DYNAMIC: A new role for training problems in EBL [1]

M. Alicia Pérez and Oren Etzioni [2]

March 1992

CMU-CS-92-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in *Machine Learning: Proceedings of the Ninth International Conference (ML92)*, D. Sleeman and P. Edwards (eds.) Morgan Kaufmann, San Mateo, CA. 1992

[1]This research was supported, in part, by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597; by a scholarship from the Ministerio de Educación y Ciencia of Spain; and by the University of Washington Graduate School Research Fund. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

[2]Second author's current address: University of Washington, Department of Computer Science and Engineering, FR-35, Seattle, WA 98195

Keywords: explanation-based learning, training examples, planning, problem solving, machine learning

Abstract

Most Explanation-Based Learning (EBL) systems construct explanations by directly translating a trace of their problem solver's search, on training problems, into proofs. This approach makes proof derivation tractable, but can focus EBL on incidental aspects of its training problems, yielding overly-specific control knowledge. Previous work has described the other extreme: STATIC, a system that generates more general control knowledge by statically analyzing problem-space definitions. However, since STATIC does not utilize training problems, it has a number of potential disadvantages compared with EBL.

This paper advocates an intermediate approach in which training problems pinpoint learning opportunities but do not determine EBL's explanations. Based on this design principle, we developed DYNAMIC, a module that learns control rules for the PRODIGY problem solver. In DYNAMIC, choosing *what* to explain and *how* to explain it are independent. DYNAMIC utilizes the analysis algorithms introduced by STATIC, but relies on training problems to achieve the distribution-sensitivity of EBL. On a highly skewed problem distribution, DYNAMIC was almost four times as effective as STATIC in speeding up PRODIGY. When tested in PRODIGY/EBL's benchmark problem spaces, DYNAMIC ran considerably faster than PRODIGY/EBL and produced control rules that were close to three times as effective. In addition, DYNAMIC required only a fraction of the training problems used by PRODIGY/EBL.

1 Motivation

Explanation-Based Learning (EBL) [Dejong and Mooney, 1986, Mitchell *et al.*, 1986] has emerged as a standard method for acquiring search-control knowledge for general problem solvers. Most EBL systems generate explanations by directly translating a trace of the problem solver's behavior, when solving a training problem, into a logical proof [Laird *et al.*, 1986, Minton *et al.*, 1989a, Mitchell *et al.*, 1991, Prieditis and Mostow, 1987, Shavlik, 1990, Subramanian and Feldman, 1990]. This approach enables EBL systems to circumvent the computational intractability of deriving proofs "from scratch."[1] Unfortunately, as a result, the problem solver's behavior on a training problem determines not only *what* EBL will explain but also *how* the explanation will be constructed, frequently yielding overly-specific control knowledge. The fundamental point of this paper is that these decisions should be made independently.

Consider, for example, avoiding goal clobbering in the Blocksworld. Suppose a means-ends analysis problem solver such as STRIPS [Fikes *et al.*, 1972] or PRODIGY [Minton *et al.*, 1989a] is given the conjunctive goal: `(and (holding c) (on a b))`. Achieving these goals, in order, results in a goal clobbering. The problem solver has to undo `(holding c)` in order to achieve `(on a b)`. To avoid such problems PRODIGY utilizes *control rules*, meta-level rules that override PRODIGY's default search strategy. For example, the control rule in Table 1 directs PRODIGY to achieve the `on` subgoal *before* achieving `holding`, thereby avoiding the goal clobbering.

PRODIGY control rules can be generated automatically by PRODIGY/EBL, PRODIGY's EBL module [Minton, 1990]. Given the training problem in Figure 1, in which the goal clobbering occurs, we would like PRODIGY/EBL to learn the rule in Table 1. Unfortunately, when this problem is presented to PRODIGY/EBL, it learns the overly-specific rule in Table 2. (Section 4 explains why). The conditions marked by `''***''` unnecessarily restrict the set of problems in which the rule will apply. In addition, because PRODIGY repeatedly matches the antecedent of each control rule during its search, this rule has a higher match cost than the rule in Table 1.

Why did PRODIGY/EBL acquire the overly-specific rule in Table 2? We answer this question and describe DYNAMIC, a program that learns the superior control rule in Table 1, when trained on the very same problem as PRODIGY/EBL. Previous work has shown how a static analyzer of problem-space definitions (called STATIC) can generate better control rules than PRODIGY/EBL, utilizing a graph representation of problem spaces, called Problem Space Graphs (PSGs) [Etzioni, 1990a, Etzioni, 1992]. Like STATIC, DYNAMIC utilizes the PSG representation and the associated algorithms. Unlike STATIC, however, DYNAMIC's analysis is focused by training problems, providing it with several advantages over STATIC (Section 6.1).

