To appear in AI Journal

Bounding the Cost of Learned Rules

Jihie Kim and Paul S. Rosenbloom

Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292, U.S.A.
jihie@isi.edu, rosenbloom@isi.edu
(310) 822-1510 (x769)
Fax: (310) 822-0751

keywords: speedup learning, problem solving, utility problem, rule match

Abstract

In this article we approach one key aspect of the utility problem in explanation-based learning (EBL) — the *expensive-rule problem* — as an avoidable defect in the learning procedure. In particular, we examine the relationship between the cost of solving a problem without learning versus the cost of using a learned rule to provide the same solution, and refer to a learned rule as *expensive* if its use is more costly than the original problem solving from which it was learned. The key idea we explore is that *expensiveness is inadvertently and unnecessarily introduced into learned rules by the learning algorithms themselves.* This becomes a particularly powerful idea when combined with an analysis tool which identifies these hidden sources of expensiveness, and modifications of the learning algorithms which eliminate them. The result is learning algorithms for which the cost of learned rules is *bounded* by the cost of the problem solving that they replace.

We investigate this idea through an analysis of EBL_{Soar}, an implementation of explanation-based learning within the Soar architecture. A *transformational analysis* is used to identify where EBL_{Soar} inadvertently introduces substantial additional costs in the process of converting a problem solving episode into a learned rule — excessive costs which all ultimately turn out to stem from losses of information during learning. Based on these results, a modified EBL_{Soar} algorithm — Bounded EBL_{Soar} (BEBL_{Soar}) — is developed from which all sources of expensiveness have been eliminated. The cost of using a rule learned by BEBL_{Soar} is provably bounded by the cost of the problem solving it replaces.

1 Introduction

Explanation-based learning (EBL) (Mitchell *et al.*, 1986; DeJong & Mooney, 1986) has been widely used as a tool for improving problem solving performance (Minton, 1988; Dechter, 1990; Fattah & O'Rorke, 1993; Katukam & Kambhampati, 1994). For example, after solving a complex problem, EBL can acquire new search-control rules[1] by generalizing the experience to solve related problems more easily. However, the overhead of using learned knowledge often overwhelms its benefits, leading to a *utility problem* where problem solving cost after learning can be greater than before learning (Minton, 1988).

Research on the utility problem can be divided up into two key issues. The first issue is the *expensive-rule problem* — in which individual learned rules are so expensive to match that the system suffers a significant slow down from learning. The second issue is the *average-growth effect* (Doorenbos, 1993), which results from the cumulative expense of having learned many rules. If the time required to eliminate from consideration all of the rules that are not relevant to a particular situation scales poorly with the total number of rules in the system, this could potentially lead to a significant slow down from learning. Fortunately, recent work on the average-growth effect has shown that, by exploiting sharing and eliminating irrelevant match effort, it is possible to learn over one million rules with a sublinear cost increase (Doorenbos, 1993; Doorenbos, 1994).[2] This leaves the expensive-rule problem as the remaining open question, and thus what is focused on here.

In this article we take a novel approach to the expensive-rule problem by investigating the idea that *expensiveness is inadvertently and unnecessarily introduced into learned rules by the learning algorithms themselves.* This becomes a particularly powerful idea when combined with an analysis tool which identifies these hidden sources of expensiveness, and modifications of the learning algorithms which eliminate them. The result is learning algorithms for which the cost of learned rules is *bounded* by the cost of the problem solving that they replace.

This focus on boundedness rather than on speed up is a bit unusual — although see (Tambe *et al.*, 1990) for other work in this vein — but it is crucial for autonomous, real-time systems which learn unsupervised by human handlers. Without such boundedness, no amount of average-case speed up can ensure that learning won't cause catastrophic slow downs in system performance at crucial times. The only other alternative to having such boundedness has

[1] EBL can also be used to acquire other types of structures, such as macro-operators, but we focus on search-control rules here.

[2] Implicit in this approach is the assumption that learned rules substitute for previously available options, rather than augmenting them, so that the branching factor of the problem solving does not increase as rules are learned (Greiner, 1991).

been to disable such learning in autonomous, real-time systems.

A potential secondary benefit of this focus on boundedness is that, by eliminating the possibility of acquiring rules that are slower than the problem solving they replace, it should increase the average speed up provided by EBL. This additional speed up is not the focus of this work, and we won't distract from the main message by going to any great experimental lengths to validate it, but it is worth notice in passing.

Much previous work on the expensive-rule problem has investigated how to improve the utility of EBL through re-structuring and/or filtering learned rules based on experimentation with those rules (Minton, 1988; Greiner & Jurisica, 1992; Gratch & Dejong, 1992; Markovitch & Scott, 1993). Heuristic approaches to generating learned rules have also been proposed that provide improved efficiency over straightforward EBL — such as (Prieditis & Mostow, 1987; Minton, 1988; Shell & Carbonell, 1991; Shavlik, 1990; Etzioni, 1990). However, none of these approaches can guarantee that the cost of using the learned rules will always be bounded by the cost of the problem solving episode from which they are learned. That is, the cost of a learned rule can be greater than the cost of solving the problem with the original set of rules.

One other technique that can provide such boundedness does so by restricting the overall expressiveness of the rules used in the system (Tambe *et al.*, 1990). This in fact provides a stronger form of boundedness, in which the match cost of each rule — not just those acquired by learning — is bounded by a linear function of the size of the rule. However, in reducing the expressibility of the system's rules, this technique can require many more rules to encode the same tasks and can lead to overspecialization of the rules that are learned. The new approach to boundedness introduced here can also lead to some increased specialization of the rules learned — albeit by a lesser amount than is engendered by restricting expressiveness — but it does not restrict expressiveness, and thus does not increase the number of rules involved in, or the difficulty of, encoding tasks.

The first step in the new approach is to develop an analytical tool that can identify the sources of expensiveness in the learning algorithm. Here we introduce a form of *transformational analysis*. EBL is analyzed as a sequence of transformations that takes a problem solving episode as input and produces a learned rule as output. The key step in enabling the analysis is ensuring that the input (the problem solving episode), the output (the learned rule), and the intermediate structures generated during the transformational sequence — which, in general, are hybrids that partially resemble the problem solving episode and partly resemble the learned rule — can be interpreted by a performance system that allows their execution costs to be computed experimentally. Once this is done, their costs can be compared in order to identify which transformations introduce additional costs.

The transformational analysis is presented here in terms of EBL_{Soar}, an implementation of EBL in Soar (Rosenbloom *et al.*, 1991). Soar is an architecture that combines general problem solving abilities with a learning mechanism called *chunking* (Laird *et al.*, 1985). Soar is particularly relevant for this investigation because it is a widely distributed and used system for which there is a significant body of existing learning results spanning many domains and more than fifteen years (see, for example, (Rosenbloom *et al.*, 1993) for thorough coverage of the first half of this). In addition, the core of the work in Soar over the past five-to-ten years has concerned autonomous, real-time systems — see, for example, (Tambe *et al.*, 1995) — for which boundedness is a critical requirement for usable learning. Chunking, as implemented and extensively investigated in Soar, is already a variant of EBL (Rosenbloom & Laird, 1986); however, for this work we have replaced it with a more standard version of EBL in order to more easily generalize the resulting analysis to other EBL systems (Kim & Rosenbloom, 1995).

The transformational analysis of EBL_{Soar}uncovered three unexpected sources of expensiveness in the learning: (1) if the search-control knowledge used during problem solving is not maintained in the match process for learned rules, then learning can engender a slowdown; (2) when optimizations employed during problem solving, such as merging equivalent rule instantiations so that they are processed only once, are ignored in the learned rules, the cost can increase; and (3) if the structure of the problem solving is not reflected in the structure of the match process for the learned rules, time after learning can be greater than time before learning.

Interestingly, all three of these newly identified sources of expensiveness can be viewed as resulting from excessive losses of information during learning. EBL in general works by losing unnecessary information — for example, about intermediate structures and structures outside of the explanation. If too much information is lost during the learning process, the resulting rule will clearly be incorrect. What the sources of expensiveness uncovered here reveal is that loss of information can be excessive, even when not so great as to cause incorrectness. This milder form of excessive loss of information can introduce expensiveness into learned rules.

The second step in achieving boundedness is to modify the EBL algorithm in order to eliminate the identified sources of expensiveness. We have done this for EBL_{Soar}, yielding an algorithm called Bounded EBL_{Soar} ($BEBL_{Soar}$). The modifications, which all involve preserving information that was previously lost, are: (1) incorporating search control in learning so that the match process for the learned rule can focus on the path that was actually followed during problem solving; (2) exploiting equivalence among partial instantiations in match by creating one representative, which is then processed only once; and (3) maintaining the graph

structure of the problem solving in match to reinstate the efficiency of the structure. With these modifications, BEBL$_{Soar}$ has been proven to provide the desired boundedness.

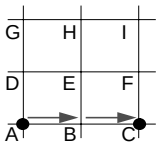
The remainder of this article is organized as follows. Section 2 overviews key aspects of EBL and Soar. Section 3 describes the match process and introduces the tool for measuring the cost of intermediate products. Section 4 presents the transformational analysis of EBL$_{Soar}$, and in the process identifies the transformations which introduce expensiveness. Section 5 presents algorithm modifications for each source of expensiveness, yielding BEBL$_{Soar}$. It also provides a proof of how BEBL$_{Soar}$ provides boundedness, plus a small empirical demonstration that the theoretical results hold up in the real system, by eliminating the slow downs caused by expensive rules in several of the domains that were previously shown to yield expensive rules. Section 6 covers related work. Section 7 concludes with a summary and a discussion of future work.

2 EBL and Soar

EBL is a learning method for making explicit a succinct concept description from the combination of a single example and underlying domain knowledge. Given its input (the *goal concept*, the *training example*, the *domain theory*, and the *operationality criterion*), the system constructs an *explanation* (also called a *proof tree*) of how the training example is an instance of the goal concept. An *explanation structure* is built from an explanation by replacing the rule instantiations with rules. Given the explanation structure, a variable unification process (called *regression*) is applied to it. Finally, a new definition — i.e., a sufficient condition for the goal concept — is generated from the operational elements of the regressed structure.

In Soar, each rule consists of a set of conditions and a set of actions. Conditions test a global set of relational facts, called *working memory*, for the presence or absence of working memory elements (WMEs). When the conditions of a rule all match elements in working memory, an instantiation of that rule is generated which binds variables in the conditions to constants in working memory. Rules fire when instantiations are generated, causing actions to execute with variables bound as in the instantiation of the conditions.

Actions create *preferences* rather than WMEs directly. Preferences may, for example, propose or reject candidate values for the given relations, or specify their relative worth (through *best*, *better*, *worse*, and *worst* preferences). Rules in Soar propose changes to working memory through these preferences, with the changes then actually being made based on a synthesis



W1: (next A B)
W2: (next A D)
W3: (right A B)
W4: (next B A)
W5: (next B C)
W6: (next B E)
W7: (right B C)
W8: (next C F)
W9: (next C B)
W10: (next D A)
W11: (next D E)
W12: (next D G)
W13: (next E B)
W14: (next E F)
W15: (next E H)
W16: (next E D)
W17: (next G D)
W18: (next G H)
W20: (at S A)
W21: (goal-point S GP)
W22: (goal-at GP C)

(a) Given WMEs

```
R1                          ; (operator proposal rule)
(at ?S ?L1)                 ; if the location of the current state
(next ?L1 ?L2)              ; is ?L1, and ?L1 is adjacent
-->                         ; to?L2, then create a candidate
(cand-operator ?S ?L2)      ; operator to go to ?L2

R2                          ; (search-control rule)
(at ?S ?L3)                 ; if the current location is ?L3,
(right ?L3 ?L4)             ; and ?L4 is on the right, and
(cand-operator ?S ?L4)      ; there is a candidate operator to
-->                         ; go to ?L4, then try the
(best-operator ?S ?L4)      ; operator before others

R3                          ; (operator application)
(operator ?S ?L5)           ; if the selected operator goes to
-->                         ; ?L5 then change the current
(at ?S ?L5)                 ; location to ?L5

R4                          ; (detection of success)
(goal-point ?S ?GP)         ; if the current location is ?L6
(goal-at ?GP ?L6)           ; and it is the goal point, then
(at ?S ?L6)                 ; the task is accomplished
-->
(success ?S ?L6)
```

(b) Given rules

FIGURE 1: A simplified Grid task in Soar.

of the preferences by a fixed *decision procedure*.[3]

The most critical decisions, and thus the most important preferences, concern the selection of *operators*. Operators represent actions in Soar, just as they do in most other problem solvers and planners. However, rather than there being a fixed operator language that is interpreted by the problem solving architecture, the execution of an operator is defined by additional rules which generate preferences for the consequences of the operator given that the operator has been selected (by the decision procedure) and that the conditions of the rules successfully match the existing state of working memory.[4]

For EBL_{Soar}, Soar's rules — both rules that participate in the selection of operators and those that apply them, plus any other rules that may also be utilized during problem solving, such as those that detect accomplishment of a goal — form the basis for the domain theory, the initial WMEs form the training example, and the operationality criterion is that WMEs are part of the training example (Rosenbloom & Laird, 1986). The explanation is built by extracting the rule firings that participated in the proof.

To make this more concrete, Figure 1 shows an example from the Grid task — a simple task known to produce expensive rules — which we'll use for illustration throughout much of this article. (For ease of interpretation, we use Lisp-style representations instead of the

[3]For a full discussion of Soar's preferences, their semantics, and how the decision procedure resolves combinations of them into decisions, see (Laird *et al.*, 1993).

