University of Wisconsin Computer Sciences Technical Report 876 (September 1989)

Acquiring Recursive and Iterative Concepts with Explanation-Based Learning

Jude W. Shavlik

Department of Computer Sciences
University of Wisconsin - Madison
shavlik@cs.wisc.edu

(Submitted to *Machine Learning*)

Abstract

In *explanation-based learning*, a specific problem's solution is generalized into a form that can be later used to solve conceptually similar problems. Most research in explanation-based learning involves relaxing constraints on the variables in the explanation of a specific example, rather than generalizing the *graphical structure* of the explanation itself. However, this precludes the acquisition of concepts where an iterative or recursive process is implicitly represented in the explanation by a fixed number of applications. This paper presents an algorithm that generalizes explanation structures and reports empirical results that demonstrate the value of acquiring recursive and iterative concepts. The BAGGER2 algorithm learns recursive and iterative concepts, integrates results from multiple examples, and extracts useful subconcepts during generalization. On problems where learning a recursive rule is not appropriate, the system produces the same result as standard explanation-based methods. Applying the learned recursive rules only requires a minor extension to a PROLOG-like problem solver, namely, the ability to explicitly call a specific rule. Empirical studies demonstrate that generalizing the structure of explanations helps avoid the recently reported negative effects of learning.

Keywords: explanation-based generalization, generalizing explanation structures, generalizing to N, generalizing number, utility of learning, operationality versus generality

1. Introduction

Many real-world concepts involve an indefinite number of components, and many real-world plans involve an unbounded number of operations. For example, physical laws such as momentum and energy conservation apply to arbitrary numbers of objects, constructing towers of blocks requires an arbitrary number of repeated stacking actions, and setting a table involves places for differing numbers of guests. However, any specific example of such concepts will only contain a fixed number of actions or components. Systems that learn from examples must be able to detect and correctly generalize repeated portions of their training instances. In some cases, the number of repetitions itself should be the subject of generalization; in others it is inappropriate to alter the number of repetitions. Explanation-based learning (EBL) (DeJong & Mooney, 1986; Mitchell, Keller, & Kedar-Cabelli, 1986) provides an approach to this issue. In this type of learning, abstracting the solution to a specific problem produces a general solution applicable to conceptually similar problems. The generalization process is driven by the *explanation* of why the specific solution works. Knowledge about the domain lets a learner develop and then generalize this explanation. The explanation of repeated portions of a solution dictates when it is valid and proper to generalize the number of times they occur.

This paper addresses the important issue in EBL of *generalizing to N* (Cheng & Carbonell, 1986; Cohen, 1988; Prieditis, 1986; Riddle, 1989; Shavlik, in press; Shavlik & DeJong, 1985, 1987; Shell & Carbonell, 1989). This involves generalizing such things as the number of entities involved in a concept or the number of times some action is performed. Previous research on explanation-based learning has largely ignored the generalization of number. Instead, it has focused on changing constants into variables and determining the general constraints on those variables without significantly altering the underlying graphical structure of the explanation. However, this precludes acquisition of concepts in which a general iterative or recursive process is implicitly represented by a fixed number of applications in the specific problem's explanation. A system that possesses the ability to generalize the graphical structure of explanations, adding additional applications of inference rules where appropriate, can learn recursive and iterative concepts from a specific example. This article presents such a system.

To see the need for generalizing explanation structures, consider the LEAP system (Mitchell, Mahadevan, & Steinberg, 1985), an early application of explanation-based learning. The system observes an example of using *NOR* gates to compute the Boolean *AND* of two *OR*'s, and it discovers that the technique generalizes to computing the Boolean *AND* of any two inverted Boolean functions. However, LEAP cannot generalize this technique to let it construct the *AND* of an arbitrary number of inverted Boolean functions using a multi-input *OR* gate. The system cannot do this even if its initial background knowledge includes the general version of DeMorgan's Law and the concept of multi-input *NOR* gates. Generalizing the number of functions requires alteration of the original example's explanation.

Ellman's (1985) system also illustrates the need for generalizing number in explanation-based learning. From an example of a four-bit circular shift register, his system constructs a generalized design for an arbitrary four-bit permutation register, but again, it cannot produce a design for an *N*-bit circular shift register. As Ellman points out, such generalization, though desirable, cannot be done using the technique of changing constants to variables.