The paper is organized as follows. Section 2 provides necessary background on PRODIGY and STATIC. Section 3 introduces DYNAMIC, focusing on how it differs from STATIC. Section 4 explains why DYNAMIC derives more general rules than standard EBL. Section 5 describes experimental results on DYNAMIC's performance, and Section 6 contrasts DYNAMIC with STATIC and PRODIGY/EBL. Section 7 concludes by suggesting a new role for training problems in EBL, validated by the DYNAMIC experiments.

2 Background

This section provides background on PRODIGY and STATIC that is directly relevant to understanding DYNAMIC.

[1] However, the expense incurred by the problem solver in generating the trace may be considerable.

```
(ON-BEFORE-HOLDING
 (if (and (current-node Node)
          (candidate-goal Node (holding BlockX))
          (candidate-goal Node (on BlockY BlockZ))))
 (then (prefer goal (on BlockY BlockZ)(holding BlockX))))
```

Table 1: A control rule that enables PRODIGY to avoid goal clobbering in the Blocksworld. Variable names are capitalized.

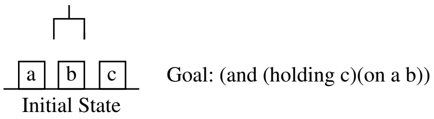

Figure 1: A Blocksworld training problem in which goal clobbering occurs.

2.1 The PRODIGY Problem Solver

A complete description of PRODIGY appears in [Minton, 1988, Minton *et al.*, 1989b]. 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. PRODIGY's sole problem-solving method is a form of means-ends analysis [Newell and Simon, 1972]. Like STRIPS [Fikes *et al.*, 1972], PRODIGY employs operator preconditions as its differences. However, PRODIGY's operator description language is considerably more expressive, allowing universal quantification and conditional effects.

PRODIGY's default search strategy is depth-first search. The search is carried out by repeating the following decision cycle [Minton, 1990]:

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

2. One of the goals at that node is chosen.

3. An operator whose postconditions unify with the goal is chosen.

4. Bindings for the variables in the operator are chosen. If the instantiated operator's preconditions match the state then it is applied to the state, otherwise PRODIGY subgoals on the operator's unmatched

```
(EBL-ON-BEFORE-HOLDING
 (if (and (current-node Node)
          (candidate-goal Node (holding BlockX))
          (candidate-goal Node (on BlockY BlockZ))
***       (known Node (on-table BlockX))
***       (known Node (on-table BlockY))))
 (then (prefer goal (on BlockY BlockZ)(holding BlockX))))
```

Table 2: A (simplified version of) the control rule learned by PRODIGY/EBL based on the problem in Table 1.

preconditions. In either case a new node is created, and the search continues from that node.

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 [Minton, 1990]. 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 1 tells PRODIGY to achieve the `on` subgoal before achieving the `holding` subgoal.

2.2 STATIC

Like PRODIGY/EBL, STATIC and DYNAMIC automatically generate control rules for PRODIGY. This section provides an overview of STATIC that is helpful in understanding DYNAMIC. STATIC's input is a problem-space definition, and its output is a set of PRODIGY control rules. STATIC does not actually construct explanations or even utilize an explicit domain theory. Instead, STATIC symbolically back-chains on PRODIGY's operator schemas to construct an AND/OR graph representation of problem-space subgoals/operators, known as a Problem Space Graph (PSG). This is an instance of the general paradigm of symbolic execution or call unfolding, which is frequently used to implement partial evaluation [Komorowski, 1982, Sestfot, 1988, van Harmelen and Bundy, 1988].[2] Essentially, the PSG represents all backward-chaining paths through the problem space (e.g., Figure 2). To keep the PSG finite, STATIC restricts its attention to nonrecursive paths, terminating PSG expansion whenever it encounters recursion. STATIC annotates each PSG 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 PRODIGY control rules based on this annotation. When tested in PRODIGY/EBL's benchmark problem spaces [Minton, 1988], STATIC generated search-control knowledge that was up to three times as effective as PRODIGY/EBL's, and did so from twenty-six to seventy-seven times faster [Etzioni, 1991].

2.2.1 The Limitations of STATIC

Because it does not utilize training problems, STATIC will not outperform EBL systems in every case. To keep STATIC's analysis tractable and to avoid generating ineffective control rules, the range of proofs utilized by STATIC is carefully restricted; STATIC only searches for pairwise goal interactions, overlooking more complex ones, and STATIC does not learn from recursive explanation.[3] Furthermore, an EBL system can exploit the distribution of problems encountered by its problem solver whereas STATIC cannot. Indeed, STATIC's running time increases with the number of uninstantiated subgoals in its problem space, whereas EBL's running time only increases with the number of subgoals *actually* encountered in training problems. Even in PRODIGY/EBL's relatively simple benchmark problem spaces, over ninety percent of STATIC's time was spent analyzing goal interactions [Etzioni, 1990a]. In general, because training problems transmit information about the problem solver's environment that EBL has access to, but STATIC does not, we would

[2]The PSG is similar to the rule/goal graph [Ullman, 1989], and related graph representations of logic programs, but is used to compute the weakest preconditions of explanations by STATIC and DYNAMIC.