[4]The test as to whether the operator has been selected is actually just another test of part of working memory.

Soar syntax.) In particular, we'll look at the task of evaluating whether point C is reachable from point A.

There are twenty two WMEs that provide a training example to the system. The WMEs record the connections among points. In this simplified task, the full connections among the points are given only in part. The rules provide the domain theory. In the rules in the Figure, preferences are represented as italicized prefixes; for example, *cand*-operator in the RHS of R1 represents a preference for the generation of candidate operators. The symbols prefixed with question marks (such as ?S, ?L1, ...) are variables. For example, variable ?S stands for state.

There are four rules in this task. Rule R1 creates a candidate operator for each point adjacent to the current point. Rule R2 creates *best* preferences, which lead the decision procedure to select the operators that go in the correct direction. Rule R3 applies the selected operator to the state, changing the current location to the new location indicated by the operator.[5] Finally, rule R4 detects the achievement of the goal by checking if the current location is the same as the goal point.[6] In this example, the goal concept is *success*; that is, the goal point is reachable from the the current position.

The cycle of rule firing — which creates preferences that yield WMEs via the decision procedure — and the cycle of operator selection and application which this engenders, underlies problem solving episodes in Soar, as illustrated for the Grid task in Figure 2.[7] The initial sequence of firings of rules R1, R2, and R3 selects and applies an operator which moves the current position from A to B. The subsequent sequence of firings of these three rules selects and applies an operator which moves the current position to C. Then, finally, rule R4 detects the achievement of the goal.

In Figure 2, each circle represents a rule trace. Traces representing the generation of candidates (e.g., traces of rules R1, R3, and R4) are shown in black, while traces representing the relative or absolute worth of candidates (e.g., traces of rule R2) are shown in gray. In the remainder of this article, rules that generate candidates will be referred to as *task-definition rules*, while those that represent the worth of candidates will be referred to as *search-control rules*.

The topmost circle in Figure 2 shows that R1 has fired twice and has created two preferences,

[5] In this simple example, we omit the condition for checking the current location for brevity.

[6] Although neither rule R3 nor rule R4 are shown in Figure 1 as explicitly creating preferences, in reality they do generate preferences for candidates rather than directly adding new elements to working memory.

[7] Problem solving in Soar also generally involves *impasses* and *subgoals* (Laird *et al.*, 1987), but an understanding of Soar at this level is not critical for the results in this article.

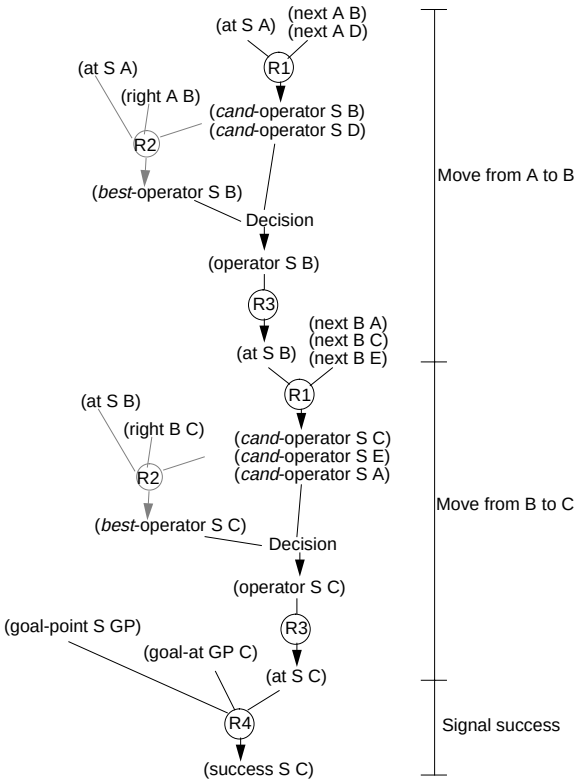


FIGURE 2: A problem solving episode from the Grid task.

(*cand*-operator S B) and (*cand*-operator S D). The two preferences propose candidate operators that go to B and D, respectively. A connection from one rule to another rule through a decision — marked by the word "Decision" — means that preferences created by the former rule are interpreted by a decision to create a WME which is then matched to a condition of the latter rule. For example, preferences (*cand*-operator S B), (*cand*-operator S D) and (*best*-operator S B) participate in a decision which creates a WME (operator S B), which is then matched to a condition of R3.

This problem solving episode forms the input to the learning system; that is, it provides the explanation for EBL$_{Soar}$.[8] However, instead of employing all of the rule traces which participated in the problem solving episode, EBL$_{Soar}$ only extracts traces from task-definition rules. The search-control rules, such as R2, are omitted in EBL$_{Soar}$ and other EBL systems (such as Prodigy (Minton, 1993)). The purpose of this omission is to increase the generality of the learned rules — including fewer rule traces in the explanation leads to fewer conditions in the resulting learned rule and thus to more generality of applicability. As a result, a smaller number of more general rules is learned rather than a larger number of more specific ones.

[8]In Soar, some problem solving activities do not involve rule firings. Because these activities are not represented as rule traces in Soar, they can leave holes in the explanation. Soar implicitly provides a set of axioms that model these activities, much as in Prodigy (Minton, 1988).

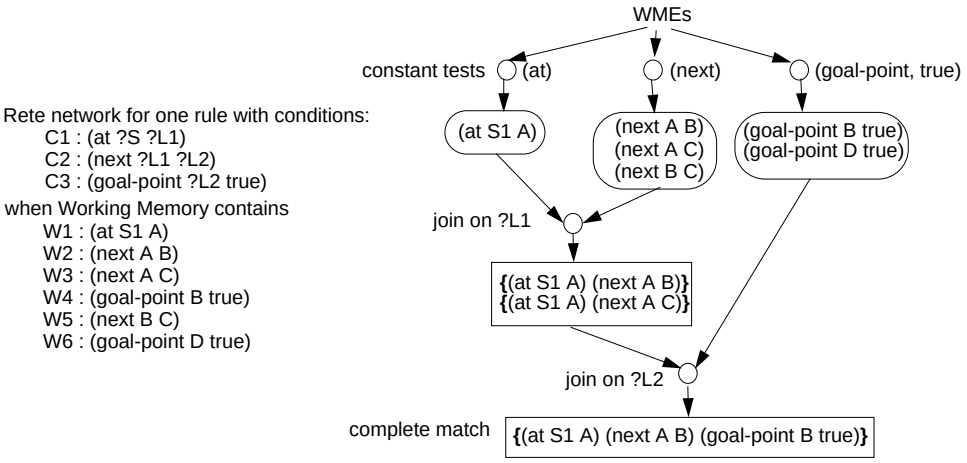


FIGURE 3: Rete network and associated tokens for a rule.

Omitting search-control rules from the explanation is considered safe because they only affect the efficiency of the problem solving — by (hopefully) directing the search down productive paths — not its correctness (which is instead determined by reaching a goal state). However, as will be seen shortly, it has a totally unexpected and very significant consequence on the match cost of learned rules.

The remaining steps in generating an executable rule involve unifying the output explanation structure via the EGGS algorithm (Mooney & Bennett, 1986), combining the rules in the unified explanation structure into a single EBL_{Soar} rule, and compiling the rule into the rule matcher. Firing of the learned rule can then generate the same result in place of what would have taken multiple rule firings by the original problem solving.

3 Measuring the Cost of Learned Rules

The cost of firing a rule is dominated by the cost of matching the rule, which is itself critically dependent on the match algorithm employed. As illustrated in (Doorenbos & Veloso, 1993), the match algorithms employed in speed-up learning can thus greatly affect the utility of the learned knowledge. Good matchers can help avoid a part of the utility problem, and bad matchers can significantly contribute to the problem. Traditional empirical utility analyses have ignored this issue, and instead implicitly assumed that any excessive costs introduced by suboptimal matchers will be taken care of by filtering out those rules that become too costly as a result. The approach we have taken instead is to start with a state-of-the-art matcher — an optimized version of *Rete* (Forgy, 1982), one of the most efficient rule-match algorithms known — and carefully analyze where excessive costs arise within the matcher.

This requires a thorough examination of the match algorithm, which may seem unusual in an article focused on learning, but is absolutely necessary under the circumstances.

Rete is based on compiling the conditions of rules into a data flow network called a *Rete network*. Figure 3 illustrates a Rete network for a simple rule with three conditions. Each WME is represented by a tuple that contains three items: relation name, object, and value. The network has two parts, the constant test part and the join part. The constant test part performs constant tests on WMEs (matching ground literals), such as tests for **at** and **true**. The outputs of these tests are stored in memories associated with the tests. Each memory contains the set of WMEs that pass all of the constant tests of a condition. The join part of the network contains join nodes and their memories. Join nodes perform consistency tests on variables shared between conditions, such as ?L1, which is shared between C1 and C2. Their memories store partial instantiations of rules, that is, instantiations of initial subsequences of conditions.[9] These partial instantiations are called *tokens*.

Rete's particular efficiency stems primarily from two key optimizations: *sharing* and *state saving*. Sharing of common conditions in a rule, or across a set of rules, reduces the number of tests performed during match. When conditions are shared, there is only a single path through the Rete network that represents all of them. State saving preserves the tokens generated by previous (partial) matches for use in the future. It occurs through the memories associated with constant and join nodes.

In this article we use the number of tokens generated during match as an analytical tool for measuring cost. This is the standard metric in the rule match community because counting tokens yields a measure that is independent of machines, optimizations, and implementation details; and because time per token is usually approximately constant (Tambe *et al.*, 1988). In Section 5.4, we will also present some results based on time to show that this does not dramatically alter the outcome.

Given the Rete match algorithm, techniques for measuring the number of tokens involved in matching a single rule are well established. However, we need to go beyond this, to be able to measure the number of tokens involved in a whole problem solving episode. We also need to be able to measure the tokens involved in executing each of the intermediate products — that is, the hybrid structures that are in between pure problem solving episodes and pure rules — in the transformational learning sequence. Moreover, we want to go beyond just measuring and comparing these numbers to proving boundedness relationships among them.

To do this we will first introduce the notion of a *quasi-rule*, and then define a set of analysis

[9] There also are negative nodes, into which negated conditions are compiled. A negative node passes a partial instantiation when there are no consistent WMEs (as with negation as failure).

tools, beginning with the concept of a *trace-graph*. A quasi-rule is a rule-like structure — in particular, one that can be executed to generate preferences (in Soar, at least) based on the situation in working memory — but one that is not simply constructed of the lists of conditions and actions that make up standard rules. For our purposes here, the quasi-rules of interest are problem solving episodes and the intermediate products generated during the transformational analysis. Neither structure fits a strict syntactic definition of a standard rule, but either can be executed to achieve an effect comparable to what a rule — such as the rule ultimately learned by EBL — would achieve.

A trace-graph represents the sequence of rule firings and decisions in a quasi-rule — a standard rule trace can be viewed as a degenerate trace-graph for a quasi-rule involving only a single rule firing. More precisely, a trace-graph of a problem solving episode specifies the WMEs participating in the rule matches, the partial instantiations generated in the matches (e.g., tokens in Rete), the rule instantiations, the output from the rule firings, and the subsequent uses of the output for the following rule firings.

Definition 1 (trace-graph) *The trace-graph of a quasi-rule is a directed acyclic graph consisting of a set of labeled nodes and directed edges.*

The nodes of the trace-graph for a problem solving episode are constructed as follows:

1. *For each rule firing in the quasi-rule, a rule node is placed in the trace-graph representing the set of instantiations that fired for the rule. The rule node is labeled with the name of the rule.*

2. *For each condition of each rule firing in the quasi-rule, a condition node is placed in the trace-graph representing the set of WMEs that matched the condition to produce the instantiations that fired for the rule. The condition node is labeled with the set of matched WMEs.*

3. *For each join of each rule firing in the quasi-rule, a join node is placed in the trace-graph representing the set of tokens generated at the join that produced the instantiations that fired for the rule. The join node is labeled with the set of tokens generated.*

4. *For each rule firing in the quasi-rule, a result node is placed in the trace-graph representing the preferences produced by the instantiations that fired for the rule. The result node is labeled with the set of preferences.*

5. *For each decision in the quasi-rule, a decision node is placed in the trace-graph representing the decision.*

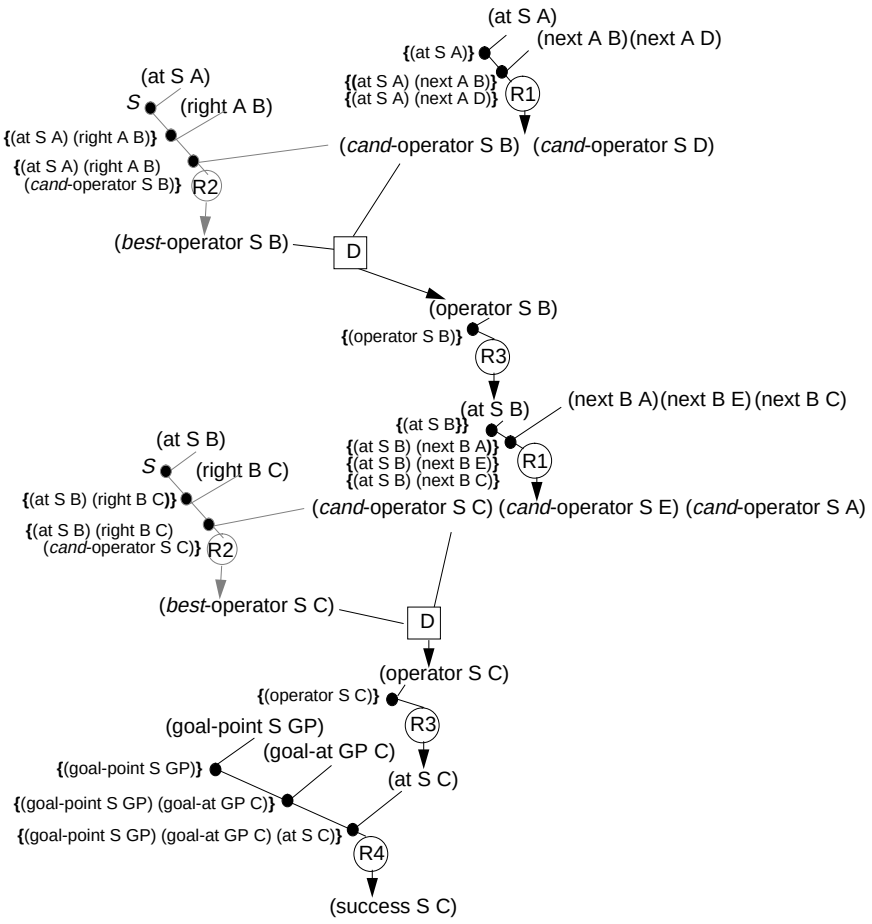


FIGURE 4: Trace-graph of problem solving in the Grid task.