Repetition of an action is not a sufficient condition for generalization to *N* to be appropriate. For instance, generalizing to *N* is necessary if one observes a previously unknown method of moving an obstructed block, but not when one sees a toy wagon being built for the first time. The initial states of these two problems appear in Figure 1. Suppose a learning system observes an expert achieving the desired states, and consider what general concept should be acquired in each case. In the first example, the expert wishes to use a robot manipulator to move a block that has four other blocks stacked in a tower on top of it. The manipulator can pick up only one block at a time. The expert's solution is to move each of the four blocks in turn to some other location. After the underlying block has been cleared, it is moved. In the second example, the expert wishes to construct a movable rectangular platform, one that is stable while supporting any load whose center of mass is over the platform. Given the platform, two axles, and four wheels, the expert's solution is to first attach each of the axles to the platform, then to select each of the four wheels in turn and mount it on an axle protrusion.

This comparison illustrates an important problem in explanation-based learning. Generalizing the block unstacking example should produce a plan for unstacking *any* number of

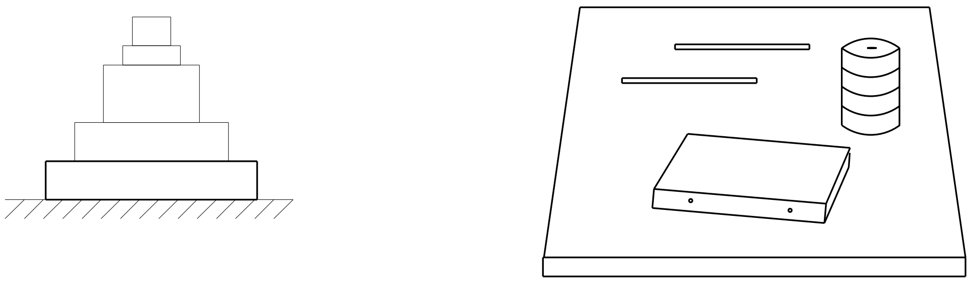


Figure 1. Initial states for two sample problems.

obstructing blocks, not just four as observed. However, in the wagon-building example the number "four" should not be generalized. It makes no difference whether the system experiences a pile of five, six, or 100 wheels, because exactly four wheels are needed to fulfill the functional requirements of a stable wagon.

Standard explanation-based learning algorithms (e.g., Fikes, Hart, & Nilsson, 1972; Hirsh, 1987; Kedar-Cabelli & McCarty, 1987; Mooney & Bennett, 1986) and similar algorithms for chunking (Laird, Rosenbloom, & Newell, 1986) cannot treat these cases differently. These methods, possibly after pruning the explanation to eliminate irrelevant parts, replace constants with constrained variables. They cannot significantly augment the explanation during generalization. Thus, the *building-a-wagon* type of concept will be correctly acquired but the *unstacking-to-move* concept will be undergeneralized. Their acquired schema will have generalized the identity of the blocks so that the target block need not be occluded by the same four blocks as in the example. Thus, any four obstructing blocks can be unstacked, but there must be exactly four blocks.[1] Unstacking five or more blocks is beyond the scope of the acquired concept.

Of course one could simply define the scope of EBL-type systems to exclude the *unstacking-to-move* concept and similar ones, but this would be a mistake for three reasons. First, the need for augmenting explanations is ubiquitous. Many real-world domains manifest it in one form or another. Second, if one simply defines the problem away, the resulting system could never guarantee that any of its concepts were as general as they should be. Even when such a system correctly constructed a concept like the *building-a-wagon* schema, it could not know that it had generalized properly. The system could not tell which concepts fell within its scope and which did not. Third, there is recent psychological evidence (Ahn, Mooney, Brewer, & DeJong, 1987) that people can generalize number on the basis of one example.

One may argue that the fault for not properly generalizing lies with the explanation module. If the explainer used a vocabulary involving recursion, then it might not be necessary to alter the graphical structure of the explanation. However, such an approach places a much larger burden on the explanation module, as well as on the domain theory writer. Constructing explanations is a demanding, often intractable, task (Mitchell *et al.*, 1986). Generalization is more focused and less computationally intensive; hence it makes sense to shift as much of the burden of learning onto this module. If they are to scale to larger problems, EBL systems must not expect the explanation module to do more than narrowly explain the solution to the specific problem at

---

[1] The SOAR system (Laird *et al.*, 1986) would seem to acquire a number of concepts that together are slightly more general. In addition to a new operator for moving four blocks, the system would acquire new operators for moving three blocks, two blocks, and one block, but not for five or more. Anderson's (1986) knowledge-compilation process would acquire a similar set of rules.

hand. It is the generalization module's responsibility to determine how broadly applicable the solution is.

Observations of repeated rule or operator applications indicate that generalizing the number of rules in the explanation may be appropriate. However, such observations alone are insufficient. Number generalization is desirable only if there exists a certain recursive structural pattern, in which each application achieves preconditions for the next. In stacking blocks, for example, the same sort of repositioning of blocks occurs repeatedly, each building on the last. This article adopts the vocabulary of predicate calculus to investigate this notion of structural recursion. The desired form of recursion is manifested as repeated application of inference rules in such a manner that a portion of each consequent is used to satisfy some of the antecedents of the next application. This means that number generalization will not occur solely because some rule appears repeatedly in an explanation. Instead, the repetitions must be in a goal-subgoal relationship.