[3]An explanation is said to be *recursive* when it refers to recursive subgoaling in the problem space. Explaining the success of `unstack b a` is recursive, for example, because `unstack` succeeds only if the block `b` can be cleared, and `b` can be cleared only if the block that is on `b` can be cleared, and so on. Avoiding recursive explanations is actually an advantage in many cases [Etzioni, 1990b].

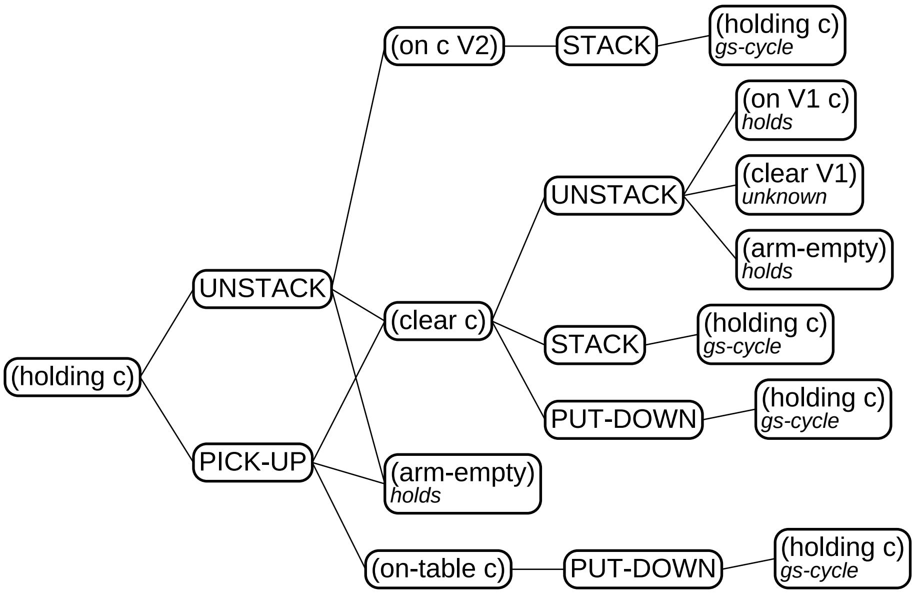

Figure 2: The `holding` PSG generated by DYNAMIC, based on the problem in Figure 1.

expect an appropriately-designed EBL system to outperform STATIC on highly-skewed problem distributions (cf. [DeJong and Gratch, 1991]). This observation motivates the design of DYNAMIC.

3 DYNAMIC

DYNAMIC utilizes both training problems and PSG analysis to learn control rules for PRODIGY. PSG-based systems like DYNAMIC or STATIC face two independent questions:

- *Which PSGs to generate?* Unlike STATIC, which generates a PSG for every achievable subgoal in the problem space, DYNAMIC chooses which PSGs to analyze based on PRODIGY's behavior on training problems. Because DYNAMIC has an additional input, compared with STATIC, it is able to tailor the rule set it learns to the problem distribution it encounters (Section 6.1.1).

- *Given a PSG, which control rules should be learned?* In essence, DYNAMIC's algorithms for analyzing PSGs and acquiring control rules are the same as STATIC's.

Section 3.1 describes the component of DYNAMIC that chooses which PSGs to generate, and Section 3.2 sketches DYNAMIC's algorithms for analyzing PSGs.

3.1 Which PSG to Generate?

After PRODIGY solves a problem, DYNAMIC traverses PRODIGY's trace, labeling each node with the labels *success, failure, goal clobbering*, or *prerequisite violation* when appropriate. Only a small subset of these labeled nodes will become PSG roots. DYNAMIC traverses the labeled trace in pre-order, searching for appropriate learning opportunities. Consider, for example, an operator that fails because one of its preconditions could not be achieved. PRODIGY's trace will contain an entire subtree of failure nodes in which PRODIGY tried (and retried) to achieve the operator's precondition. However, only the operator's goal is output as a PSG root. All the failure nodes "below" the operator are ignored.

We have two motivations for carefully selecting PSG roots. First, for the sake of tractability, DYNAMIC seeks to avoid creating redundant PSGs. Creating a PSG for each node in the above example would be redundant since each PSG would contain roughly the same information. Second, since DYNAMIC does not employ utility evaluation [Minton, 1988] to discard or "forget" rules once they are learned, it seeks to avoid generating ineffective control rules in an *a priori* manner.

