/EBL's benchmark problem spaces in turn.

6.1.1 The Blocksworld

PRODIGY's Blocksworld is a standard encoding of the Blocksworld problem space based on the encoding in [41]. The Blocksworld exhibits short nonrecursive failure proofs, and

fortuitous recursions for each subgoal in the space. As a result, PRODIGY/EBL is able to form low-cost operator and bindings choice rules that obviate backtracking in achieving individual subgoals. Thus, after PRODIGY/EBL learns from the appropriate examples, PRODIGY can achieve any individual subgoal without search on every possible problem. Search is still required to find the appropriate subgoal ordering.

There are five Blocksworld goal predicates. Most Blocksworld problems can be solved without backtracking if the subgoals are achieved in the following order:

1. Achieve any `on-table` subgoals.
2. Achieve any `clear` subgoals.
3. Build any desired towers from the bottom up (`on` goals).
4. Achieve `holding` or `arm-empty`, if necessary.

PRODIGY/EBL is able to approximate this goal ordering strategy by learning goal-ordering control rules from nonrecursive proofs of goal clobbering. As Minton points out, however, PRODIGY/EBL is unable to learn the rule "build towers from the bottom up" in its full generality, because the rule requires knowing which blocks are constrained to be below each other in the goal, and PRODIGY/EBL can only determine this using recursive proofs whose generality is limited [32, page 385]. Thus, not all Blocksworld problems are nonrecursively serializable.

As a result, PRODIGY/EBL's ability to serialize PRODIGY's goals in the Blocksworld depends on PRODIGY's problem distribution. PRODIGY/EBL will be effective when the problems encountered by PRODIGY are covered by a small number of tower-height-specific rules, or when the problems are nonrecursively serializable. In fact, most of the problems in Minton's experiment were nonrecursively serializable. Thus, PRODIGY/EBL's ability to avoid goal clobbering in the experiment was due both to the Blocksworld's structure and to Minton's problem distribution.

The features of the Blocksworld discussed above enabled PRODIGY+EBL[13] to solve Minton's Blocksworld test problems with little to no backtracking. Without backtracking PRODIGY expands $2L+2$ nodes to produce a solution of length L in each problem. The total length of PRODIGY+EBL 's solutions is 650 steps, and the total number of problems is 100. Thus, the minimal number of nodes PRODIGY+EBL could expand to produce its solutions is $(2 * 650 + 2 * 100)$ which is 1500. In fact, PRODIGY+EBL expanded 1689 nodes which is only 1.13 times the minimal node count. In contrast, PRODIGY's minimal node count is 1400 (since it found shorter solutions to some problems) but it expanded 217,948 which is 155.67 times its minimal node count.

6.1.2 The Stripsworld

PRODIGY's Stripsworld is an extended version of the STRIPS robot-planning problem space [16]. The Stripsworld also exhibits nonrecursive failure and goal-clobbering proofs.

[13] Recall that PRODIGY+EBL denotes PRODIGY guided by PRODIGY/EBL's rules.

Thus, PRODIGY's primary challenge in the Stripsworld is finding its way through each problem's room configuration. Since PRODIGY has no "sense of direction," it finds room configurations difficult to navigate. Furthermore, since the doors to different rooms may be locked, necessitating the retrieval of keys from rooms whose doors may themselves be locked and so on, navigation can be quite difficult in general.

PRODIGY's plans to move from one room to another invariably contain recursive calls to the operator GO-THRU-DR. As a result, its proofs of both successful and failed paths are recursive and thus path-length-specific. Potentially, therefore, the Stripsworld could present a severe challenge for PRODIGY/EBL. In Minton's experiments, PRODIGY/EBL is aided considerably by the fact that each of the problems takes place in one of three simple room configurations (See [30, page 182]). As a result, the rules PRODIGY/EBL learns from recursive proofs, which enable it to look ahead along certain paths, turn out to be reasonably compact and frequently applicable. Although expensive to match, the rules are useful since PRODIGY only encounters one of three room configurations.

6.1.3 The Schedworld

PRODIGY's Schedworld is a simplified process planning and machine-shop scheduling problem space. PRODIGY/EBL speeds up PRODIGY in the Schedworld in two ways: by quickly detecting unsolvable problems and by ordering top-level goals to avoid goal interactions. While PRODIGY+EBL is almost four times faster than PRODIGY on Minton's test-problem set, it is only 1.6 times faster when the top-level node rejection rules (which detect unsolvable problems) and goal ordering rules are removed from the PRODIGY/EBL's rule set. The difference in the number of nodes expanded is more striking. PRODIGY+EBL expands only 5,401 nodes compared with PRODIGY's 181,938 nodes in the Schedworld. With the node rejection and goal ordering rules removed PRODIGY+EBL expands 110,611 nodes, demonstrating the value of the removed rules.

Virtually all of the removed rules can be derived via nonrecursive proofs of failure or goal interaction. Consider, for example, the goal of painting an object. Only two operators are available for painting an object: SPRAY-PAINT and IMMERSION-PAINT. An object cannot be painted if PRODIGY does not `have-paint-for-immersion` and the required paint is not `sprayable`. This observation suffices to prove that any problem that matches this description is unsolvable. A graphical depiction of this nonrecursive proof of failure appears in Figure 8.

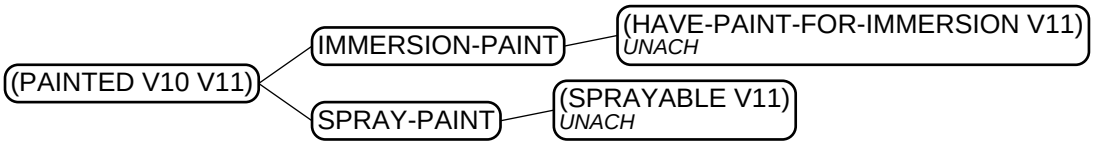

Figure 8: The PSG subgraph corresponding to the proof that an object cannot be `painted`, using a certain paint, when neither `sprayable` nor `have-paint-for-immersion` hold for that paint.