The edges of the trace-graph for a quasi-rule are constructed as follows:

1. *Each condition node points to the join nodes it feeds.*
2. *Each join node points to the join nodes and rule nodes it feeds.*
3. *Each rule node points to the result node it feeds.*
4. *Each result node points to the decision nodes it feeds.*

Figure 4 shows a trace-graph for the example Grid problem solving episode, where rule nodes are represented as circles, condition nodes are represented by leaves (nodes not pointed to by any other node), join nodes are represented as dots, tokens at a join node are shown within curly brackets next to the join node, and decisions are represented by squares. For brevity, arrow heads are omitted for all edges except for those from rule nodes and decision nodes to result nodes (all other edges are implicitly directed downwards). An italicized letter S for a join node in the Figure indicates sharing of match effort with other rules having the same

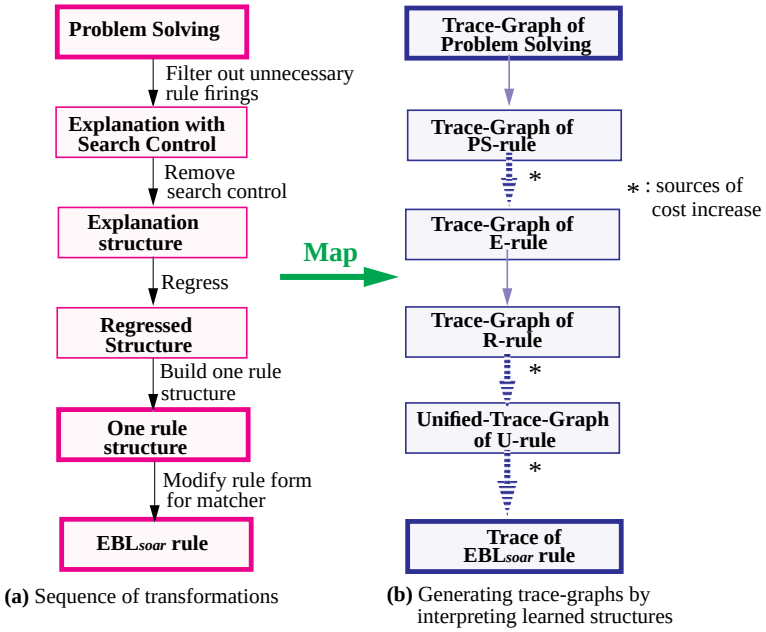


FIGURE 5: Transformational analysis of EBL_{Soar}.

pattern of conditions — for example, the first join node of R1 and R2 is shared. Sharing reduces the number of tokens generated because processing the shared structures just once yields matches for all of the structures shared.

This trace-graph shows the details of the match process. The tokens are generated by consistency tests between the instantiations of the previous conditions and the WMEs matching the current condition. The Rete algorithm creates two instantiations of R1 based on these tokens, and each instantiation creates a new candidate operator by executing the action. The total number of tokens created for firing R1 is 3. The total cost of the quasi-rule (the problem solving episode in this case) can be computed by summing the number of tokens across the full sequence of rule firings, yielding 16 in this case.

4 A Transformational Analysis of EBL_{Soar}

In this section we perform a transformational analysis of EBL_{Soar} by: (1) decomposing it into a sequence of transformations, (2) creating trace-graphs of the quasi-rules in the sequence, (3) comparing the trace-graphs and (4) identifying which transformations introduce expensiveness. We'll first describe each of these steps at a high level, and then go through them in more detail, on a transformation by transformation basis.

Figure 5-(a) shows EBL$_{Soar}$ decomposed into a sequence of transformations. The input to the sequence is the problem solving episode. The transformations in the sequence are successively: remove rules that were unnecessary for the EBL proof (Section 4.1); remove search-control rules (Section 4.2); regress the variables (Section 4.3); combine the set of rules that made it through the previous transformations into a single hierarchical "rule" (Section 4.4); and flatten the rule structure into an EBL$_{Soar}$ rule and compile it into a Rete network (Section 4.5).

Figure 5-(b) maps this transformational sequence onto one in which each stage is represented by a trace-graph (or simply a rule trace). By computing the cost of each quasi-rule, and of the final rule, we can analyze the cost changes brought about by the transformations. However, to do so, we much ensure that each of these structures has the same effective cost measure, so that the costs are comparable. As discussed in Section 2, we will use the number of tokens generated during match as an analytic tool for measuring and comparing the costs. The asterisks (*) in Figure 5 mark the transformations which these comparisons identify as introducing additional cost. More details on this will be provided as we proceed through the examination of each individual transformation in the sequence.

4.1 Remove Unnecessary Rule Firings ($\Rightarrow$ PS-rule)

The first step of EBL is filtering out unnecessary rule firings that did not participate in the proof. For the given example, this transformation eliminates all other rule firings, if there were any, beyond those shown in Figure 2. The resulting structure can be mapped to a type of quasi-rule, called a *PS-rule* (Problem-Solving rule), by providing an interpreter for it. The interpretation of the resulting PS-rule is similar to the original problem solving episode, except for the missing unnecessary parts.[10] That is, the trace-graph of a PS-rule is similar to the trace-graph of the problem solving, modulo the unnecessary parts.

The cost (in number of tokens) of the trace-graph is bounded by the cost of problem solving. If there were unnecessary rule firings in the problem solving, as is usually the case, the cost of a PS-rule would be strictly less than the cost of the problem solving. Otherwise, the cost would be the same as that of the problem solving.

[10] One other difference, but one that doesn't alter the token count, is that interpretation of a PS-rule is encapsulated; that is, communication of intermediate products during the interpretation of a PS-rule is localized completely within the PS-rule rather than going through global structures such as working memory (Kim, 1996).

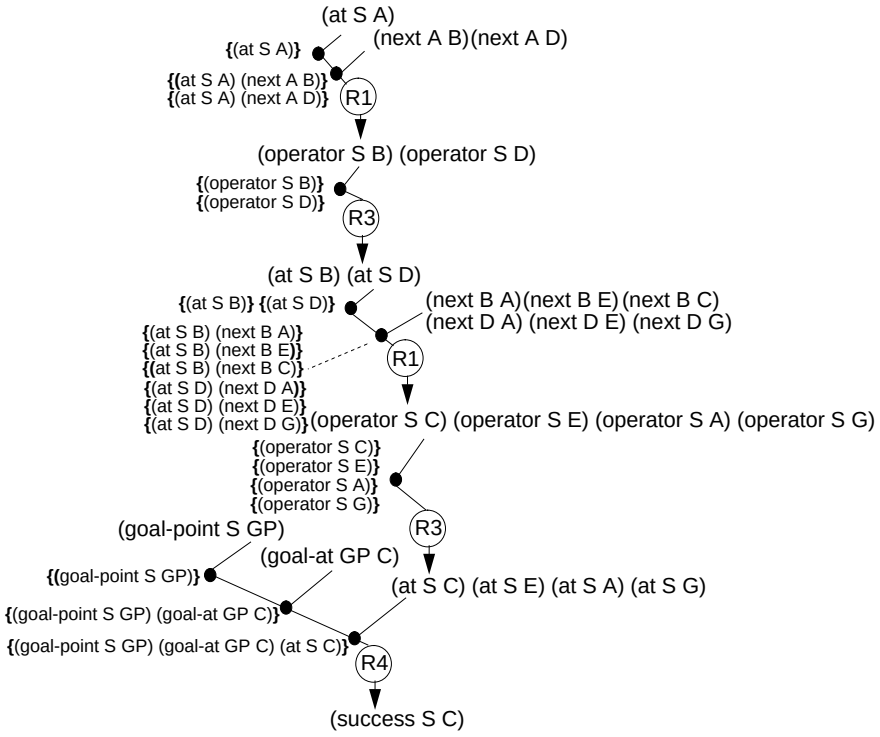


FIGURE 6: The trace-graph of the E-rule for the Grid task.

4.2 Remove Search Control ($\Rightarrow$ E-rule)

The PS-rule contains all the rule firings involved in the proof. However, EBL systems ignore search-control knowledge generated during the problem solving. The second step corresponds to this, explicitly removing search-control rules from the explanation structure. As explained in Section 2, the search-control rules are removed in order to increase the generality of the learned rules.

Figure 6 shows the trace-graph of the *E-rule* (Explanation rule) created from the PS-rule by removing the search-control rules. The traces of the search-control rule R2 and the nodes representing the decisions are gone, and only the traces of the task-definition rules remain. All candidates proposed by R1 now become WMEs without being filtered by the search control in a decision. The trace-graph of the E-rule can be mapped onto the normal proof tree or explanation in EBL. The interpreter for the E-rule is similar to the interpreter for the PS-rule except that the E-rule does not have to perform decisions.

The consequence of eliminating search-control knowledge is that the interpretation of the E-rule is not constrained by the path actually taken in the original solution. The interpretation can perform extra search even when the original search was highly directed (by the control rules). In the above example, without constraining the operator to the best candidate —

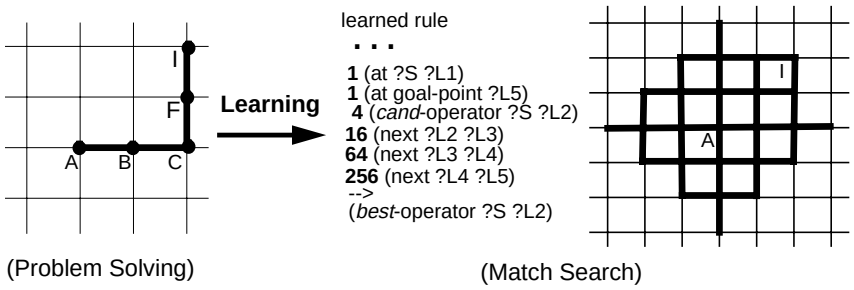


FIGURE 7: The difference between the search during problem solving and the search during the match of the learned rule.

which goes to the right — the number of tokens in the match of R1 in step (2) increases from 4 to 8, as shown in the Figure. Overall, the total number of tokens increases from 16 to 20 for this problem.

Although the increase in cost in this example is not terribly large, it can in other cases be huge. Consider a longer problem from the Grid task of going from point A to point I in Figure 7. With suitable control knowledge, the system can solve the problem of finding a path from A to I — for example, A, B, C, F, and I — in time that is linear in the length of the path. However, because of the elimination of search control rules in this transformation, the EBL_{Soar} rule learned from this search may be so general that, when it matches, it searches over all paths of length four instead of just a single path. Figure 7 shows the relationship between the search upon which the learning is based and the search performed, during the match, by the EBL_{Soar} rule learned from this search. The rule says that if you are at location ?L1 and want to get to location ?L5, and there is an operator that takes you from ?L1 to ?L2, and there is a connected path from ?L2 to ?L5 (via two intermediate points, ?L3 and ?L4), then the operator is the best choice. This rule is quite general, as it can solve any problem that has a solution of length four, and find all such paths, which is a key difference from the original problem solving with search control. This generality, however, is only obtained at an enormous cost. That is, the cost is exponential in the length of the path. Although, using this learned rule, the system can solve the same problem within a single rule firing instead of requiring multiple rule-firing cycles, the run time can become significantly longer because of this exponential match search.