DYNAMIC utilizes the following policy to select appropriate learning opportunities, which is quite different from PRODIGY/EBL's example selection heuristics [Minton, 1988]. An appropriate learning opportunity is a node in the trace where one of the following conditions holds:

- *failure:* DYNAMIC finds the top nodes in the trace that have at least one child labeled *failure* and at least one child labeled *success*. DYNAMIC returns the subgoals at these nodes as PSG roots. In Figure 1, for example, `unstack` fails to achieve `(holding c)` but `pick-up` succeeds. Thus, DYNAMIC selects `(holding c)` as a PSG root for analyzing failure. Learning from failure is useful at such nodes because some alternatives were tried, leading to failure and backtracking, before the successful alternative was found. A similar heuristic is used by PRODIGY/EBL [Minton, 1988, page 84]. However, in contrast to PRODIGY/EBL, once such a node is selected, DYNAMIC does not analyze nodes in the subtrees below that node's failure children.

- *goal clobbering:* a node is labeled *goal clobbering* if an operator was applied at that node and its effects negate a previously achieved subgoal p. DYNAMIC collects all the nodes labeled *goal clobbering* from the trace and, for each of them, returns as a PSG root the top-level goal "pursued" by PRODIGY when the goal clobbering occurred as well as the top-level goal pursued when the subgoal p was achieved. In figure 1, for example, once `(holding c)` has been achieved, achieving `(on a b)` requires `(arm-empty)` which clobbers `(holding c)`. Thus, DYNAMIC returns `(on a b)` as a PSG root, and `(holding c)` as the clobbered goal.

- *prerequisite violation:* a node is labeled *prerequisite violation* if a precondition of the operator at the node was negated by a previously applied operator, and that precondition cannot be reachieved. DYNAMIC collects all the nodes labeled *prerequisite violation*. For each of them, it finds the top-level goal p that PRODIGY was pursuing at that node, and the top-level goal v pursued when the precondition of the operator was negated. Both p and v are returned as PSG roots for analyzing the prerequisite violation.

3.2 PSG Creation and Analysis

Once PSG roots are chosen, DYNAMIC's algorithm for creating and analyzing PSGs is as follows: [Etzioni, 1990a, Etzioni, 1992]

1. **Create a PSG:** Symbolically back-chain on the operators for achieving the subgoal at the root. PSG expansion is terminated according to a number of criteria (Figure 2). Most importantly, PSG expansion terminates whenever recursion is encountered. PSG leaves are labeled by a *failure*, *success*, or *unknown* label, depending on the termination criterion invoked at the leaf.

2. **Label the PSG:** Traverse the PSG, starting at the leaves, labeling each internal node based on the labels of its children. For example, because the failure of a precondition implies the failure of its parent operator, an operator is labeled *failure* if any of its preconditions is labeled *failure*. Some nodes are labeled *unknown* because their success or failure depends on the outcome of recursion, which DYNAMIC does not explore.

3. **Propagate failure/success conditions:** Traverse the PSG in post-order computing the conditions under which each PSG node would lead to failure or success during actual problem solving, based on the information at its children nodes.

4. **Extract operator and bindings rules:** Traverse the PSG in pre-order. Stop at the first node labeled *success* or *failure* and form a control rule that applies when the condition at the node is met, and whose recommendation is based on the node's label. An operator node labeled *failure* gives rise to an operator rejection rule, and an operator node labeled *success* gives rise to an operator preference rule. No control rules are derived from nodes labeled *unknown*.

In the case of goal clobbering, DYNAMIC also computes necessary effects. Specifically, DYNAMIC traverses the PSG in post-order determining the conditions that *necessarily* hold in the state *after* each node is reached during actual problem solving. For example, since STACK is the only Blocksworld operator that achieves the `(on a b)` subgoal, and since STACK invariably achieves `(arm-empty)` as a side-effect, `(arm-empty)` is a necessary effect of `(on a b)`, which negates `(holding c)`. DYNAMIC detects this fact by analyzing the PSG rooted in `(on a b)` and forms the control rule in Table 1.

To analyze prerequisite violations DYNAMIC computes necessary prerequisites. Specifically, DYNAMIC traverses the PSG in post-order determining the conditions that *necessarily* hold in the state *before* each subgoal can be reached during actual problem solving. For example, since all of the operators that achieve `(holding c)` require that `(arm-empty)` hold before they are executed, `(arm-empty)` is a necessary prerequisite of `(holding c)`. DYNAMIC acquires goal and operator ordering rules based on this analysis.