Remarkably, scheduling conflicts rarely arise in Minton's problems. Thus, learning about

such conflicts, which would require analyzing recursive interactions, was not required for success in Minton's experiment.

	Blocksworld	**Stripsworld**	**Schedworld**
Nonrecursive failures:	All subgoals	Most	All
Fortuitous recursions:	All operators	Few	Most
Additional recursions:	Goal interactions	Room configs.	Scheduling conflicts
Nonrec. Serializable:	Most goals	Some	Almost all

Table 4: A summary of the structural explanation of PRODIGY/EBL's success in Minton's experiments. Note that in all problem spaces the "additional recursions," which are potentially intractable for PRODIGY/EBL, appeared very infrequently due to the problem distributions used.

6.2 The Power of Static PSG Analysis

The STATIC program is a natural product of the structural theory. STATIC learns by analyzing PSG subgraphs that correspond to nonrecursive proofs; it does not construct logical proofs, analyze training examples, or even utilize an explicit domain theory. Instead, STATIC symbolically back-chains on PRODIGY's operator schemas to construct the PSG representation of a problem space. STATIC traverses its PSGs, annotating each node with a label indicating which operators and subgoals will succeed or fail, and a logical expression indicating under what conditions the label holds. Finally, STATIC derives control rules based on this annotation. See [11] for a complete description of STATIC's algorithms.

The previous section suggested that, on PRODIGY/EBL's benchmark tasks, effective control knowledge can be derived from nonrecursive proofs. The experiment described below supports this claim by comparing the impact of STATIC, PRODIGY/EBL, and control rules written by human experts on PRODIGY/EBL's benchmark tasks.

Experimental Methodology In each problem space PRODIGY was run on one hundred randomly generated problems. PRODIGY was run under three experimental conditions: guided by STATIC's rules, guided by PRODIGY/EBL's rules, and guided by control rules written by human experts. PRODIGY/EBL's rules, the "human control rules," and the randomly generated problem sets, were taken from Minton's experiments.[14]

The experiments were run in Allegro Common Lisp on a SUN-4 workstation. PRODIGY's problem-solving time, on each problem was bounded by 150 CPU seconds. A time bound is necessary for the experiments to complete in reasonable time. This time bound is less

[14]The PRODIGY system, control rules, benchmark problem spaces, and problem sets are publicly available by sending mail to `prodigy@cs.cmu.edu`.

restrictive than Minton's, and the machine used is faster. Thus, relative to Minton's experiments, more problems were solvable within the time bound, more nodes were expanded within the time bound, the average time per node was smaller, and the problem-solving time per problem was smaller. This change does not influence the experimental results because all the comparisons made below are between data sets obtained on the same machine, under the same time bound.

Data Presentation Figure 9 shows the total time (in CPU seconds) spent tackling Minton's test problem set (Y-axis) graphed against a bound on CPU time (X-axis). The Y-coordinate of a point on a curve in Figure 9 represents the amount of time spent by a system summed over all the problems, given the time bound in the X-coordinate. When all the problems are solved, the total problem-solving time (Y-axis) does not change as the time bound increases (X-axis). Thus, when all the problems are solved the curve is horizontal and its slope is zero.

The departure from Minton's cumulative graphs format is motivated by Segre *et al.*'s argument that, when some problems are unsolved within the CPU time bound, changing the bound can influence the graphical relationship between systems being compared using the cumulative graphs format [45]. The graphs used here are specifically designed to address this problem, showing how the relative performance of the systems scales on larger and larger CPU time bounds. Segre *et al.* also argue that the experimenter's choice of time bound can bias the results of the experiment. To address this problem, Etzioni and Etzioni [13] develop statistical hypothesis tests designed to analyze speedup learning data that is "truncated" due to the use of time bounds. The tests show that the differences between PRODIGY+STATIC and PRODIGY+EBL are statistically significant.

The graphs in Figure 9 show that PRODIGY+STATIC is faster than PRODIGY+EBL in each of PRODIGY/EBL's problem spaces. Problem-solving time is a function of the match cost and the number of nodes pruned by the learned control rules. How does the number of nodes expanded by PRODIGY+EBL compare with the number of expanded by PRODIGY+STATIC? Figure 10 graphs the total number of nodes expanded on all problems (Y-axis) against an increasing bound on the number of nodes expanded in each problem (X-axis).[15] The node-bound graphs in figure 10 show that STATIC's ability to curtail search, measured by the number of nodes pruned, rivals PRODIGY/EBL's in each of PRODIGY/EBL's problem spaces.

6.3 PRODIGY/EBL is Foiled in the ABworld

Section 1.1 showed how a seemingly minute modification to the Blocksworld, creating the ABworld, foiled PRODIGY/EBL. This section explains this observation using the structural theory. Before reporting in detail on the experiment, I explain how the ABworld was constructed. The presence and location of recursion is sensitive to problem space encoding. A problem space can be "robbed" of nonrecursive proofs of failure by inserting operators. Since *all* operators have to fail if a given subgoal is to fail, inserting an operator means that

[15] The maximal node bound chosen was 120 nodes per problem.