4.3 Regress ($\Rightarrow$R-rule)

The next step in EBL is regression. Replacing the variable names with unique names (building the explanation structure) and then unifying each connection between an action and a

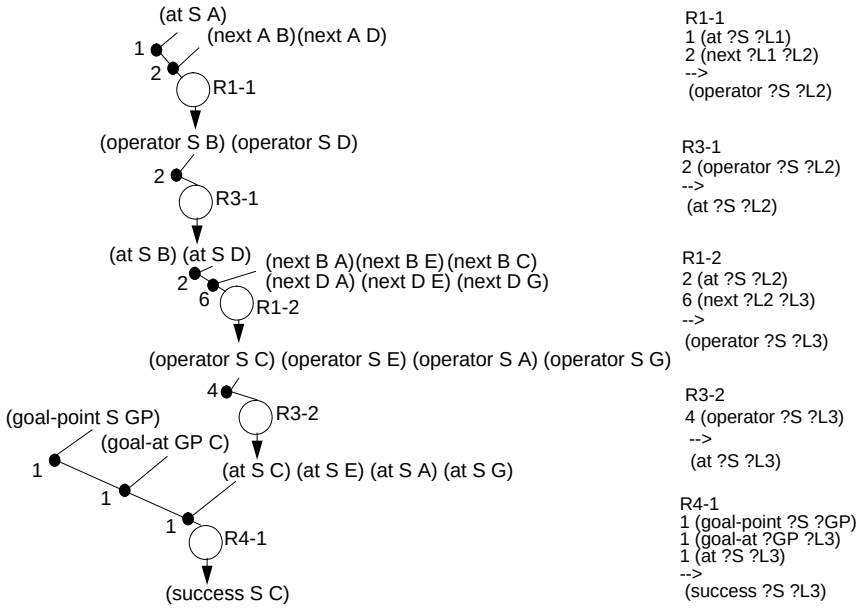


FIGURE 8: The trace-graph of the R-rule.

condition, can create an *R-rule* (Regressed rule) from an E-rule. We build the explanation structure by examining the trace-graph of the E-rule that is equivalent to the explanation (or proof tree), and applying the regression process of the EGGS algorithm (Mooney & Bennett, 1986) to the explanation structure. The trace-graph of the R-rule, resulting from the regression is shown in Figure 8. (For brevity, labels on join nodes have been omitted and just the numbers of tokens are shown.) In this example, the structure remains the same as in the E-rule. The interpreter for the R-rule is the same as the interpreter for the E-rule. Except for the differences in the variable names, the structures of the R-rule and the E-rule are identical. With respect to the cost, regression does not increase the number of tokens. The number of tokens should be the same, or be reduced by the extra constraints introduced by regression.

4.4 Unify Multiple Rules into One ($\Rightarrow$ U-rule)

This step of EBL unifies the separate rules in the explanation structure into a single rule. Figure 9 shows the result of unifying the example R-rule into the corresponding *U-rule* (Unified rule).[11] Although R1-1′, R3-1′, R1-2′, R3-2′and R4-1′ still have their own identifiable conditions in the U-rule, there are now no intermediate rule firings. The boundaries between

[11]The level of indentation in Figure 9, reflects the rule firing order in the problem solving. For example, the deepest indented conditions represent R1-1′, which corresponds to the instance of R1-1 that appears first (topmost) in Figure 8.

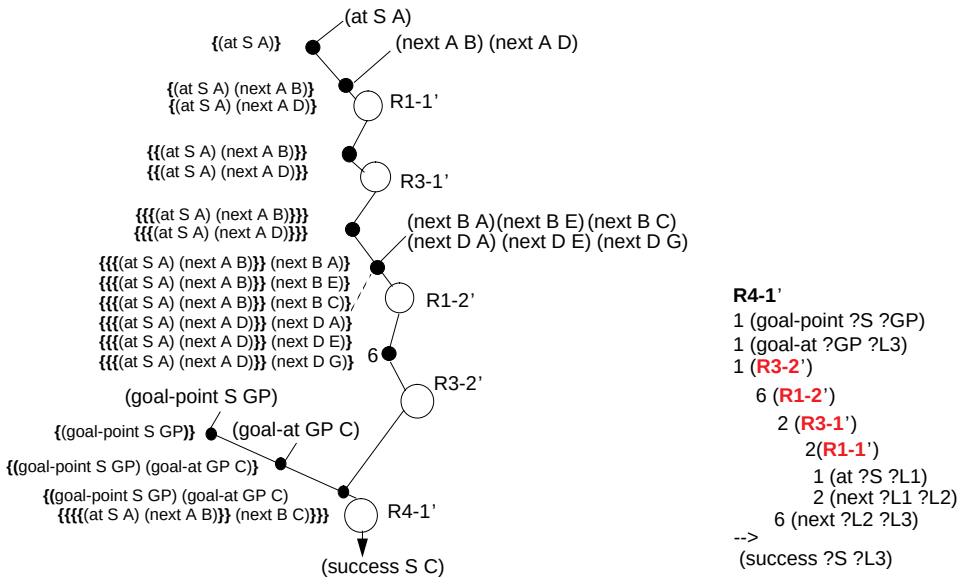


FIGURE 9: A unified-trace-graph of a U-rule for the Grid task.

the rules are eliminated by removing the intermediate processes of WME creation. In lieu of these processes, the instantiations generated by matching the earlier rules in the firing sequence (i.e., the tokens produced by their final conditions) are passed directly to the match of the later rules. In effect, this step replaces the intermediate WMEs with the instantiations that created the WMEs. For example, one of R3-1′'s conditions receives the instantiations of R1-1′ directly as intermediate tokens, rather than receiving WMEs created from the instantiations. Thus, R1-1′, R3-1′, R1-2′, R3-2′, and R4-1′ are no longer (separate) rules. Here, they are called *subrules*. A condition which matched intermediate WMEs created by a rule in the R-rule, is replaced by a *nonlinear condition* which tests the tokens generated by the subrule that is built for the rule. (What makes a condition *nonlinear* is explained in the next paragraph.)

To be able to properly interpret this structure (i.e., to measure the cost change through the transformation), an extension is required to the match algorithm. The traditional Rete algorithm, as shown in Figure 3, requires a *linear* match network, in the sense that a total ordering must be imposed on the conditions to be matched; such as C1, then C2, and then C3. In (linear) Rete, each *join* node checks the consistency of a token (a partial instantiation) against a WME, with each token itself being a linear sequence of WMEs, each of which matches one condition. Since the intermediate WMEs of the R-rule are replaced with instantiations in the U-rule, whenever the current condition receives instantiations instead of WMEs, testing consistency (via a join node) between the tokens of previous conditions and the current (nonlinear) condition should join two tokens, instead of joining a token and a WME. That is, U-rules require the ability to perform *nonlinear* matches, in which conditions

are matched via join nodes that compare pairs of tokens, rather than just a single token and a WME. They also require the ability to create hierarchically structured tokens (when pairs of incoming tokens are consistent); that is, a token must now be a sequence of WMEs or tokens (instantiations of subrules), instead of a sequence of WMEs only. An extension of Rete, called *nonlinear Rete* has been implemented to interpret this intermediate structure.[12]

Given this extension of the match algorithm, we need to extend the definition of the trace-graph to represent the cost of a U-rule.

Definition 2 (unified-trace-graph) *The unified-trace-graph of a U-rule is a directed acyclic graph consisting of a set of labeled nodes and directed edges.*

The nodes of the unified-trace-graph for a U-rule are constructed as follows:

1. *Rule nodes, condition nodes, and join nodes are constructed exactly as in a trace-graph.*
2. *For each subrule firing in the quasi-rule, a subrule node is placed in the trace-graph. The subrule node is labeled with the name of the rule from which it is generated.*

The edges of the unified-trace-graph for a U-rule are constructed as follows:

1. *Each condition node points to the join nodes it feeds.*
2. *Each join node points to the join nodes, subrule nodes, and rule nodes it feeds.*
3. *Each subrule node points to the join nodes it feeds.*

The unified-trace-graph in Figure 9 shows how tokens are created while matching (interpreting) the U-rule. Instantiations of subrule R1-1′ are provided as the instantiations of the condition of R3-1′. Also, instantiations of subrule R3-1′ are provided as the instantiations of the first condition of R1-2′. The consistency checking between the WMEs created by firing R3-1 and the instantiations of the second condition are replaced by consistency checking between the instantiations of R3-1′ and the instantiations of the second condition, based on the common variables between the subrule R3-1′ and the second condition. In this case, there is one common variable, ?L2, and the join node checks the equality of the instantiations of the variable. This process continues until R4-1′ is instantiated.

Cost problems are introduced in this transformation because the number of instantiations of a rule can be greater than the number of WMEs created from those instantiations. For

[12] We borrow the term used in (Scales, 1986; Lee & Schor, 1992) for referring to extensions of Rete to interpret generalized join nodes.

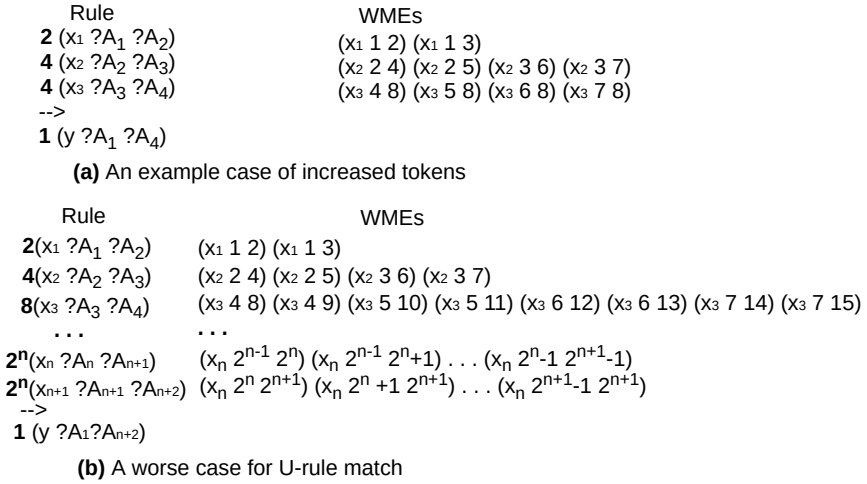


FIGURE 10: Number of tokens can increase in a U-rule.

example, given the rule and the WMEs in Figure 10-(a), four instantiations — (x_1 1 2) (x_2 2 4) (x_3 4 8), (x_1 1 2) (x_2 2 5) (x_3 5 8), (x_1 1 3) (x_2 3 6) (x_3 6 8), and (x_1 1 3) (x_2 3 7) (x_3 7 8) — are created. Because all these instantiations generate the same bindings for variables ?A and ?D, only one tuple (WME:(y 1 8)) is generated in the problem solving. Working memory is a set in Soar (as in many other forward-chaining rule systems), and does not include duplicate elements. Thus, the number of tokens is increased after the WMEs are replaced by the instantiations.

The grid task also suffers from this problem. In the R-rule, the six instantiations of R1-2 create four WMEs since there are only four points that can be reached by moving two steps from A. The four WMEs are then matched to the second condition of R3-2. However, in the U-rule, the six instantiations are directly used, creating two additional tokens. This increases the total number of tokens from 20 to 22. A worse case can arise when the working memory is structured as in Figure 10-(b). While the number of instantiations is exponential in the number of conditions, the number of WMEs remains at one.

4.5 Modify Rule Form for Matcher ($\Rightarrow$ EBL$_{Soar}$ rule)

The final step in EBL is creating a new rule in the system, and storing it in the rule memory for future matches. In the process of creating a new rule in EBL, the hierarchy in the explanation structure is linearized into a total ordering (as is required by nearly all rule-based systems). That is, EBL systems ignore the hierarchical structure of rule firings, and the structure of the match process for the learned rules differs from the structure of the problem solving. For example, the hierarchical structure in Figure 9 is linearized (totally

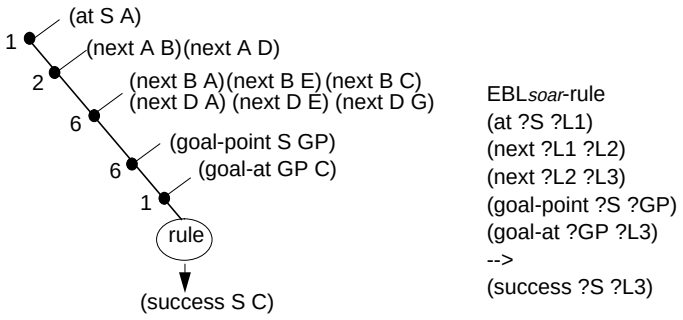


FIGURE 11: The trace-graph of the learned EBL$_{Soar}$ rule for the Grid task resulting from linearizing the U-rule.

ordered) to the structure in Figure 11. (The new rule is a task-definition rule, generating candidates for successful reach to the goal point.)

Note that standard EBL systems perform little optimization in the process. However, in EBL$_{Soar}$, when a new rule is created, conditions are reordered to improve the match performance. Also, when common conditions occur across multiple rules, duplicate match effort is saved by sharing within the Rete network.

The critical consequence of this step (linearization and condition ordering) is that the match structure of the learned rule is no longer constrained by the problem-solving structure. That is, how instantiations of different conditions are combined can be different from how they were combined during problem solving. This structural change introduces several different sources of expensiveness (Kim & Rosenbloom, 1996). Here we give two examples of these sources.