To illustrate this process, consider an example from PRODIGY's Schedworld problem space. Suppose that PRODIGY has the goal of making some machine part, say `p1`, `cylindrical` and `polished`. Rolling `p1` to make it cylindrical makes the part `hot` and polishing `p1` requires first to clamp it, which can only be done if `(temperature p1 cold)`. When PRODIGY first achieves `(shape p1 cylindrical)` using the `roll` operator, it cannot clamp `p1` and thus it cannot polish the part.[4] DYNAMIC labels the failure node in PRODIGY's trace as a *prerequisite violation* and builds the PSGs for `(surface-condition p1 polished)` and `(shape p1 cylindrical)` (figure 3). In this example, the prerequisite violation only occurs when `roll` is chosen to achieve `(shape a cylindrical)`. Thus, DYNAMIC learns the rule to prefer other operators over `roll` in table 3.

```
(DONT-ROLL-IF-NEED-POLISHED
 (if (and (current-node Node)
          (current-goal Node (shape Part cylindrical))
          (in-goal-exp Node
                  (surface-condition Part polished))
          (candidate-op Node roll)
          (candidate-op Node Op2)(not-equal roll Op2)))
 (then (prefer operator Op2 roll)))
```

Table 3: A rule learned by DYNAMIC to avoid a prerequisite violation.

[4]The problem space has no operator for cooling machine parts.

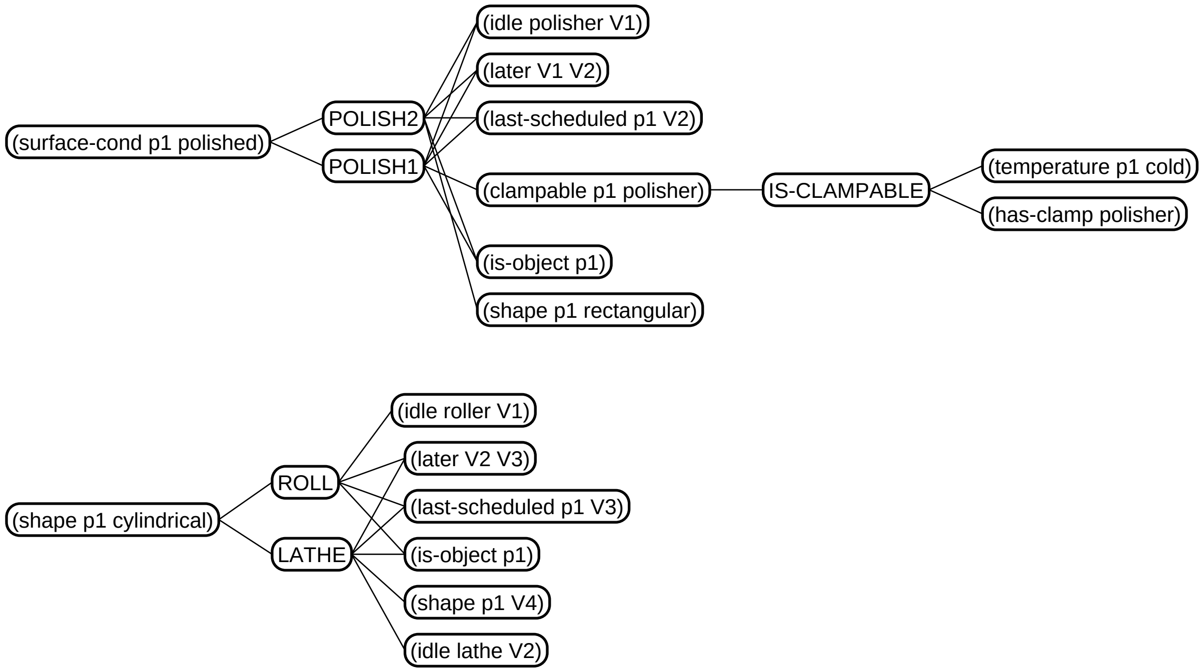

Figure 3: Two Schedworld PSGs built by DYNAMIC to avoid a prerequisite violation.

4 Local Analysis versus Global Analysis (an Example)

The description of DYNAMIC in Section 3.1 shows how it uses training problems to decide *what* to explain (i.e., which PSG to generate and what analysis strategy to use) enabling DYNAMIC to tailor its analysis to the training problems it is given. Once DYNAMIC has picked a PSG root, it then creates the PSG and analyzes it a-la-STATIC. Thus, explanation construction is independent of the training problem. This section provides an extended example illustrating the benefit of PSG analysis, compared with the local approach embodied by PRODIGY/EBL and other EBL systems.