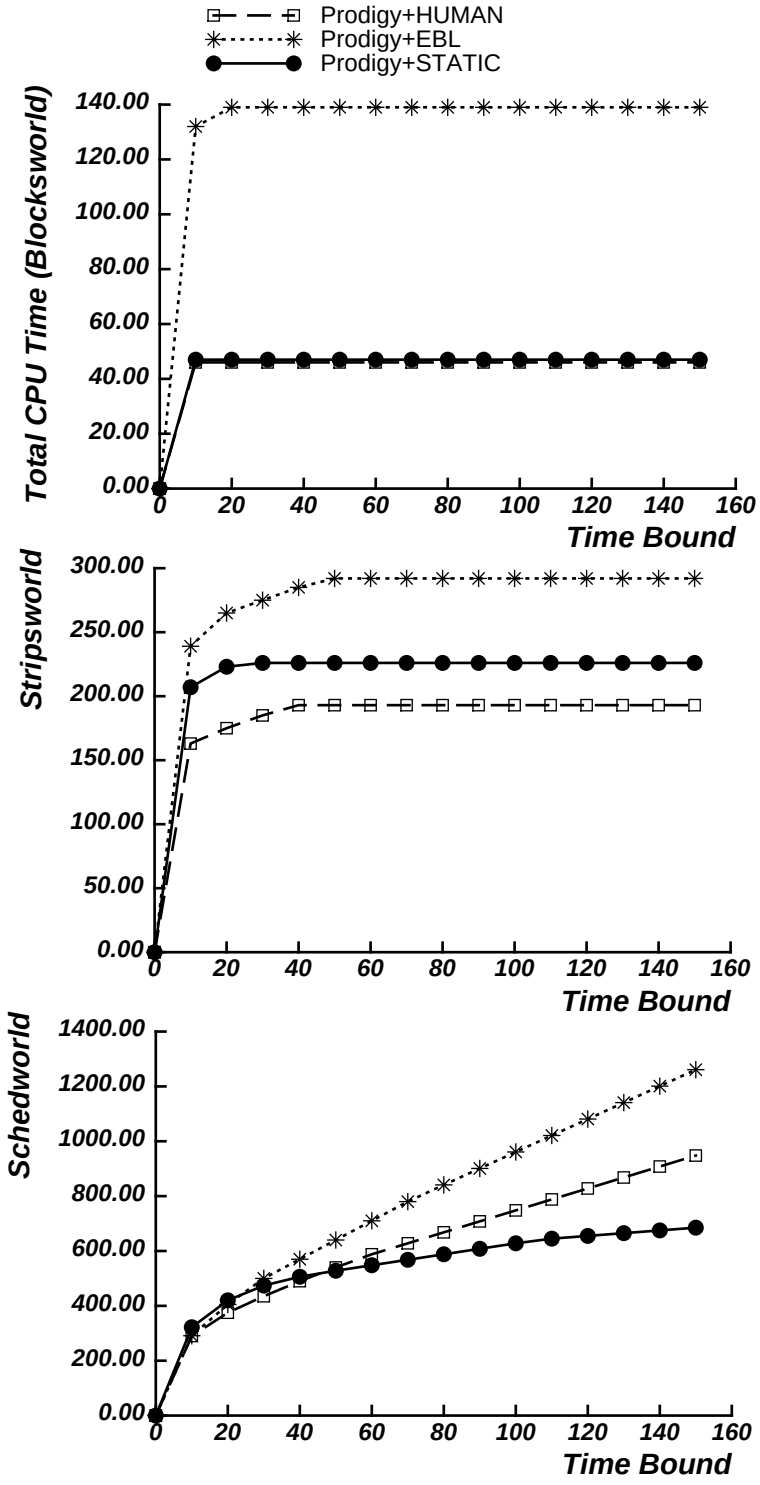

Figure 9: Total problem-solving time in PRODIGY/EBL's problem spaces.

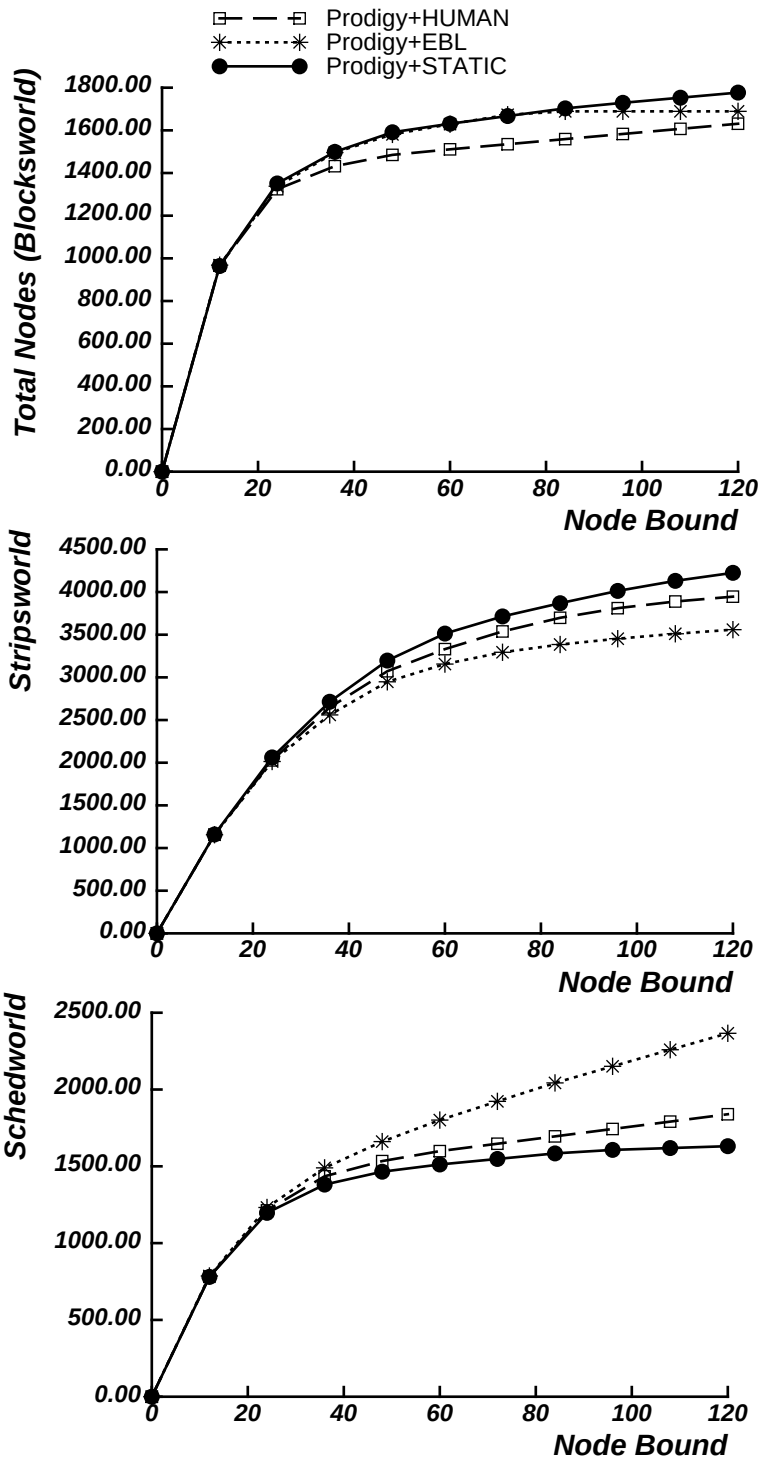

Figure 10: Total number of nodes expanded in PRODIGY/EBL's problem spaces.

an AND-node is necessarily added to any proof that the subgoal fails. When the inserted operator results in a recursion, its addition transforms a nonrecursive proof into a recursive one.