Generalizing number, like more traditional generalization in explanation-based learning, results in the acquisition of a new inference rule. Unlike traditional methods, it generates a rule that describes the situation after one has made an indefinite number of world changes or other inferences. Each such rule subsumes a potentially infinite class of rules that standard explanation-based generalization techniques would acquire. Thus, number-generalized rules can dramatically improve storage efficiency, increase the expressive power of the system, and, as shown later, improve the system's performance efficiency.

Section 2 presents an algorithm for generalizing the structure of explanations, illustrates the method with an example, and empirically compares its behavior to a standard EBL algorithm. Section 3 describes some extensions to the basic algorithm that increase the efficiency of the rules it acquires and presents empirical studies involving a second domain. The final sections discuss related work and describe several open research problems.

2. The Basic BAGGER2 Algorithm

Unlike most earlier approaches to explanation-based learning, the BAGGER2 algorithm (Building Augmented Generalizations by Generating Extended Recurrences) is capable of generalizing explanation structures. This system is a successor to an earlier structure-generalizing EBL system (Shavlik, in press; Shavlik & DeJong, 1987) that learned iterative concepts (manifested as linear chains of rule applications). Unlike its predecessor, BAGGER2 is capable of acquiring recursive concepts involving arbitrary tree-like applications of rules; in addition it can produce multiple generalizations to *N* from a single example and can integrate the results of multiple examples.

This section describes the basic components of BAGGER2. The first subsection compares it to standard explanation-based generalization. The next presents algorithmic details and a correctness proof. Next, a circuit implementation task illustrates the algorithm. Following that appear a discussion of learning from multiple examples and a description of how learned rules are used during future problem solving. Finally, an empirical study demonstrates BAGGER2's efficacy.

2.1. Comparison to Standard Explanation-Based Generalization

BAGGER2 extends Mooney and Bennett's (1986) EGGS algorithm, a standard domain-independent method for explanation-based generalization. Both techniques assume that, in the course of solving a problem, the solver interconnects a collection of pieces of general knowledge (e.g., inference rules, rewrite rules, or plan schemata), using unification to insure compatibility. The generalizers then produce an *explanation structure* (Mitchell *et al.*, 1986) from the specific problem's explanation. To build the explanation structure, they first strip away the details of the specific problem and then replace each instantiated rule in the explanation with a copy of the original general rule. If the same general rule is used multiple times, its variables are renamed each time it appears in the explanation structure. This prevents spurious equalities among variables in the explanation structure.

EGGS determines the most general unifier that lets the solver connect the explanation structure's general pieces of knowledge, and produces a new composite knowledge structure which contains the unifications that must hold in order to combine the knowledge pieces in the given way. If one assumes tree-structured explanations, then satisfaction of the leaf nodes implies that the root (goal) node will also be satisfied. There is no need to reason again about combining the pieces of knowledge to achieve the goal. The problem solver may have performed a substantial amount of work constructing the original solution, following many unsuccessful paths and then backtracking. The new knowledge structure can lead more rapidly to solutions in the future, because it avoids the unsuccessful paths and eliminates the need to rederive the intermediate conclusions. However, note that EGGS does not change the graphical structure of the explanation. If some process is repeated three times in the specific problem's explanation, it will be repeated exactly three times in the rule EGGS acquires.

In contrast, BAGGER2 generalizes explanation structures by looking for repeated, inter-dependent substructures in an explanation. Figure 2 schematically presents this process. Assume that in explaining how a goal is achieved, the same general subproblem (P) arises several times. The full explanation can be divided into several qualitatively different portions. First, there are the subexplanations in which an instantiation of P is supported by the explanations of other instantiations of the general problem P. In the figure, these are the

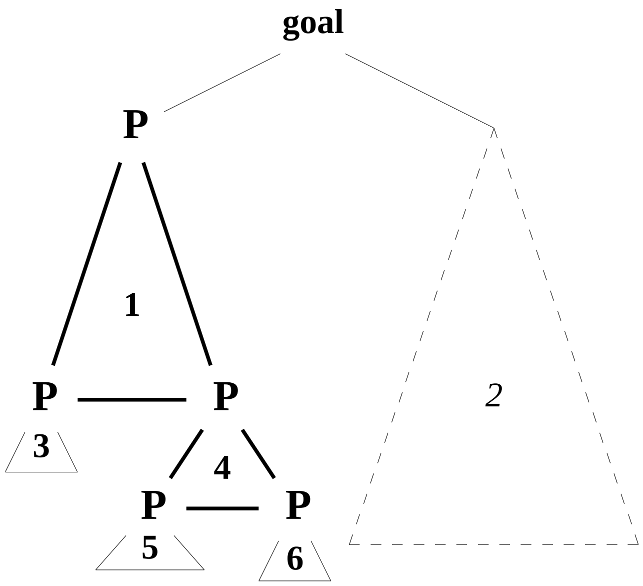


Figure 2. Partitioning the structure of an explanation.