When presented with the training problem in Figure 1, PRODIGY/EBL learns the overly-specific rule in Table 2. PRODIGY/EBL explains how achieving the `holding` subgoal followed by the `on` subgoal results in a goal interaction. Achieving `holding` violates the `arm-empty` prerequisite of `on`. PRODIGY/EBL's explanation is based on the *particular* plan PRODIGY used in this training problem. PRODIGY/EBL shows that the plan results in the goal interaction and *that other attempts to achieve* `holding` *fail, given the block configuration in the problem.*

Unfortunately, the second part of the explanation refers to the fact that blocks `a` and `c` are on the table. Since these features of the problem are mentioned in PRODIGY/EBL's explanation, they are incorporated into the antecedent of PRODIGY/EBL's control rule (Table 2). Yet, the location of the two blocks is incidental to the goal interaction that occurred. The goal interaction occurs *regardless* of the blocks' location. Because EBL mechanically accumulates every feature referenced by its explanation, it overlooks this fact and includes two extraneous conditions in the antecedent of its control rule.

Given the very same training problem, DYNAMIC discovers that *all* plans for achieving `on` clobber the `holding` subgoal. DYNAMIC makes this discovery by computing (a subset of) the necessary effects of each node in the PSG rooted in `(on a b)` and establishing that one of the necessary effects at the root of the PSG is `(arm-empty)`, which clobbers the `(holding c)` subgoal. Thus, DYNAMIC is able to acquire the more general rule in Table 1. DYNAMIC's analysis is more global than PRODIGY/EBL's because DYNAMIC (implicitly) considers *all* plans for achieving `on`, whereas PRODIGY/EBL analyzes only the plans that would succeed in PRODIGY/EBL's training problem, conjoining the conditions under which other plans fail to the antecedent of its control rule.

5 Experimental Results

This section compares the performance of DYNAMIC, STATIC and PRODIGY/EBL in PRODIGY/EBL's benchmark problem spaces (the Blocksworld, an extended version of the STRIPS robot-planning domain, and a machine-shop process-planning and scheduling domain). We utilize the same problem sets and rule sets used in the PRODIGY/EBL [Minton, 1988] and STATIC experiments [Etzioni, 1990a].

5.1 Experimental Methodology

To train DYNAMIC and PRODIGY/EBL we followed the training procedure outlined in [Minton, 1988], using a set of seventy-five to one-hundred randomly generated training problems. In addition, PRODIGY/EBL requires an additional "settling" phase, consisting of up to thirty randomly generated problems; during this phase learning is turned off, except for utility evaluation. DYNAMIC does not require utility evaluation. In the test phase, we ran PRODIGY on one hundred randomly generated problems in each of the problem spaces, under three experimental conditions: guided by PRODIGY/EBL's control rules, guided by STATIC's control rules, and guided by DYNAMIC's control rules. Following standard practice, a CPU time-bound (150 CPU seconds per problem) was imposed, to allow the experiments to complete in reasonable time.

5.2 Impact

Table 4 shows total problem-solving time, in CPU seconds, for solving the test problems in each of the problem spaces. DYNAMIC is able to speed up PRODIGY more than PRODIGY/EBL in the three problem spaces. With the exception of the Schedworld, the impact of DYNAMIC and STATIC is fairly close, compared with the impact of PRODIGY/EBL. In the Schedworld, both DYNAMIC and PRODIGY/EBL are hampered by an incomplete sequence of training problems. Given the appropriate training sequence, DYNAMIC's impact would converge to STATIC's. As shown in Section 6.1.1, DYNAMIC greatly outperforms STATIC on a highly skewed Schedworld problem distribution.

	PROD/EBL (orig.)	PROD/EBL (new)	STATIC	DYNAMIC
Blocksworld	261	263	115	99
Stripsworld	546	409	390	391
Schedworld	1311	2411	811	1211

Table 4: Total problem-solving time in CPU seconds. The results in the first column correspond to PRODIGY guided by the rule set generated by PRODIGY/EBL used in the experiments presented in [Minton, 1988, Etzioni, 1990a]. The second column refers to PRODIGY guided by the rule set obtained when running the experiments to measure learning time in Section 5.3. The difference between the two columns is due to machine-dependent parameters of PRODIGY/EBL's utility evaluation module.

5.3 Cost of Learning

Learning time is an important aspect of a learning method. Table 5 shows the learning time for the three systems. Both DYNAMIC's and PRODIGY/EBL's learning time include the time required to solve the training problems. Table 6 decomposes the learning time for both systems into three components: time to solve the training examples, time to analyze the problem solving trace and generate new rules, and time to perform utility analysis (for PRODIGY/EBL). Time spent by PRODIGY/EBL in the settling phase is included in the utility analysis time.

	PROD/EBL	STATIC	DYNAMIC
Blocksworld	1227	27	161
Stripsworld	1803	112	363
Schedworld	2984	61	1100

Table 5: Learning time in CPU seconds.