In the Blocksworld, the failure of any operator can be explained by a nonrecursive proof. When explaining the success of an operator is recursive, explaining the failure of its sibling operators is not. In the ABworld, 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 V)` when the block is on the table. This path is doomed to failure in both problem spaces. In the Blocksworld, the failure can be explained nonrecursively since trying to achieve `on` leads to a goal cycle immediately (Figure 7, Section 4.2).

In the ABworld, in contrast, `on` can be achieved by the GRASP-SECOND-BLOCK macro-operator. 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 (see Figure 2, Section 1.1). Thus, PRODIGY has another means of achieving `on` which does not immediately cause a goal 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 ABworld is recursive. The PSG representation of (a fragment of) this proof appears in Figure 11. Contrasting the fragment with Figure 7 shows how adding the GRASP-SECOND-BLOCK macro-operator to the Blocksworld makes explaining the failure UNSTACK recursive.

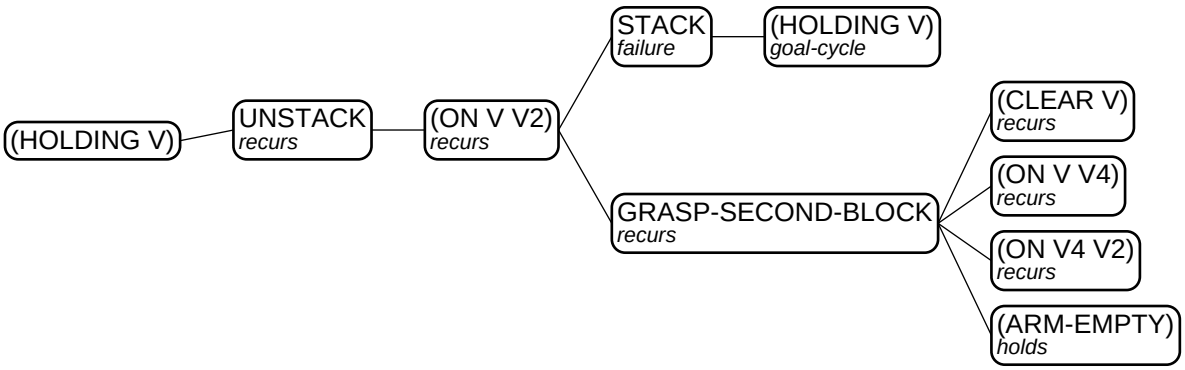

Figure 11: A fragment of the PSG subgraph corresponding to the proof that UNSTACK fails to achieve `holding` in the ABworld. For brevity, the recursive subgraph below the node `(clear V)` is not shown. The subgraph leads the node `(clear V)` to be labeled *recurs*.

The above discussion explains how adding a macro-operator to the Blocksworld led to PRODIGY/EBL's lackluster performance in the ABworld. In essence, the ABworld robs

PRODIGY/EBL of key nonrecursive proofs, forcing it to analyze a complex and unwieldy recursion. Much of the work on recursive proofs (e.g., the work on generalization-to-N discussed in Section 3.3.2) has focused on simple tail recursions of the sort found when clearing a block in the Blocksworld. The ABworld recursions are far more complex. For example, in the ABworld PRODIGY expands over eight-thousand nodes in the process of trying to grasp the block at the bottom of a three-block tower. In order to learn, PRODIGY/EBL is forced to analyze the resulting problem-solving trace with dire consequences.

Several additional aspects of the experiment are worth noting. First, not all ABworld proofs are recursive as evidenced by the fact that STATIC was able to learn several control rules in this space. Second, the problem distribution used is *highly* skewed in that no problems with more than five blocks or goal conjuncts were used. Third, all of the Blocksworld predicates are unique attributes so that the cost of matching PRODIGY/EBL's control rules scales linearly with the state size. Finally, PRODIGY/EBL's impact was significantly improved by its use of utility evaluation.[16] Despite all these factors (which aid EBL) PRODIGY/EBL still failed to be effective in the ABworld demonstrating the difficulty of analyzing complex recursions, even in relatively simple Blocksworld problems.

Experimental Methodology In the ABworld PRODIGY was tested on the first 30 problems of Minton's Blocksworld test-problem set. The massive searches required to solve the larger problems in Minton's test set exhausted the machine's memory. Figure 3, in Section 1.1, shows that PRODIGY/EBL was foiled in this problem space. Comparing Figure 3 with Figure 1, which shows PRODIGY/EBL's impact in the Blocksworld, demonstrates that PRODIGY/EBL's impact in the ABworld did indeed degrade significantly relative to the Blocksworld.

6.4 Modified Problem Distributions

Section 6.1 argues that PRODIGY/EBL is effective in the Blocksworld due to the nonrecursive proofs found by PRODIGY/EBL. An important advantage of learning from nonrecursive proofs is the ability to acquire control rules that are effective across changes in problem distribution. I illustrate this point by showing how the rule set learned by PRODIGY/EBL in the Blocksworld remains effective across two problem distributions that are very different from Minton's original distribution.

Experimental Methodology A Blocksworld problem consists of an initial state and a goal conjunction. A distribution of Blocksworld problems is defined by a problem generator which determines the initial state and the goal. Blocksworld states can be characterized by the number of blocks, the height and number of towers, and whether the robot is holding a block. Blocksworld goal conjunctions can be characterized by the length of the conjunctions and their composition. I drastically modified the distribution defined by Minton's problem

[16] The results reported in this experiment and in all others are for PRODIGY/EBL using utility evaluation.

generator, randomly generating fifty problems using the new distribution, and comparing PRODIGY+EBL problem-solving time with PRODIGY's on the randomly generated problems.