The first source arises directly from the linearization of the graph structure. By combining sub-graphs (of the subrules) together, some of the previously independent conditions become joined with other parts of the structure before they finish their sub-hierarchy match. Figure 12 shows an example. Figure 12-(b) shows the trace-graph of the problem solving, given the WMEs and rules in Figure 12-(a). For readability, the condition nodes of rules rule-conn and rule-close are also labeled with the name of the condition. In the problem-solving episode (and the U-rule), the conditions in a subrule (e.g., the conditions in rule-conn) are matched independently from the other parts of the structure (e.g., the conditions of rule-close) before its created WMEs are joined with the WMEs created by rule-close. By combining these sub-graphs together — through linearization — some of these previously independent conditions are joined with other parts of the structure before they finish their sub-graph match. In the linearized rule (Figure 12-(c) shows some possibilities that differ only in condition ordering), it is no longer possible to maintain independence between the conditions of rule-conn and rule-close. For example, in the first case in Figure 12-(c), tokens for the conditions from

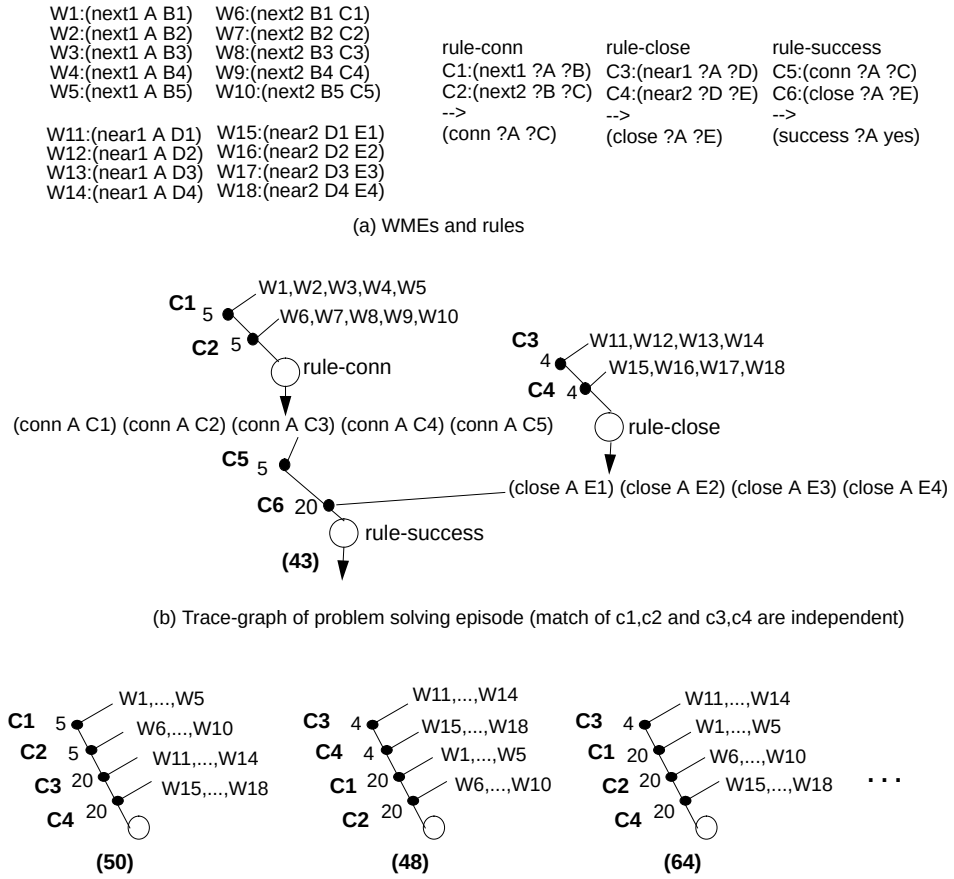


FIGURE 12: Loss of independence by linearization.

rule-close — (near1 ?A ?D) and (near2 ?D ?E) — are dependent on tokens for the conditions of rule-conn.

This *loss of independence* can increase the number of tokens. For the three orderings shown in Figure 12-(d), the number of tokens for the linearized structures are 50, 48, and 64, which are all greater than 43. Even with an optimal ordering, the number of tokens still increases in this example.

Another source of cost increase comes from *non-optimal ordering* of the conditions. As with (Smith & Genesereth, 1985), Soar uses a heuristic ordering algorithm because of the cost of finding optimal orderings (Scales, 1986). Whenever this heuristic condition-ordering algorithm creates a non-optimal ordering, additional cost may be introduced. For example, the Grid task can create the non-optimally-ordered rule shown in Figure 11. The cost is 16 with this rule. However, with an optimal ordering, as shown in Figure 13, the cost can be reduced to 6.

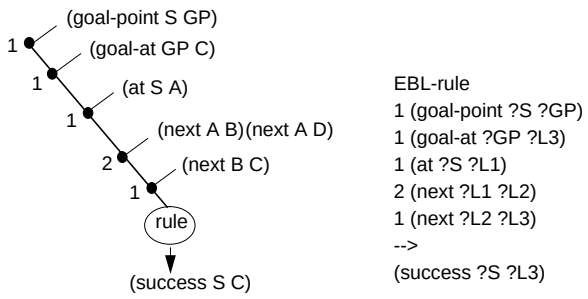


FIGURE 13: The match cost of an optimally ordered EBL$_{Soar}$ rule for the Grid task.

5 Modifying the Transformations

The transformational analysis in Section 4 identified three sets of sources of expensiveness: (1) removing search control, (2) disrupting the optimizations based on equivalent knowledge (by unifying), and (3) losing efficiencies stemming from problem-solving structures (by linearizing). As mentioned earlier, these sets of sources can all be viewed as stemming from excessive loss of information that was available during problem solving. In particular, information was lost about: (1) search control that was used; (2) which rule instantiations created equivalent results; and (3) the structure of the problem solving.

In this section these problems are eliminated by applying three modifications which enable the learning to reflect the lost information:

1. *Removing search control* $\Rightarrow$ *incorporate search control in learning.* By incorporating search control in the explanation structure, the match process for the learned rule can focus on the path that was actually followed.

2. *Disrupting the optimizations based on equivalent knowledge* $\Rightarrow$ *preprocess knowledge before it is used.* By preprocessing the knowledge, either by grouping the equivalent pieces of knowledge or by selecting one piece as a representative, an equivalent optimization can be achieved. These optimizations are called *token compression.*

3. *Losing efficiencies stemming from the problem-solving structure* $\Rightarrow$ *keep the problem-solving structure.* By keeping the graph structure employed in the problem solving, the efficiencies can be reinstated.

By applying these modifications to the original EBL$_{Soar}$ transformational sequence (Figure 14-(a)), it is altered into the Bounded EBL$_{Soar}$ (BEBL$_{Soar}$) transformational sequence shown in Figure 14-(b). Figure 14-(b) is annotated with the modifications.

The remainder of this section discusses the new BEBL$_{Soar}$ sequence of transformations in

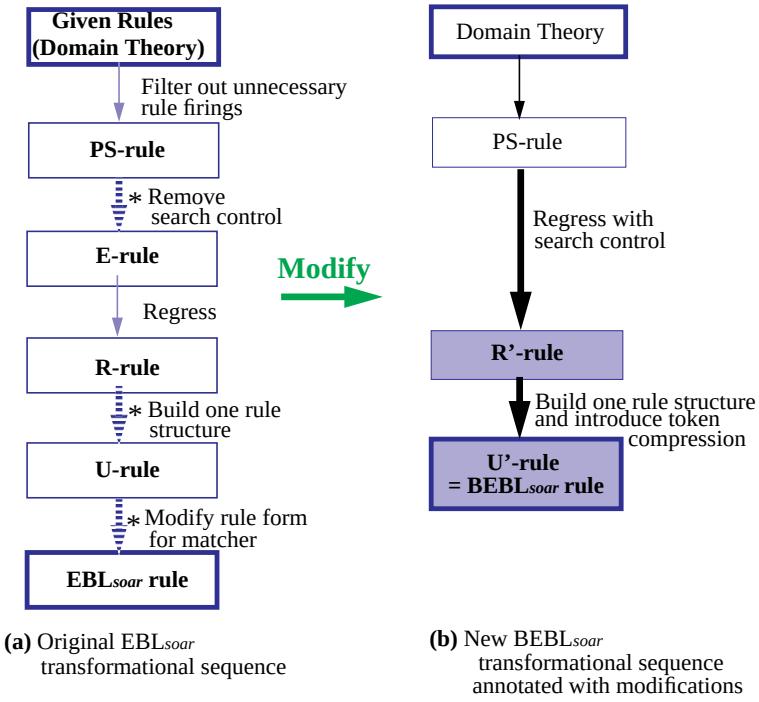


FIGURE 14: Modifications to avoid the sources of expensiveness.

detail, including proofs that the transformations avoid cost increases. However, before doing this, two qualifications must be mentioned. The first qualification is that there are aspects of the Soar architecture that can lead to overgenerality in learned rules (Laird *et al.*, 1986) — and thus to cost increases — whether the learning occurs via chunking or EBL_{Soar}. However, these architecture-level factors are separable from the rule-level transformations that are the core of explanation-based learning. Also, in practice, these factors do not engender significant cost increases (see, for example, (Tambe *et al.*, 1990; Doorenbos, 1993) and Section 5.4). Thus, the analysis of these aspects can be, and has been, left to future work.

The second qualification is that, while the boundedness theorems in this section do guarantee that a learned rule will not yield cost increases when the rule is used in the same situation for which it was learned, things are trickier when a learned rule transfers to a different situation. In particular, it is possible that if the learned rule had not transferred, then some other more efficient problem solving method would have been used. This is a form of *Einstellung* (Luchins, 1942) or *masking* (Tambe & Rosenbloom, 1993) that needs to be addressed separately. Einstellung is a general problem that can occur in any system that prefers to use more recently learned knowledge and methods over previously existing ones. The phenomenon was, in fact, first identified in natural systems such as people.

Now turning to the BEBL_{Soar} transformational sequence, the first transformation — from the domain theory to a PS-rule — is the same as in the original transformation sequence. We

prove the safety of this transformation in terms of the number of tokens in the trace-graphs. We then discuss the transformation from a PS-rule to an R′-rule, and the transformation from an R′-rule to a U′-rule (which is equivalent to a BEBL_{Soar} rule), respectively. We wrap up with a small empirical demonstration that these theoretical results in terms of tokens — which are the core validation of BEBL_{Soar} — still hold when cost is measured in terms of time.

5.1 Domain Theory ⇒ PS-rule

Before we prove that this transformation is safe, we define additional tools for comparing the relationships among quasi-rules.

Definition 3 (trace-subset) *Given the initial WMEs, a quasi-rule A is a trace-subset of a quasi-rule B, if (1) each rule node in A's trace-graph maps to a unique rule node in B's trace-graph, where both rule nodes were derived from the same rule application in the original problem solving episode; and (2) the label of each condition node is a subset of the label of the mapped condition node in B's trace-graph, where the mapping is based on (1).*

Theorem 1 *Given the initial WMEs, if a quasi-rule A is a trace-subset of a quasi-rule B, the number of tokens produced while interpreting A is less than or equal to the number of tokens produced by B. That is, the number of tokens in A's trace-graph is less than or equal to that in B's trace-graph.*

proof. Because A is a trace-subset of B, by condition (1) each rule node R in A can be mapped into a unique rule node R′ in B which is derived from the same rule application in the original problem solving episode. (Two different rule nodes in A cannot be mapped into the same node in B.) Also, because each condition in R matches to a subset of the WMEs matching the condition in R′ (condition (2)), there will be fewer (or the same number of) partial instantiations (tokens) produced while matching R as is produced for R′. Thus, the total number of tokens in A's trace-graph is bounded by the total number of tokens in B's trace-graph.□

Theorem 2 *For the same situation, the number of tokens produced while interpreting a PS-rule is bounded by the number of tokens produced by the problem solving episode from which the PS-rule is created.*

proof. Because the PS-rule is produced by eliminating the rule firings and the decisions not connected to the result creation, the problem solving episode employs either more rules and decisions than the PS-rule's trace-graph (when there is at least one excessive rule firing), or the same rules and decisions (when there is no excessive rule firing). Thus, condition (1) of trace-subset holds. Also, in PS-rule interpretation, only the WMEs created by the connected decision are matched by the rule condition, while in the problem solving episode, all WMEs are matched to all conditions. Thus, condition (2) of trace-subset holds. By Definition 3, the PS-rule is a trace-subset of the domain theory. By Theorem 1, the number of tokens produced in matching the PS-rule is bounded by that in the problem solving episode. □

For the Grid task example, the number of tokens remains the same — 16.

5.2 PS-rule ⇒ R′-rule (Regress with search control)

The transformation from a PS-rule to an E-rule has been dropped in order to retain the search control in learning.[13] The retention of search control, rather than the removal of it, may specialize the learned rules (and the quasi-rules created between this transformation and the learned rules), but in return it enables the rule's cost to remain bounded by the cost of the original problem solving. Also, the learned rules are specialized only as much as was the original problem solving episode (by the search-control rules employed in it).

The transformation from a PS-rule directly to an R′-rule requires a regression over the PS-rule. Figure 15 shows the new R′-rule built from the PS-rule in the Grid example. Although the characteristics of this transformation are similar to those of the original transformation, the resulting R′-rule is different from the R-rule in that copies of the search-control rules and the subsequent decisions are kept in the structure. The total cost in number of tokens is 16, which is the same as the match cost of the PS-rule. In general, the number of tokens generated should either be unchanged or reduced by the introduced constraints.

Theorem 3 *For the same situation, the number of tokens produced while interpreting an R′-rule is bounded by the number of tokens of the PS-rule from which it is created.*

proof. The R′-rule has the same set of rules as the PS-rule because the rules remain the

[13] If there are excessive control rules and the set of preferences over-determines the choice, the redundant preferences (and their rule traces) can be pruned from the explanation to make the created rule as general as possible (Kim, 1996).