DYNAMIC's learning time is smaller than PRODIGY/EBL's for several reasons. First, DYNAMIC does not perform utility evaluation. Second, DYNAMIC learns more effective control rules and does so earlier. Since both systems use the learned control rules to guide PRODIGY during the training phase, DYNAMIC is able to solve the training problems faster. In addition, DYNAMIC's analysis of PSGs is more efficient than PRODIGY/EBL's schema-based explanation machinery. STATIC is faster than DYNAMIC for two reasons. First, DYNAMIC invokes PRODIGY to solve training problems (*ptm* in Table 6). Second, in addition to PSG-analysis, DYNAMIC traverses PRODIGY's traces searching for appropriate PSG roots (*ltm* in Table 6).

	PRODIGY/EBL				DYNAMIC		
	ua-tm	ltm	ptm	total	ltm	ptm	total
Blocksw	327	780	120	1227	117	44	161
Stripsw	169	1442	192	1803	223	140	363
Schedw	156	2436	392	2984	606	494	1100

Table 6: Learning time decomposed into utility analysis time (*ua-tm*), time to perform the learning itself (*ltm*), and time to solve the training problems (*ptm*). DYNAMIC's *ptm* is substantially smaller than PRODIGY/EBL's, for the same training problems, because DYNAMIC learns more effective control rules and does so earlier.

6 Comparative Analysis

This section analyzes the advantages of the DYNAMIC approach as compared with STATIC and PRODIGY/EBL.

6.1 Comparing DYNAMIC and STATIC

STATIC has one advantage over DYNAMIC. Running STATIC does not require invoking PRODIGY on training problems, an expensive process (column 6 in Table 6). Because it utilizes training problems, DYNAMIC has access to information about PRODIGY's task environment that STATIC is not privy to. This information provides DYNAMIC with the following advantages over STATIC:

- DYNAMIC can produce a more effective rule set that is tailored to the problem distribution it encounters (Section 6.1.1).
- STATIC's running time increases with the number of *possible* subgoals, whereas DYNAMIC's running time only increases with the number of subgoals *actually* encountered in PRODIGY's training problems. Although STATIC was faster than DYNAMIC in our experiments, we expect DYNAMIC to be dramatically faster in problem spaces with large numbers of possible subgoals, only few of which are actually encountered.
- To detect potential goal interactions, STATIC analyzes pairs of uninstantiated literals. As a result, STATIC is forced to consider all possible co-designation constraints between the different literals. If the two literals are `(on V1 V2)` and `(on-table V3)`, for example, the number of distinct co-designation relationships is three.[5] In contrast, DYNAMIC only considers the co-designation relationships actually encountered during problem solving.
- DYNAMIC has the potential to learn from recursive explanations whereas STATIC cannot. In addition, DYNAMIC has the potential to learn from N-ary goal interactions, whereas STATIC is restricted to pairwise goal interactions to keep its analysis tractable. Neither feature is implemented in DYNAMIC, and did not contribute to its success in Section 5 above. However, we plan to study learning from recursive explanations and N-ary goal interactions in future work.

6.1.1 Highly Skewed Problem Distributions

In essence, DYNAMIC utilizes its additional input (PRODIGY's trace on training problems) to learn a *subset* of STATIC's rules. On highly skewed problem distributions this subset can be considerably cheaper to match than STATIC's complete set, providing the DYNAMIC with a much more effective rule set. To demonstrate this point, we generated a Schedworld problem distribution where all the goals were instances of only two predicates: `(shape Part cylindrical)` and `(surface-condition Part polished)`. The problems were generated randomly with this restriction and an increasing number of parts and instances of these goals. We generated fourty problems for the training phase and eighty problems for the test phase. Table 7 shows the impact of the two systems on PRODIGY's problem-solving time.[6] On this problem distribution, DYNAMIC's smaller rule set is significantly better than STATIC's. Presumably, similar results could be obtained by running STATIC followed by a settling phase, leading STATIC to discard unused rules and achieve the distribution-sensitivity of DYNAMIC. However, this process would be inefficient in large problem spaces because, unlike DYNAMIC, STATIC would be forced to analyze each and every subgoal.

6.2 Comparing DYNAMIC and PRODIGY/EBL

DYNAMIC contains a number of advances over PRODIGY/EBL that are responsible for DYNAMIC's superior performance in Section 5, including simplified target-concepts, reliance on nonrecursive explanations, and the compactness of PSGs relative to problem-solving traces. These advances are discussed at length in [Etzioni, 1990a]. Here, we emphasize the global nature of DYNAMIC's analysis (Section 4), an important issue better understood only recently [Etzioni and Minton, 1992]. An additional benefit of DYNAMIC's approach is described below.

---

[5] STATIC does *not* build distinct PSGs for each case. Instead, it utilizes its matcher to *implicitly* consider all possible co-designation relationships [Etzioni, 1990a]. While elegant, this mechanism is still expensive.