The number of blocks in Minton's problem set ranged from three to twelve, and the number of goal conjuncts ranged from one to ten. In the first experiment, the number of blocks was increased to twenty and number of goal conjuncts was uniformly ten. The results of the experiment on this problem distribution appear in Figure 12. Prodigy, running with no control rules, was unable to solve *any* of the large Blocksworld problems within the time bound (150 CPU seconds). Hence, the curve representing PRODIGY's problem-solving time appears linear, and the difference between PRODIGY+EBL's problem-solving time and PRODIGY's is understated. PRODIGY+EBL was able to solve all of the large Blocksworld problems within the time bound.

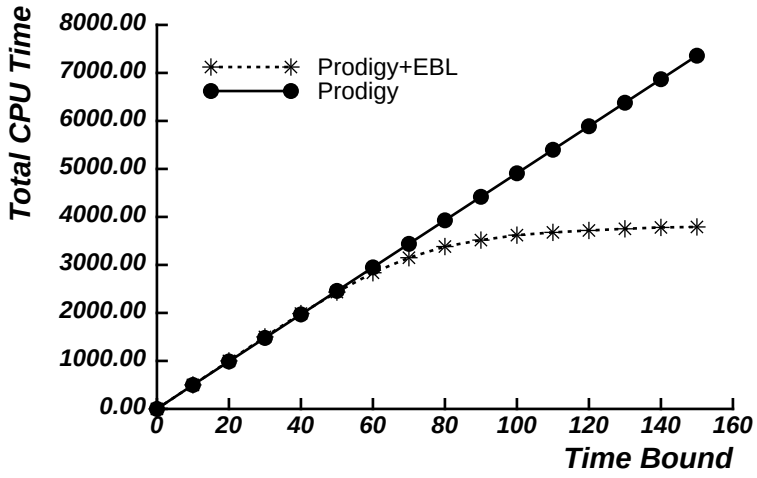

Figure 12: PRODIGY/EBL's impact on large Blocksworld problems. Due to the size of the problems PRODIGY was unable to solve *any* of the problems within the time bound.

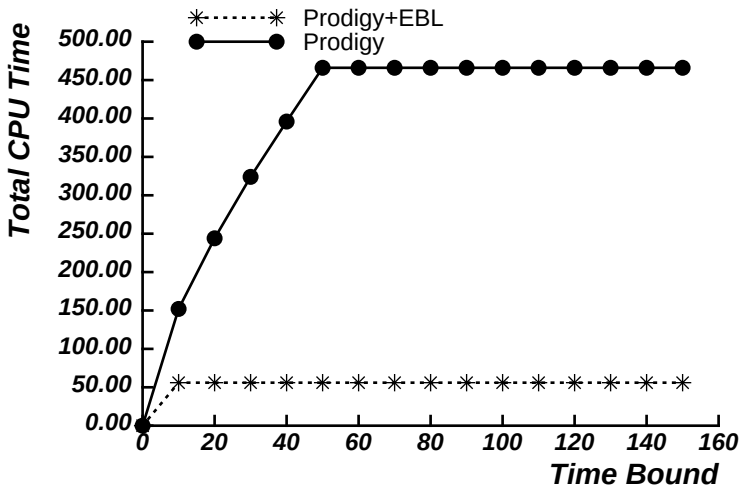

Figure 13: PRODIGY/EBL's impact on a modified Blocksworld problem distribution.

Minton reports on three parameters used to determine the nature of his Blocksworld problem set: the probability that a block was being held in the initial state, that a block was on the table (as opposed to on another block) in the initial state, and the probability that a goal conjunct was in fact satisfied in the initial state. To modify the problem distribution in the second experiment, the three parameters were changed from 1/3 to 2/3, from 1/3 to 1/2, and from 1/3 to 1/6. The intended impact of the change was to make towers in the initial state shorter and to increase problem difficulty by requiring more goal conjuncts to be actively achieved. The results of the experiment on this problem distribution appear in Figure 13. In both cases, as in Minton's experiments, PRODIGY+EBL remained significantly faster than PRODIGY.

6.5 Complementary Target Concepts are Essential

We might hypothesize that, although PRODIGY/EBL relies strongly on the failure and goal interaction target concepts in Minton's experiments, it might be able to compensate for their absence by relying more heavily on learning from success. This hypothesis is important because many EBL (and partial evaluation) systems rely exclusively on learning from success. If the hypothesis were true, these systems would not need to be extended to cover a wider range of target concepts. The structural theory suggests that this hypothesis is false due to the arguments in Section 5.3. Briefly, learning only from success in recursive problem spaces will yield recursive proofs which, by the recursive heuristic, are unlikely to be useful. In addition, once the problem solver has diverged from a solution path, the only control rules that can curtail backtracking are control rules learned from complementary target concepts such as failure and goal interaction.

To test the hypothesis PRODIGY/EBL was run, using the training procedure in Minton's thesis, with its failure and goal interaction target concepts "turned off." PRODIGY/EBL only learned from success. I refer to PRODIGY guided only by learning from success as PRODIGY+SUCCESS. Figure 14 contrasts PRODIGY+SUCCESS with PRODIGY+EBL and with PRODIGY in PRODIGY/EBL's problem spaces. In all cases, PRODIGY+SUCCESS was close to PRODIGY, and much slower than PRODIGY+EBL. This observation invalidates the hypothesis suggested above; the experimental results support the structural theory which argues that complementary target concepts enhance the effectiveness of EBL.

This section relied on the structural theory to account for PRODIGY/EBL's success in Minton's experiments and to informally explain several additional experimental results originally reported in [10]. The experiments, and their analysis, provide a degree of confirmation for the structural theory.