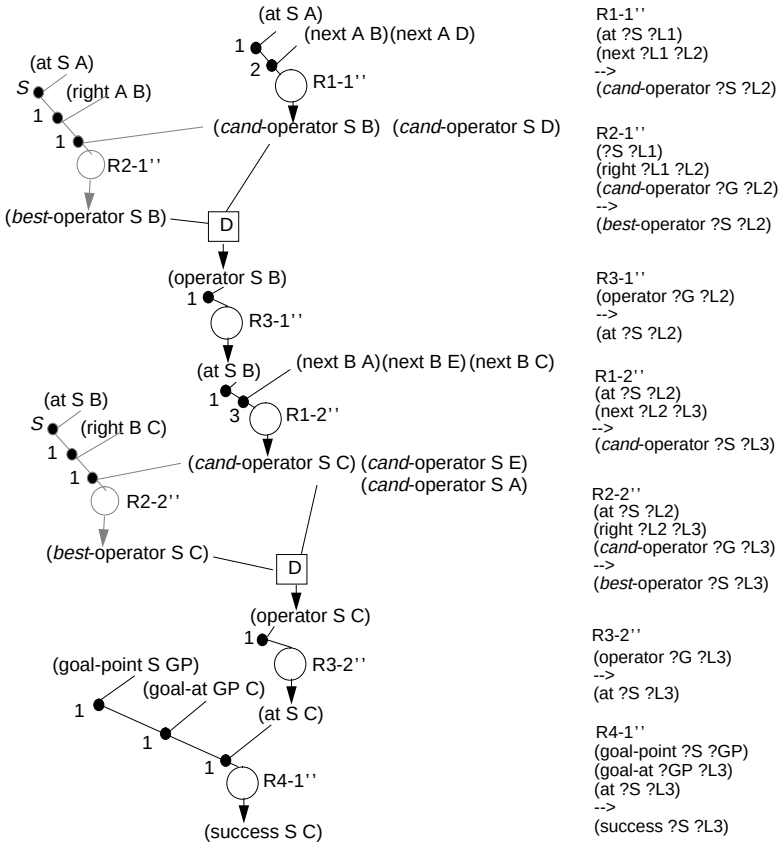


FIGURE 15: The interpretation of the R′-rule that is built while learning a rule from the Grid task.

same. Thus, condition (1) of trace-subset holds. The changes made by the transformation either make different variables the same or constrain the variables as constants. Because of these changes, a rule condition in the R′-rule either matches fewer WMEs than match the corresponding condition in the PS-rule, or matches the same WMEs (condition (2) of trace-subset). By Theorem 1, the number of tokens produced by the R′-rule match is bounded by that in the PS-rule match. □

5.3 R′-rule ⇒ U′-rule (create BEBL$_{Soar}$ rule)

This transformation has to perform two sub-transformations: (1) unifying the separate rules and decisions into one structure, and (2) applying token compression. These two sub-transformations should be applied together, because performing (1) without (2) can increase the cost. Also, (2) is not meaningful without (1); token compression is needed only when intermediate WMEs are replaced by tokens, which is what (1) does. The first sub-transformation removes intermediate preferences along with the subsequent intermediate

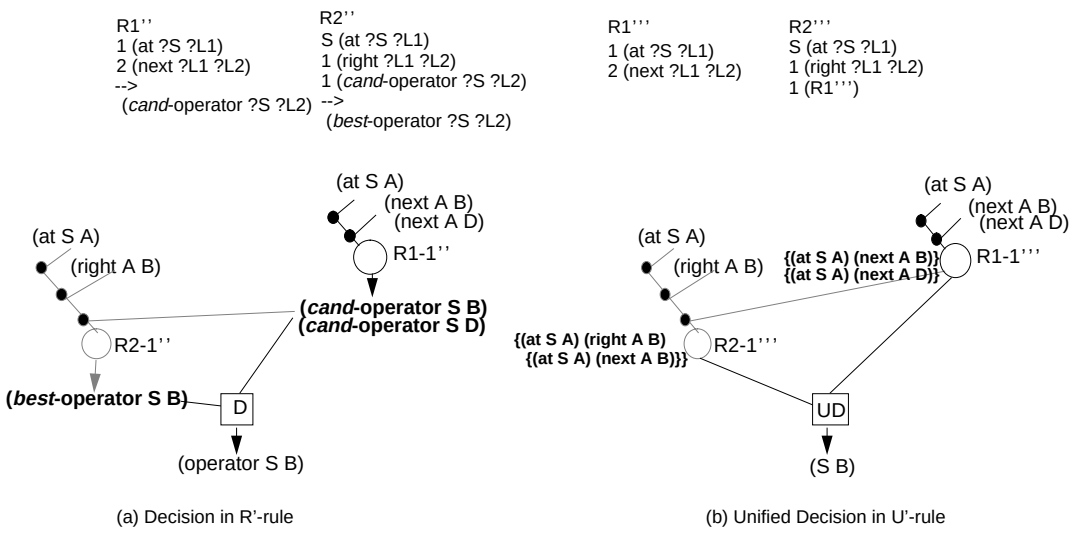


FIGURE 16: Transforming a decision in an R′-rule to a decision in a U′-rule.

WMEs, and produces a single rule structure. Token compression needs to be introduced to prevent any increase in the number of tokens caused by this unification.

5.3.1 Unifying: Removing intermediate preferences and WMEs

Removing intermediate preferences means that the instantiations of the rules that created the preferences are directly used in the decisions (instead of creating the preferences and processing them in the decisions). This requires a new decision algorithm which embodies the semantics of the decision procedure but processes instantiations instead of preferences. *Removing intermediate WMEs* also means that this decision algorithm does not create WMEs. The set of instantiations (of conditions) that participated in the decision is directly used for further matches. Figure 16 illustrates this through an example transformation in which intermediate preferences and WMEs are removed, and instantiations are directly processed by the decision. Since the decision process in the U′-rule is different from the normal decision procedure, in having to process instantiations rather than preferences, the decision is represented in the Figure as **UD** (*Unified Decision*) instead of **D** to denote the difference.

Unified decisions also differ from normal decisions in that they cannot be performed by calling a general decision procedure. As each rule has particular patterns in its conditions and actions, each unified decision has particular patterns for its input and output. These input and output patterns are determined by the subrules which participate in the decision. For example, as shown in Figure 16-(b), the given unified decision will process the instantiations of the subrules and make a unified decision based on the variable patterns of the subrules.

Execution of a U′-rule happens via a single match process — since a U′-rule is a single rule

— implying that all unified decisions in the U′-rule must be performed within the match algorithm. To support this, the match process has been augmented with a new algorithm that simulates the decision procedure in the match in order to produce the same result without performance overhead. Each unified decision only simulates the portion of the full decision procedure needed for the search control types which affected the decision. Further details of this modification are described partly in (Kim & Rosenbloom, 1996) and fully in (Kim, 1996).

5.3.2 Applying token compression

Without token compression, all the instantiations of the subrules are directly transferred to the connected rules in the U′-rule match. Figure 17 shows an example. In the figure, R4-variant is an extension of R4 which has one more condition that tests for prizes at the goal point. Also, R5 tests success and proposes a new game starting from the current position. Given these rules and the additional WMEs, matching R4-variant creates three instantiations: I1, I2, and I3 (Figure 17-(b)). Because all of these instantiations have the same values for the variables in the action in the R′-rule (?S and ?L3), only one WME (success S C) is created from them in the R′-rule match. The WME is then matched to the first condition of R5′.

Figure 17-(c) shows a part of the U′-rule match without token compression. Because the three instantiations are directly passed to R5′, the match cost can increase. At least for the first condition of R5′, the number of tokens increases from 1 to 3. The increase can be compounded when combined with likely increases from subsequent conditions.

To avoid such an increase in the number of tokens, token compression merges equivalent tokens (tokens for the same WME) into one token. Because the variables in the action determine which WMEs are created by action execution, one way of implementing token compression is to explicitly represent only the values of the variables in the action (and the mapping between the values and the variables). We call the variables in the action *exposed variables*. For example, the exposed variables for the action of R4-variant are ?S and ?L3. Given these exposed variables, as shown in Figure 17-(d), the three instantiations are merged into one tuple (S C) which is used instead of the three instantiations in the U′-rule match. Because the tuple represents any of the three instantiations, it is not removed until all three instantiations are removed.

Unifying rules replaces intermediate WMEs with instantiations that created the WMEs, and token compression replaces the instantiations with tuples of exposed variables' values. A tuple is different from a WME in that its creation and deletion are performed within

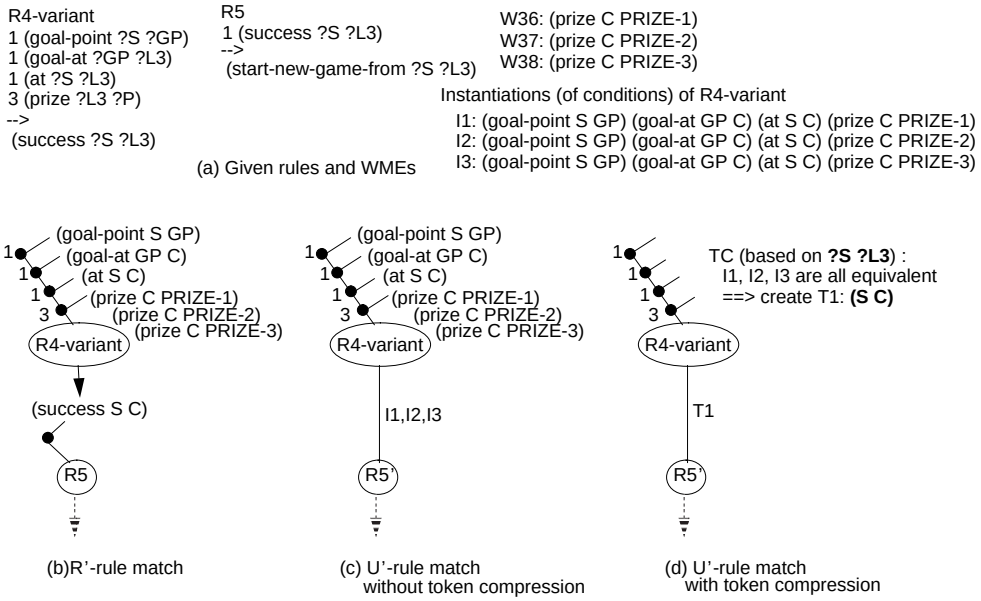


FIGURE 17: Building a U′-rule with and without token compression.

one rule (U′-rule) match, instead of across multiple rule matches and decisions. In general, because the number of tuples is always bounded by the number of WMEs, and tuples provide the same binding information about exposed variables as do WMEs, the cost increase from unifying can be avoided.

In order to analyze U′-rules quantitatively we need to extend our earlier concept of a unified-traced-graph to an *extended-trace-graph*. In the following definition, the extensions are shown in bold face.

Definition 4 (extended-trace-graph) *The extended-trace-graph of a U′-rule is a directed acyclic graph consisting of a set of labeled nodes and directed edges.*

The nodes of the extended-trace-graph for a U′-rule are constructed as follows:

1. *Rule nodes, subrule nodes, condition nodes and join nodes are constructed just as in a unified-trace-graph.*
2. **For each unified decision in the U′-rule, a unified-decision node is placed in the extended-trace-graph representing the unified decision.**

The edges of the extended-trace-graph for a U′-rule are constructed as follows:

1. *Each condition node points to the join nodes it feeds.*
2. *Each join node points to the join nodes, subrule nodes, and rule nodes it feeds.*

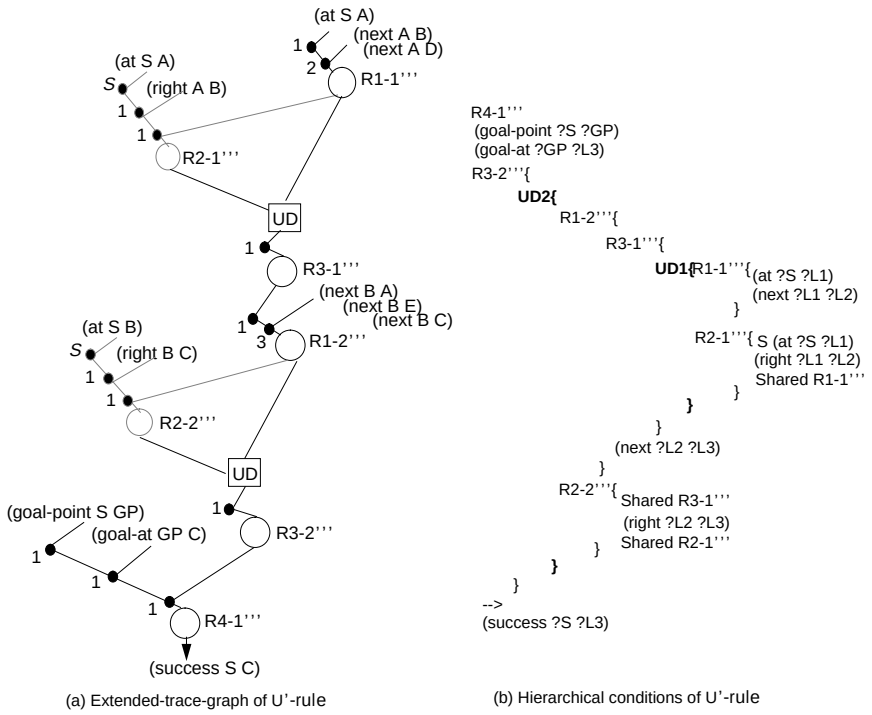


FIGURE 18: The extended-trace-graph and conditions of the U′-rule that is built while learning a rule from the Grid task.

3. Each subrule node points to the join nodes **and unified-decision nodes** *it feeds.*

4. **Each unified-decision node points to the join nodes it feeds.**

The new U′-rule built from the original R′-rule in Figure 15 is shown in Figure 18. The copies of the search-control rules and the subsequent decisions are kept in the structure. The total cost in tokens remains unchanged, instead of increasing.

R4-1‴ in Figure 18-(b) shows the hierarchical condition structure of the U′-rule. There are two unified decisions, and each of them introduces the constraints required to avoid the sources of additional cost, as the original search-control rules and decision procedure did. The number of tokens generated will be either the same, or reduced, by applying the above set of optimizations. Before we prove that this transformation is safe, we define one more tool for comparing the relationships between an R′-rule and a U′-rule.