[6] PRODIGY/EBL's performance was close to DYNAMIC's on this problem distribution.

	STATIC	DYNAMIC
Total CPU time	363	101
Total nodes	1156	1156
Avg time per node	0.314	0.087

Table 7: Total CPU time, number of nodes and average time per node for a skewed problem distribution in the Schedworld.

6.2.1 Fewer Training Problems

Our experiments show that DYNAMIC requires considerably fewer training problems than PRODIGY/EBL to learn the same (or better) control rules. Table 8 shows the number of training problems from which each system actually learned one or more control rules. The values for PRODIGY/EBL include problems from which rules were learned but later discarded because of their low utility. In addition, the table shows the number of problems that PRODIGY/EBL used in its settling phase. Note that PRODIGY/EBL tunes its utility measures on every problem in the training set, not only on those in the settling phase, so it might be argued that the number of problems actually used by PRODIGY/EBL exceeds the numbers reported in Table 8. As shown in [Minton, 1990], PRODIGY/EBL's performance deteriorates sharply when utility evaluation is not used.

	PRODIGY/EBL			DYNAMIC
	probs used	settling phase	total	probs used
Blocksworld	27	25	52	11
Stripsworld	33	35	68	9
Schedworld	44	25	69	19

Table 8: Number of training problems from which one or more rules were learned.

7 Conclusion

Previous EBL work represented one of two extremes: standard EBL methods that directly translate problem-solving traces into proofs, or static analysis methods (e.g., STATIC) that do not utilize training problems. This paper described the intermediate approach embodied by DYNAMIC which utilizes both training problems and PSG analysis. The main advantage of DYNAMIC over STATIC, is DYNAMIC's ability to tailor its analysis to the problem distribution it encounters. As shown in Section 6.1.1, DYNAMIC is more effective than STATIC on highly skewed problem distributions. We expect DYNAMIC to be significantly faster than STATIC in problem spaces with large numbers of possible subgoals, few of which are actually encountered.

We have experimentally demonstrated the power of the DYNAMIC approach and provided a detailed example (Section 4) illustrating how DYNAMIC generates more general control rules than PRODIGY/EBL. DYNAMIC suggests a general lesson for EBL systems: *training problems can pinpoint learning opportunities, but should not determine EBL's explanations.* Choosing *what* to analyze should be independent of *how* to analyze it. The only information DYNAMIC extracts from its training problems is which target concept (or analysis strategy) to apply to which subgoal. No information about the problem solver's state is used. While this approach has some advantages over standard EBL and static analysis, it is not necessarily optimal. Extracting more information from the training problem, and utilizing it to guide PSG analysis, is a worthwhile direction for future work.

Acknowledgments

The authors would like to thank Jaime Carbonell and Craig Knoblock for helpful comments and suggestions. Steve Minton contributed technical advice and some of the low-level routines used by DYNAMIC. The following people provided comments on previous drafts: Mike Barley, Ruth Etzioni, and Yolanda Gil.

References

[DeJong and Gratch, 1991] G. DeJong and J. Gratch. Book review of Learning search control knowledge: an explanation-based approach (by Steve Minton). *Artificial Intelligence*, 50:117–128, 1991.

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

[Etzioni and Minton, 1992] O. Etzioni and S. Minton. Why EBL produces overly-specific knowledge: A critique of the PRODIGY approaches. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference (ML92)*. Morgan Kaufmann, San Mateo, CA., 1992.

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

[Etzioni, 1990b] O. Etzioni. Why Prodigy/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[Etzioni, 1991] O. Etzioni. STATIC: A problem-space compiler for Prodigy. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

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

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

[Komorowski, 1982] H. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. In *Proceedings of The Ninth Conference on the Principles of Programming Languages (POPL)*, pages 255–267, Albuquerque, NM, 1982.

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

[Minton *et al.*, 1989a] S. Minton, J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989. Available as technical report CMU-CS-89-103.

[Minton *et al.*, 1989b] S. Minton, C. A. Knoblock, D. R. Kuokka, Y. Gil, R. L. Joseph, , and J. G. Carbonell. Prodigy 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

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

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

[Mitchell *et al.*, 1986] T. M. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[Mitchell *et al.*, 1991] T. M. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, , and J. C. Schlimmer. Theo: A framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, 1991.

[Newell and Simon, 1972] A. Newell and H. A. Simon. *Human Problem Solving*. Prentice Hall, 1972.

[Prieditis and Mostow, 1987] A. E. Prieditis and J. Mostow. Prolearn: towards a Prolog interpreter that learns. In *Proceedings of the National Conference on Artificial Intelligence*. Morgan Kaufmann, 1987.

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

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

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

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

[van Harmelen and Bundy, 1988] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36, 1988. Research note.

Contents