7 Related Work

A number of attempts have been made to analyze EBL's performance [6, 21, 28, 30, 44, 50]. Most of these analyses are very different from the analysis in this paper, and are discussed

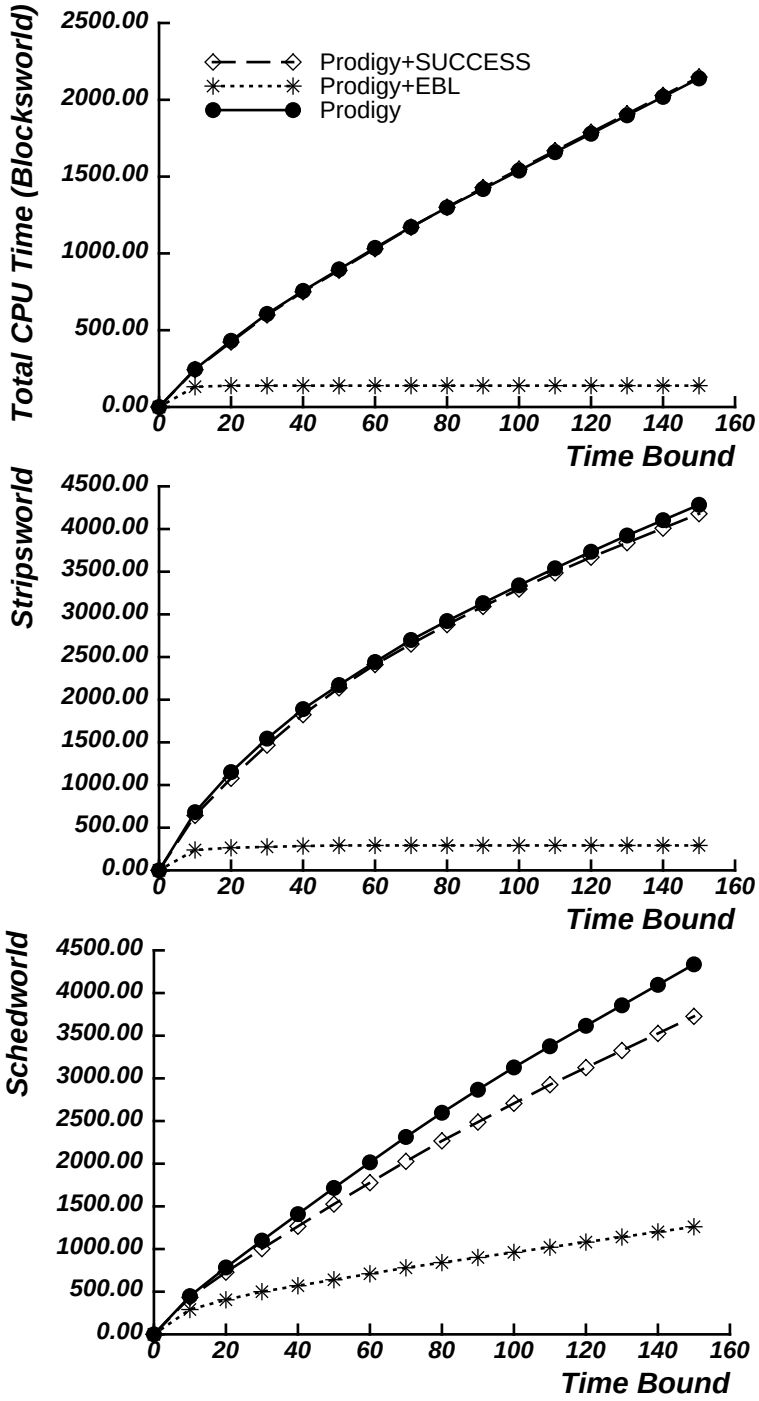

Figure 14: PRODIGY/EBL's performance when learning only from success.

only briefly below. One analysis [52, 53] independently reaches a conclusion that is closely related to the recursive heuristic, and is discussed at greater length. No analysis of EBL has derived conclusions akin to the nonrecursive heuristic or the structural thesis.

Mahadevan, Natarajan, and Tadepalli [28] present a formal model of learning as improving problem-solving performance based on the framework in [39]. Based on the model, Tadepalli [55] argues that EBL relies on structural constraints (e.g. serial decomposability) and that, like inductive learning, EBL can be analyzed using Valiant's PAC framework [59]. Although Tadepalli's model is different from my own, the spirit of the two approaches is similar: we both seek to identify classes of problem spaces in which EBL results in tractable problem solving. Cohen [6] describes a solution-path caching mechanism that operates in polynomial time and provably improves performance (see also [5]). Both Cohen and Tadepalli make useful connections between Valiant's framework and the learning of search control knowledge.

Minton's thesis contains a thorough analysis of PRODIGY/EBL. Minton's focus, however, is on the impact of PRODIGY/EBL's components on its performance. His thesis does consider how problem space characteristics influence the efficacy of learning, but the discussion is, by and large, qualitative and informal. For example, Minton points out that a problem space's "solution density and distribution" influences the utility of different learning strategies. Minton himself concedes that "we have not adequately characterized the types of domains for which the learning method produces good results." [32].

Greiner and Likuski [21] model EBL as adding redundant rules to a set of inference rules. They report that, in general, finding the optimal inference strategy in a redundant search space is NP-hard. However, if a single EBL rule is added to a nonconjunctive, nonrecursive, and irredundant rule set, then a simple extension to Smith's algorithm [51] can still yield an optimal inference strategy in linear time. Most of the issues considered in this paper (e.g. recursion and complementary target concepts) do not arise in Greiner and Likuski's model.

Subramanian and Feldman [52, 53] extend Greiner and Likuski's model to cover recursive and conjunctive rule sets. Adopting Minton's utility criterion [33] they show that, relative to an arbitrary probability distribution and theory, adding a redundant rule m can be justified exactly when

$$p_m > C_{mf}/(C_{mf} + C_d - C_{ms})$$

where p_m is the probability that m applies, C_{mf} is the average cost of determining that m does not apply, C_d is the average cost of answering queries using the theory, and C_{ms} is the average cost of using m to answer a query. Subramanian and Feldman argue that estimating the various costs and probabilities required to apply their inequality can be done by sampling (in a manner similar to PRODIGY/EBL's utility evaluation module, perhaps). They go on to show that "unless we make very strong assumptions about the nature of the distribution of future problems, it is not profitable to form recursive macro-rules via EBL."