Definition 5 (extended-trace-subset) *Given the initial WMEs, a U′-rule A is an extended-trace-subset of a quasi-rule B, if (1) each rule node and subrule node in A's extended-trace-graph maps to a unique rule node in B's trace-graph, where both rule nodes (or the subrule node and the rule node) were derived from the same rule application in the original problem solving episode; and (2) for each condition node C in A and its corresponding condition node*

D in B, each WME (or tuple, when C is for a nonlinear condition) T in C's label can be mapped to a unique WME W in D's label in that T and W contain equivalent information about the variable bindings.

Theorem 4 *Given the initial WMEs, if a U′-rule A is an extended-trace-subset of quasi-rule B, the number of tokens produced while interpreting A is less than or equal to the number of tokens produced by B.*

proof. Because A is an extended-trace-subset of B, each (sub)rule R in A can be mapped to a unique rule R′ in B which is derived from the same rule application in the original problem solving episode. For each condition node C in R, since each tuple (or WME) in the node's label can be mapped to a unique WME in the label of the corresponding condition node C′ in R′ (extended-trace-subset), there will be fewer (or the same) partial instantiations (tokens) produced while matching R than are produced for R′. Thus, the total number of tokens in A's extended-trace-graph is bounded by the total number of tokens in B's trace-graph. □

Theorem 5 *Given the initial WMEs, the number of tokens produced while interpreting a U′-rule is bounded by the number of tokens of the R′-rule from which it is created.*

proof. Each subrule in the U′-rule is created from a unique rule in the R′-rule. Thus, each subrule in the U′-rule's extended-trace-graph maps to a unique rule in the R′-rule's trace-graph (condition (1) of extended-trace-subset). Since the unified decision filters out candidates based on search-control semantics, and token compression picks one representative for each set of duplicate instantiations, condition (2) of extended-trace-subset holds. By Theorem 4, the number of tokens produced by the U′-rule match is bounded by that in the R′-rule match. □

This completes the proofs that the individual transformations are bounded. From here it is a short step to proving the overall boundedness of BEBL_{Soar}.

Theorem 6 *For the same situation, the number of tokens produced while interpreting a $BEBL_{Soar}$ rule is bounded by the number of tokens produced by the problem solving episode from which it was learned.*

proof. Follows directly from Theorems 2, 3, and 5. □

To illustrate how match works differently for the rules learned by BEBL$_{Soar}$ from those learned by EBL$_{Soar}$, Figure 19 shows the number of tokens generated at each condition during the match of a corresponding pair of learned rules from the Grid task (but from a longer problem than previously used as an example). In the EBL$_{Soar}$ rule (Figure 19-(a)), there are huge cross products in the match, leading to a maximum number of 3989 tokens at a condition. In the BEBL$_{Soar}$ rule (Figure 19-(b)), the number of tokens at a condition does not grow to more than 4.

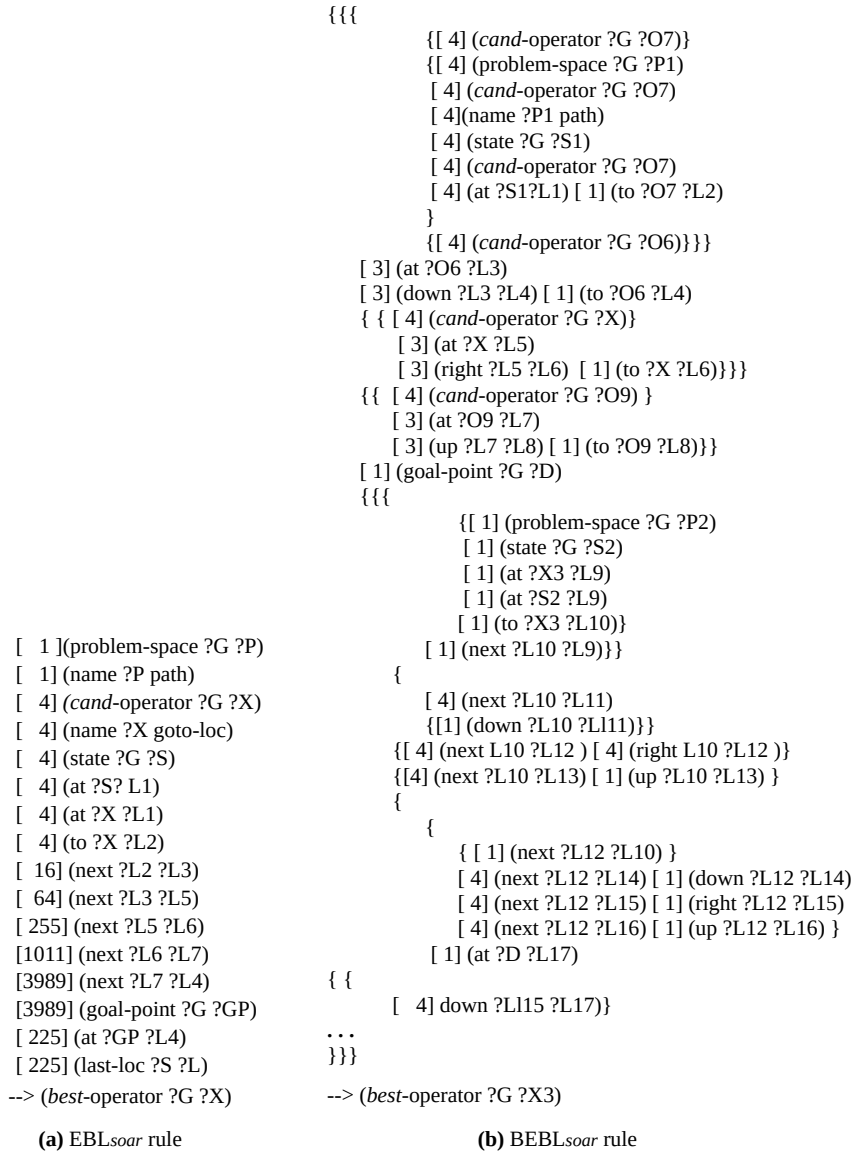
In Figure 19-(b), braces mark the beginning and ending of subrules. This hierarchical structure reflects the problem-solving structure. Shared subrules are not shown in the figure for brevity. The conditions shared across the different sub-parts reflect the multiple usage of those conditions in the original problem solving. This multiple usage keeps the cost bounded, by constraining the sub-parts as they were in the problem solving. Although the rule conditions built by BEBL$_{Soar}$ look rather complex, and can be difficult to read, they introduce the constraints required to avoid the sources of cost increase and thus to bound the cost of the learned rule.

5.4 Timing Demonstration

The new BEBL$_{Soar}$ learning system follows the transformation sequence shown in Figure 14-(b), and utilizes an interpreter for search-control incorporated, token compressing, nonlinear rules. Since the previous sections have proven that BEBL$_{Soar}$ provides bounded learning in terms of the number of tokens generated during rule match, an extensive experimental validation of this implementation is not called for. However, it is important to at least demonstrate that this theoretical boundedness in tokens carries over to real boundedness in time, with all the additional complexities that are involved (such as variations in time per token and growth effects from learning multiple rules).

So, in this section, we take a brief look at comparative timings for EBL$_{Soar}$ and BEBL$_{Soar}$ in several tasks that have proven in the past to learn expensive rules: the Grid task, the N-Queen task, and the Magic Square task (Tambe *et al.*, 1990). We compare the CPU times from three different variations of Soar 6.0.4 (a C-based version of Soar) on a Sun SPARCstation-20/61: without learning, with the rules learned by EBL$_{Soar}$, and with the rules learned by BEBL$_{Soar}$. The results are presented in Figure 20.

The results from the Grid task are the average problem solving time (in seconds) for two sequences of problems, one containing problems with a path length of six and the other problems of length seven. In the length six Grid tasks, the average CPU time with EBL$_{Soar}$



(a) EBL*soar* rule

(b) BEBL*soar* rule

FIGURE 19: Number of tokens for corresponding learned rules in the Grid task.

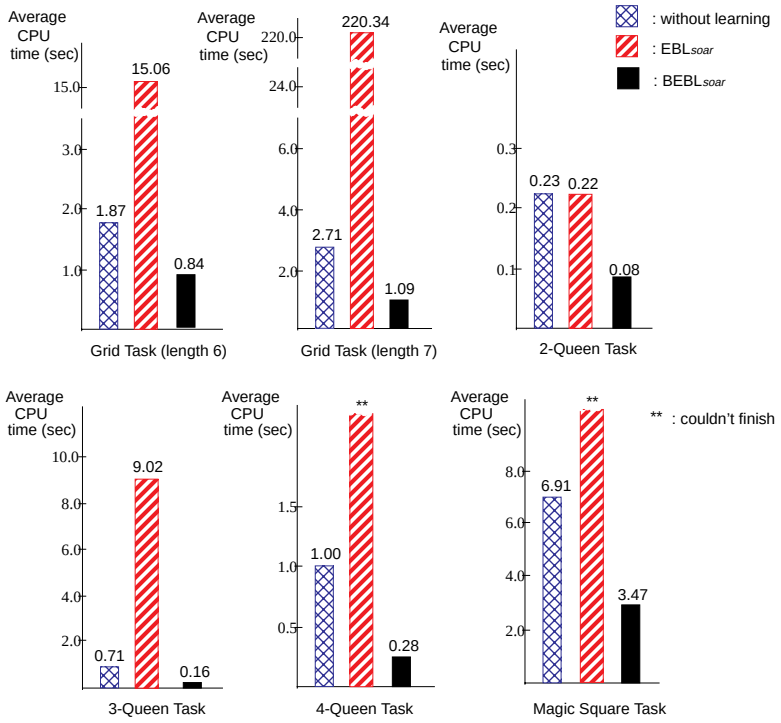


FIGURE 20: Average CPU times for expensive-rule tasks.

rules is eight times greater than the average CPU time of the system without learning. In the length seven tasks, the time with EBL_{Soar} rules is roughly eighty times greater than the time without learning. EBL_{Soar} clearly slows down the problem solving in both cases; that is, it is an expensive-rule task. Also, the slowdown factor in the length seven tasks is greater than that in the path six tasks, so the problem is getting worse as the problems get bigger. However, the time with BEBL_{Soar} rules is always less than the time before learning; in particular, the time is less than half of that without learning.

The 2-Queen task is to place two queens in a 3 $\times$ 3 grid without them being attacked by each other. The 3-Queen task and 4-Queen task place three and four queens, respectively, in a 4 $\times$ 4 grid. In the 2-Queen task, the time with EBL_{Soar} rules is almost the same as the time without learning. In the 3-Queen task, the time with EBL_{Soar} rules is more than ten times greater than the time without learning. As in the Grid task cases, the slowdown factor seems to increase as the size of the task increases. In fact, in the 4-Queen task, the system could not even finish learning with EBL_{Soar}. The number of tokens for the learned rule reached more than eight million and the system could not allocate enough memory. In contrast, with BEBL_{Soar} rules, time after learning is always bounded by time without learning, and is roughly three to four times better than the time without learning.

The Magic Square task is to place tiles 1 through 9 in a 3 $\times$ 3 square in such a manner that all of the rows, columns, and diagonals sum to the same number. The results show the same

pattern as in the N-Queen tasks. With EBL_{Soar} rules, the system could not finish learning. However, the CPU time with BEBL_{Soar} rules is bounded by the time without learning — the time without learning is greater than the time with BEBL_{Soar} rules by a factor of two.

6 Related Work

Considerable prior work has occurred on reducing the expensiveness of learned rules. One class of approaches to the expensive-rule problem has focused on directly reducing the cost of learned rules. Some approaches have restructured and simplified the learned rules to semantically equivalent ones in order to reduce the match cost of the rules (Prieditis & Mostow, 1987; Minton, 1988). Other approaches have analyzed the problem solving structure that is the basis of the learning to either avoid particular structures (such as recursion) in the learned rules (Etzioni, 1990), or to preserve such structures in the learned rules (Shell & Carbonell, 1991; Shavlik, 1990; Subramanian & Feldman, 1990). The approach described here is similar in spirit to this last thread. Although we don't focus on explicitly identifying particular problem solving structures, we do use the structure of the problem solving as a strong constraint on the form of the learned rule. Where we go beyond the earlier work is in identifying all the aspects of the problem solving structure of the given performance system that can lead to increased cost in the learned rule if not accounted for, and then eliminating them.

A second class of approaches to the expensive-rule problem is selective learning, where the utility of learned knowledge is evaluated and only the useful knowledge is kept. Several systems assume a fixed distribution of problems, and select those performance-system transformations that allow increased utility; for example, PRODIGY/EBL's utility evaluation (Minton, 1988), where it measures the utility in terms of the savings and cost of a rule, and rules are deactivated if their utility is estimated as negative. PALO (Greiner & Jurisica, 1992) and Composer (Gratch & Dejong, 1992) navigate through the space of performance elements, and select rules only if they show incremental utility. The information filtering model (Markovitch & Scott, 1993) proposes a more general framework for selective learning, and defines various methods for eliminating harmful knowledge from the learning system. The approach presented in this paper is able to dispense with the processes of utility evaluation and filtering, yet still provide a boundedness guarantee with respect to the cost of individual rules.

Although the selective approaches are weaker with respect to this form of boundedness, they

do have the advantage that they can deal directly with both the potential issue of expensive transfer to problems other than the one for which a rule was learned and the issue of average growth effect — in both cases, by filtering learned rules whose net effects are negative. With respect to expensive transfer, we have not yet run across a case where this is actually a problem; however, should it turn out to be a real problem, some form of solution may need to be added to the boundedness techniques introduced here. With respect to the average growth effect, we expect to be able to utilize recent optimizations that enable acquiring over one million rules while still allowing their efficient use (Doorenbos, 1993; Doorenbos, 1994), although the combination of this result and the approach described here still needs to be analyzed.