Thus, based on an entirely different model, Subramanian and Feldman reach a conclusion similar to the recursive heuristic. Although a simple complexity analysis suffices to argue against learning from recursive proofs via EBL (the overhead of utilizing rules learned from such proofs is exponential in the depth of the recursions encountered and the rules are

recursion-depth-specific), they rely on a more elaborate cost model of horn-clause theorem proving. This detailed model is motivated by their broader project which seeks to "quantitatively estimate" the cost of inference. The formal analysis in their paper is supported by experiments using Shavlik's [47] circuit synthesis domain theory, providing further confirmation of the value of the recursive heuristic. Subramanian and Feldman did not reach conclusions akin to the nonrecursive heuristic, or the structural thesis.

8 Concluding Remarks

I conclude with a critique of the structural theory and a discussion of directions for future work.

8.1 Critique of the Structural Theory

Although the structural theory has been tested extensively on several problem spaces using the PRODIGY system, additional experiments using different problem solvers, problem spaces, and target concepts are necessary to further test and refine the theory. Several directions for investigation are particularly intriguing:

- Can we substitute "short" for "nonrecursive" and "long" for "recursive" in the structural theory?[17] Although short rules are guaranteed to have low-match cost per node, Proposition 5 (Section 2.3) shows that short rules will not necessarily yield average speedup. In fact, STATIC's nonrecursive proofs resulted in control rules of widely-varying lengths demonstrating that the terms "short" and "nonrecursive" are not co-extensive. Furthermore, as explained in Section 5.1.1, the "short heuristic" would lead EBL to learn from bounded-depth recursive proofs forbidden by the recursive heuristic.

  While the long/short dichotomy is clearly distinct from the recursive/nonrecursive one, it is worth investigating the performance of an EBL system restricted to acquiring only short rules. To study this issue, the terms "long" and "short" would have to be defined precisely. Does a "short" proof have the same length in all problem spaces? Or is "short" a function of problem-space parameters such as the number of operators and predicates?

- Recall that complementary target concepts are concepts, such as failure in PRODIGY/EBL, whose proofs are not direct translations of the problem solver's plans. The structural theory advocates learning from complementary target concepts in order to obtain nonrecursive explanations of the problem solver's behavior. In PRODIGY/EBL's benchmark problem spaces, all but one of STATIC's rules were learned from complementary target concepts. Is it possible that learning from complementary target concepts, rather than learning from nonrecursive proofs, is actually the key to the success of EBL? To test this conjecture we need to identify tasks where

[17] This insightful question was posed by Paul Rosenbloom and Prasad Tadepalli.

complementary target concepts yield recursive proofs, and compare the effectiveness of EBL, learning from complementary concepts, to its effectiveness learning only from nonrecursive proofs.

- Problem space Graphs (PSGs) are defined for backward-chaining problem solvers. It is not entirely clear what the corresponding notion is for forward-chaining problem solvers. Consequently, determining the precise scope of the structural thesis remains an open question.

- Can the recursive heuristic be refined to exclude cases in which EBL *will* succeed when learning from recursive proofs? When will EBL be effective on problem spaces with bounded recursive depths such as the Eightpuzzle or the three-disk Tower of Hanoi? How will EBL fare when generalization-to-N methods (seeking broad coverage for EBL's knowledge) are applied to unique-attribute problem space (ensuring low local match cost for EBL's weakest preconditions)?

8.2 Task Encoding and EBL

In some cases EBL is effective and in others it is not. The structural theory explains this observation in two steps. First, the theory exhibits a correspondence between EBL's proofs and the PSG, a graph representation of problem spaces. Second, the theory shows how certain subgraphs (corresponding to nonrecursive proofs) aid EBL whereas other subgraphs (corresponding to recursive proofs) hinder EBL. Thus, structure of the PSG is a major factor influencing the effectiveness of EBL.

The PSG representation changes with the problem space encoding. In practice, problem space encoding is often modified and tweaked repeatedly until EBL generates effective control knowledge. As Letovsky [27] put it: "It is not the case that EBG systems can take any, or even most, inefficient encodings of a task and turn out efficient versions; rather it is *sometimes* the case that there exists *some* encoding of a task for which an EBG system can do something reasonable." Today, successfully applying EBL is something of a black art. If EBL is to be more broadly useful, this art needs to mature into a well-understood body of knowledge. Given a task, the ideal theory would recommend an appropriate problem space encoding, problem solving method, and "brand" of EBL (Which target concepts? What operationality criterion? Macros versus control rules? etc.). The structural theory of EBL is only a first step in this direction. The theory does, however, provide an account of EBL's performance that is based on problem-space characteristics, providing a problem-space designer with a principled basis for making representational choices.

Whether a problem space yields recursive or nonrecursive proofs depends, in part, on the problem space's PSGs. Since minor modifications to the problem space's representation can result in major changes to its PSG, the problem-space designer has to understand the impact of different representational choices on the PSG. For example, reordering operator preconditions has no appreciable impact on the PSG, suggesting that this transformation will not aid EBL. Changing problem-space predicates, in contrast, can rid the problem space

of troublesome recursions. In the Blocksworld, for example, deciding whether one block is above another block requires a recursive computation. If the Cartesian coordinates of each block are specified, however, determining whether one block is above another merely requires comparing the X-coordinates of the blocks, ridding the PSG of a recursion. Thus, the theory suggests a general methodology for understanding the impact of problem space encoding on EBL: catalog the impact of representational transformations on the location and presence of recursion in PSGs.

Acknowledgments

This paper is based on my Ph.D. dissertation at Carnegie Mellon University, which was supported by an AT&T Bell Labs Ph.D. Fellowship. Thanks are due to my advisor, Tom Mitchell, to the members of my committee, Jaime Carbonell, Paul Rosenbloom, and Kurt Vanlehn, and to Allen Newell for their impact on the dissertation. Special thanks are due to Steve Minton—whose thesis work made studying PRODIGY/EBL possible. Steve Hanks, Craig Knoblock, Phil Laird, Neal Lesh, Alicia Pérez, Prasad Tadepalli, Dan Weld and the anonymous reviewers provided helpful comments on earlier drafts.