A third class of approaches to the utility problem is to use inductive learning techniques to learn simpler (or approximate) control rules with reduced match cost (Cohen, 1990; Zelle & Mooney, 1993). These approaches are on the other side of the spectrum of maintaining versus dropping (or simplifying) information for efficiency in the learning process. BEBL_{Soar} keeps the performance information as well as the accuracy information in learning to provide boundedness of learned rules.

A fourth class of approaches has focused explicitly on providing boundedness in learned rules, as we do here. They have focused on *absolute* boundedness — where guarantees are provided on the absolute cost of using learned (or any) rules — rather than on the *relative* boundedness that has been the focus here (where the cost of using learned rules is bounded by the cost of the problem solving they replace). Absolute boundedness comes closer to the kinds of guarantees that are required in hard real-time systems; however, they can only do so by sacrificing either expressivity (Tambe *et al.*, 1990; Tambe & Rosenbloom, 1994) or completeness (Haley, 1987; Barachini & Verteneul, 1988). Also learned rules may become very specific when expressivity is sacrificed. Relative boundedness can maintain whatever real-time guarantees have already been provided by the pre-learning performance system — although while not providing real-time guarantees on the pre-learning performance system — without these limitations.

A fifth class of approaches has focused on reducing the cost of learned rules by utilizing better match algorithms (Doorenbos & Veloso, 1993). The approach described here also focuses on the match algorithm. It starts with a state-of-the-art Rete algorithm and augments it in various ways to enable cheaper matches of learned rules. However, simply reducing the cost of using existing learned rules is not enough for boundedness. It is also necessary to change the structure of the learned rules in fundamental ways, and thus also of the match algorithm used for them.

In addition to these five classes of approaches to the expensive-rule problem, the work de-

scribed here is related to other work that has employed a transformational view of EBL (Keller, 1983; Mostow, 1983; Bostrom, 1993; Segre & Elkan, 1994). In this related work, learning is explicitly viewed as occurring via a sequence of transformations. The focus is on developing a useful set of basic transformations, and via them a large space of potentially useful sequences of transformations. EBL is then one possible transformational sequence in this space. What this related work has in common with the work described here, is the focus on learning via a transformational sequence. However, our focus here has been on using the transformational sequence as a tool for developing a bounded learning algorithm, rather than on a general exploration of the space of learning sequences.

7 Conclusion

Many learning systems suffer from the utility problem, where the time for problem solving after learning is greater than the time before learning. Discovering how to assure that learned knowledge will in fact speed up system performance has been a focus of research in explanation-based learning (EBL). This article focused on ensuring that the cost of using learned rules is no more than the cost of the problem solving they replace, based on the idea that expensiveness is inadvertently and unnecessarily introduced into learned rules by the learning algorithms themselves. We developed a two step process: (1) finding the complete set of sources that can make learned rules expensive for a particular system, and then (2) modifying the learning process to avoid these sources. To find the set of sources of expensiveness, we introduced a novel way of analyzing the learning process — the transformational analysis. The essence of the analysis is to decompose the learning process into a sequence of transformations in which the cost of intermediate products can be computed. By computing and comparing the match cost of each intermediate product, the cost changes through the learning were measured and isolated within particular transformations.

This research used EBL_{Soar}, with a state-of-the-art Rete match algorithm, as a vehicle for the investigation. EBL_{Soar} has been decomposed into a sequence of transformations from a problem solving episode to the matching and firing of a learned rule. The match cost of each intermediate product (quasi-rule) was measured by counting the number of tokens produced in the match to generate the result. By analyzing the transformations, we identified a set of sources which can make the output rule expensive. In addition to identifying the sources, the analysis also pointed the way towards modifications of the transformational sequence that could eliminate the sources. The set of sources and the corresponding modifications are:

1. *Removing search control ⇒ incorporate search control in learning.* By incorporating search control in the explanation structure, the match process for the learned rule can focus on the path that was actually followed.

2. *Disrupting the optimizations based on equivalent knowledge ⇒ preprocess knowledge before it is used.* By preprocessing the knowledge, either by grouping equivalent pieces of knowledge or by selecting one as a representative, an equivalent optimization can be achieved.

3. *Losing efficiencies stemming from the problem-solving structure ⇒ keep the problem-solving structure.* By keeping the graph structure employed in the problem solving, the efficiencies can be reinstated.

As mentioned earlier, one general way of viewing these three sources of cost increase is as *excessive loss of information.* The kinds of information that are lost in the learning process don't lead to incorrectness in the resulting rules, but do lead to significant losses in efficiency. The corresponding modifications avoid the loss in efficiency by retaining the appropriate information through the learning process.

Implementing the identified modifications requires significant changes in the underlying Soar architecture, especially in the learning and match algorithms. This set of modifications has been applied to EBL_{Soar}, converting the original sequence of transformations into a provably bounded one, called BEBL_{Soar}. In addition, the match algorithm has been extended to handle search-control incorporated, token compressing, nonlinear rules. The boundedness of the resulting system has been empirically demonstrated in three different expensive-rule tasks.

Looking towards the future, several outstanding problems still require attention. First, it is important to understand the extent to which this same approach can apply to other implemented EBL systems, and perhaps even more broadly to other speed up learning systems, to see if similar gains in boundedness can be achieved there. For example, PRODIGY/EBL (Minton, 1988) can be viewed as consisting of a transformational sequence that starts with a target concept, and recursively specializes the concept with respect to a problem solving episode until it reaches primitive concepts. The unified result of the specialization becomes the conditions of a new rule. Then PRODIGY/EBL simplifies the rule to reduce the match cost and performs a utility evaluation to discard high cost rules. Finally, the resulting learned rule is added into its rule system. This sequence of transformations may be analyzed in terms of how each one affects the cost. That is, the intermediate products may be interpreted based on its performance system, and compared in order to identify which transformations introduce additional costs. It may even be possible to design structures similar to trace-graphs

that could serve as the basis for computing these costs (Minton, 1993), and to further focus the search for sources of cost increase by concentrating on those locations where the learning mechanism discards information used during problem solving.

Second, using nonlinear rules leads to diminished rule readability. Even with indenting to reflect the level of hierarchy, the sharing of sub-conditions is still difficult to understand. One way of relieving this problem is to further simplify the structure of the rules. There are additional ways of simplifying the graph structure beyond those already implemented, including modifications to the nonlinear structure to make it more efficient (Kim, 1996). By implementing such optimizations, the match performance may be improved, as well as the readability of the rules. It is also possible to improve readability by linearizing the rules when presenting them to humans. In this way it would be possible for them to focus on the content of the learned rule, without being distracted by the structure.

Third, as mentioned in Section 5, the proof of how BEBL$_{Soar}$ provides boundedness does not cover the aspects of the Soar architecture that can lead to overgenerality in learned rules. A similar analysis for these aspects is needed.

Fourth, a more thorough investigation of the additional specialization engendered by BEBL$_{Soar}$ is warranted. We know that its level of specialization is no worse than the problem solving from which it learns rules (Section 5), and that it is less than what is engendered by existing approaches which limit expressiveness in order to guarantee absolute boundedness (Kim & Rosenbloom, 1993). However, we do not yet have a full characterization of its impact on performance or the number of rules that must be learned.

Fifth, and finally, BEBL$_{Soar}$ needs to be combined with a solution to the average growth effect. Earlier work on the average growth effect in chunking has shown that it is possible to learn large numbers of rules without hurting overall system performance (Doorenbos, 1993; Doorenbos, 1994). However, because the rules created by BEBL$_{Soar}$ can be different from the rules created by chunking, the problem still needs to be addressed in terms of BEBL$_{Soar}$.

Acknowledgment

This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Defense Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL); and under contract N66001-95-C-6013 from the Information Systems Office (ISO) of the Defense Advanced Research Projects Agency (ARPA) and the Naval Command and Ocean Surveillance

Center, RDT&E division (NRaD). We would like to thank Jonathan Gratch, Steve Minton, and Milind Tambe for helpful comments on this work.

References

Barachini, F. & Verteneul, G. (1988). The challenge of real-time process control for production systems. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 705–709.

Bostrom, H. (1993). Improving example-guided unfolding. In *Proceedings of ECML-93*, pages 124–135.

Cohen, W. W. (1990). Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 268–276.

Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutsets decomposition. *Artificial Intelligence*, 41:273–312.

DeJong, G. F. & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176.

Doorenbos, B. (1993). Matching 100,000 learned rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence.*

Doorenbos, B. (1994). Combining left and right unlinking for matching a large number of learned rules. In *Proceedings of the Twelfth National Conference on Artificial Intelligence.*

Doorenbos, B. & Veloso, M. M. (1993). Knowledge organization and the utility problem. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning.*

Etzioni, O. (1990). Why Prodigy/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 916–922.

Fattah, Y. E. & O'Rorke, P. (1993). Explanation-based learning for diagnosis. *Machine Learning*, 13:35–70.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37.

Gratch, J. & Dejong, G. (1992). COMPOSER: A probabilistic solution to the utility prob-

lem in speed-up learning. In *Proceedings of the Tenth National Conference on Aritificial Intelligence*, pages 235–240.

Greiner, R. (1991). Finding optimal derivation strategies in redundant knowledge bases. *Artificial Intelligence*, 50(1):95–115.

Greiner, R. & Jurisica, I. (1992). A statistical approach to solving the EBL utility problem. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 241–248.

Haley, P. V. (1987). Real-time for Rete. In *Proceedings of ROBEXs'87: The Third Annual Workshop on Robotics and Expert Systems*.

Katukam, S. & Kambhampati, S. (1994). Learning explanation-based search control rules for partial order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 582–587.

Keller, R. M. (1983). Learning by re-expressing concepts for efficient recognition. In *Proceedings of the National Conference on Artificial Intelligence*, pages 182–186.

Kim, J. (1996). *Bounding the Cost of Learned Rules: A Transformational Approach*. PhD thesis, University of Southern California.

Kim, J. & Rosenbloom, P. S. (1993). Constraining learning with search control. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 174–181.

Kim, J. & Rosenbloom, P. S. (1995). Mapping explanation-based learning onto Soar: The sequel. Technical Report :Transformation analyses of learning in SOAR. ISI/RR-95-4221, Information Sciences Institute and Computer Science Department University of Southern California.

Kim, J. & Rosenbloom, P. S. (1996). Learning efficient rules by maintaining the explanation structure. In *Proceedings of the Thirteenth National conference on Artificial Intelligence*, pages 763–770.

Laird, J. E., Congdon, C. B., Altmann, E., & Doorenbos, R. (1993). *Soar User's Manual: Version 6*, 1 edition.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1985). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Overgeneralization during knowledge

compilation in Soar. In *Proceedings of the Workshop on Knowledge Compilation*, pages 46–57.

Lee, H. S. & Schor, M. I. (1992). Match algorithms for generalized Rete networks. *Artificial Intelligence*, 54:249–274.

Luchins, A. S. (1942). Mechanization in problem solving. *Psychological Monographs*, 54.

Markovitch, S. & Scott, P. D. (1993). Information filtering : Selection mechanism in learning systems. *Machine Learning*, 10(2):113–151.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569.

Minton, S. (1993). Personal communication.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization – a unifying view. *Machine Learning*, 1(1):47–80.

Mooney, R. J. & Bennett, S. W. (1986). A domain independent explanation-based generalization. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 551–555.

Mostow, D. J. (1983). Machine transformation of advice into a heuristic search procedure. In R. Michalski, J. C. & Michell, T., (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA, Tioga Press. In press.

Prieditis, A. E. & Mostow, J. (1987). PROLEARN: Towards a Prolog interpreter that learns. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 494–498.

Rosenbloom, P. S. & Laird, J. E. (1986). Mapping explanation-based generalization onto Soar. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 561–567, Philadelphia. AAAI.

Rosenbloom, P. S., Laird, J. E., & Newell, A., (Eds.) (1993). *The Soar Papers: Research on Integrated Intelligence*. Cambridge, MA, MIT Press.

Rosenbloom, P. S., Laird, J. E., Newell, A., & McCarl, R. (1991). A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1-3):289–325.

Scales, D. J. (1986). Efficient matching algorithms for the Soar/Ops5 production system. Technical Report KSL-86-47, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Segre, A. & Elkan, C. (1994). A high-performance explanation-based learning algorithm. *Artificial Intelligence*, 69:1–50.

Shavlik, J. W. (1990). Aquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5:39–70.

Shell, P. & Carbonell, J. (1991). Empirical and analytical performance of iterative operators. In *The 13th Annual Conference of The Cognitive Science Society*, pages 898–902. Lawrence Erlbaum Associates.

Smith, D. E. & Genesereth, M. (1985). Ordering conjunctive queries. *Artificial Intelligence*, 26:171–215.

Subramanian, D. & Feldman, R. (1990). The utility of EBL in recursive domain theories. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 942–949.

Tambe, M., Johnson, W. L., Jones, R. M., Koss, F., Laird, J. E., Rosenbloom, P. S., & Schwamb, K. B. (1995). Intelligent agents for interactive simulation environments. *AI Magazine*, 16:15–39.

Tambe, M., Kalp, D., Gupta, A., Forgy, C. L., Milnes, B. G., & Newell, A. (1988). Soar/PSM-E: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146–160.

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

Tambe, M. & Rosenbloom, P. S. (1993). On the masking effect. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 526–533.

Tambe, M. & Rosenbloom, P. S. (1994). Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, 68(1):155–199.

Zelle, J. & Mooney, R. (1993). Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1106–1111.