References

[1] Neeraj Bhatnagar and Jack Mostow. Adaptive search by explanation-based learning of heuristic censors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

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

[3] Henrik Bostrom. Generalizing the order of goals as an approach to generalizing number. In *Proceedings of the Seventh International Conference on Machine Learning*. Morgan Kaufmann, 1990.

[4] 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.

[5] William W. Cohen. *Concept Learning Using Explanation Based Generalization as an Abstraction Mechanism*. PhD thesis, Rutgers University, 1990.

[6] William W. Cohen. Using distribution-free learning theory to analyze chunking. In *Proceedings of the Eighth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*. Morgan Kaufmann, 1990.

[7] R. Davis. Meta-rules: reasoning about control. *AI*, 15:179–222, 1980.

[8] G. F. Dejong and R. J. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(1), 1986.

[9] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.

[10] Oren Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, Carnegie Mellon University, 1990. Available as technical report CMU-CS-90-185.

[11] Oren Etzioni. Acquiring search-control knowledge via static analysis. Technical Report 92-04-01, University of Washington, 1992.

[12] Oren Etzioni. An asymptotic analysis of speedup learning. In *Proceedings of the Ninth International Conference on Machine Learning.* Morgan Kaufmann, 1992.

[13] Oren Etzioni and Ruth Etzioni. Statistical methods for analyzing speedup learning experiments. Submitted for publication., 1992.

[14] Oren Etzioni and Steven Minton. Why EBL produces overly-specific knowledge: A critique of the PRODIGY approaches. In *Proceedings of the Ninth International Conference on Machine Learning.* Morgan Kaufmann, 1992.

[15] Oren Etzioni and Tom M. Mitchell. A comparative analysis of chunking and decision-analytic control. In *The Soar Papers: Research on Integrated Intelligence.* MIT Press, Cambridge, MA, 1992. To Appear.

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

[17] Michael R. Garey and David S. Johnson. *Computers And Intractability A guide to the Theory of NP-Completeness.* Freeman, New York, NY, 1979.

[18] M.R. Genesereth. An overview of meta-level architecture. In *Proc. AAAI*, Washington, D.C., August 1983. AAAI.

[19] Jonathan Gratch and Gerald Dejong. Composer: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of AAAI-92*, 1992.

[20] Russell Greiner. A solution to the ebl utility problem. In *Proceedings of AAAI-92*, 1992.

[21] 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.

[22] Kevin Knight. Unification: A multidisciplinary survey. *Computing Surveys*, 21(1), March 1989.

[23] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

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

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

[26] J. E. Laird, P. S. Rosenbloom, and Newell A. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

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

[28] 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.

[29] Steven Minton. EBL and weakest preconditions. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*, pages 210–214, 1988.

[30] 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.

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

[32] Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3), March 1990.

[33] Steven Minton, Jaime G. Carbonell, Oren Etzioni, Craig A. Knoblock, and Daniel R. Kuokka. Acquiring effective search control rules: Explanation-based learning in the Prodigy system. In *Proceedings of the Fourth International Workshop on Machine Learning*, 1987.

[34] 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 technical report CMU-CS-89-103.

[35] Steven Minton, Craig A. Knoblock, Daniel R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. Prodigy 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, Carnegie Mellon University, 1989.

[36] Tom M. Mitchell. *Version Spaces: An Approach to Concept Learning.* PhD thesis, Stanford University, December 1978. also Stanford CS report STAN-CS-78-711, HPP-79-2.

[37] Tom M. Mitchell, John Allen, Prasad Chalasani, John Cheng, Oren Etzioni, Marc Ringuette, and Jeffrey C. Schlimmer. Theo: A framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence.* Erlbaum, 1991.

[38] Tom M. Mitchell, Rich Keller, and Smadar Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[39] B. K. Natarajan and Prasad Tadepalli. Two new frameworks for learning. In *Proceedings of the Fifth International Conference on Machine Learning*, 1988.

[40] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice Hall, 1972.

[41] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing, 1980.

[42] M. Alicia Pérez and Oren Etzioni. DYNAMIC: a new role for training problems in EBL. In *Proceedings of the Ninth International Conference on Machine Learning.* Morgan Kaufmann, 1992. An expanded version available as technical report CMU-CS-92-124.

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

[44] Paul Resnick. Generalizing on multiple grounds: Performance learning in model-based troubleshooting. Master's thesis, M.I.T., 1988. Available as AI Lab Technical Report 1052.

[45] Alberto Segre, Charles Elkan, and Alex Russell. A critical look at experimental evaluations of EBL. *Machine Learning*, 1991. forthcoming methodological note.

[46] Peter Sestfot. Automatic call unfolding in a partial evaluator. In D. Bjorner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation.* Elsevier Science Publishers, 1988. Workshop Proceedings.

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

[48] 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.

[49] 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.

[50] Jeff Shrager, Tad Hogg, and Bernardo A. Huberman. A graph-dynamic model of the power law of practice and the problem-solving fan-effect. *Science*, 242:414–416, 1988.

[51] David E. Smith. *Controlling Inference*. PhD thesis, Stanford, 1986. Available as technical report STAN-CS-86-1107.

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

[53] Devika Subramanian and Ronen Feldman. The utility of EBL in recursive domain theories. An extended version of the AAAI 1990 paper., 1991.

[54] Devika Subramanian and Scott Hunter. Measuring utility and the design of provably good ebl algorithms. In *Working Notes of the AAAI Spring Symposium on Knowledge Assimilation*, Menlo Park, CA, 1992. AAAI Press.

[55] Prasad Tadepalli. A formalization of explanation-based macro-operator learning. In *Proceedings of the Twelveth International Joint Conference on Artificial Intelligence*, 1991.

[56] M. Tambe, Newell A., and P. Rosenbloom. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning Journal*, 5(3):299–348, 1990.

[57] Milind Tambe and Paul Rosenbloom. Eliminating expensive chunks by restricting expressiveness. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[58] Jeffrey D. Ullman. *Database and Knowledge-base systems*, volume II. Computer Science Press, 1989.

[59] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11), 1984.

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