subexplanations marked 1 and 4. Second, there are the subexplanations in which an instantiation of *P* is explained without reference to another instantiation of itself. These are the subexplanations labeled 3, 5, and 6. Finally, there are the portions not involving *P* (subexplanation 2).

The explanation in Figure 2 can be viewed as the trace of a recursive process. This is exactly what one must recognize in the explanation of a specific example in order to learn a recursive or iterative concept. The generalizations of subexplanations 1 and 4 form the recursive portion of the concept, whereas the generalizations of subexplanations 3, 5, and 6 produce the termination conditions. BAGGER2 partitions explanations into groups, as Figure 2 illustrates, from which it produces a new recursive concept.

Roughly speaking, BAGGER2 produces the following two rules from Figure 2's explanation.

$$goal \leftarrow P \wedge gen_2 .$$

$$P \leftarrow gen_3 \vee gen_5 \vee gen_6 \vee recursive{-}gen_1 \vee recursive{-}gen_4 .$$

To achieve the goal, a problem solver must satisfy the recursive subgoal *P* and whatever general preconditions subexplanation 2 requires. The solver can satisfy *P* by satisfying the general preconditions of any of the non-recursive or recursive subexplanations; the generalizations of the recursive subexplanations lead to recursive calls to subgoal *P*.

2.2. Algorithmic Details and Correctness Proof

Table 1 contains the BAGGER2 generalization algorithm. Although the algorithm appears here in a pseudo-code, the actual implementation is written in COMMON LISP. The remainder of this subsection elaborates the pseudo-code and presents a theorem about its correctness.

The BAGGER2 approach assumes that explanations are derivation *trees*, which are structures that could be produced by a Horn clause theorem prover such as PROLOG. The algorithm starts at the root of the explanation. If the general consequent at the root appears elsewhere in the structure, then the method produces a recursive rule (called a *recurrence*) whose consequent is the root node. Otherwise, it collects the general version of the root node's antecedents and produces a new rule. Since a recurrence can also arise within an explanation structure, this discussion will assume that the root node does not directly lead to a recurrence.

As shown in Table 1, *CollectGeneralAntecedents* produces sufficient requirements for the consequent of a rule to hold. Ignoring for a moment the possibility of recurrences being constructed, this entails traversing through the explanation structure and stopping at *operational* (Keller, 1988) nodes. Operational nodes are antecedents somehow judged to be easily satisfied, for example, because they are satisfied by a problem-specific fact. Along the way, the function collects all the unifications necessary to connect the rules in the explanation structure, thus eliminating the need to check these when the acquired rule is later applied. This portion of the algorithm is merely a rehash of EGGS. Hence, when BAGGER2 detects no potential generalizations to *N*, it produces the same result as EGGS.

More interesting events occur when BAGGER2 detects a potential recurrence. This is done by seeing if a unifiable version of the general antecedent appears in its own derivation (e.g., the *P*'s in Figure 2). If so, *ProduceRecurrence* partitions the explanation structure headed by the general antecedent into two types of subexplanations: *terminal proofs* where a unifiable version of the antecedent does not appear in its proof, and *recursive proofs* where at least one does. In the recursive proofs, the function replaces the recursive subexplanations by a call to the recurrence being constructed. These calls contain the term that must be unified with the consequent of the recurrence. Hence, when cutting out Figure 2's subexplanation 1, the function removes subexplanations 3 and 4 and replaces them by a call to the recurrence whose consequent is *P*. Notice that the subexplanations are non-overlapping.

Once *ProduceRecurrence* produces the subexplanations, it generalizes each by calling BAGGER2. This means that another recurrence may be found within a subexplanation, allowing multiple generalizations to *N* in a single example. When generalizing the subexplanations, the function collects the necessary unifications between the root of the subexplanation and the

Table 1. The BAGGER2 generalization algorithm.

Procedure BuildNewBAGGER2Rule(*goal-node*) /* Generalize the explanation headed by this node. */
  Let *consequent* be the consequent of the goal node.
  If *consequent* is supported by a term that unifies with it,
  Then return *ProduceRecurrence(goal-node)*,
  Else let *antecedents* be *CollectGeneralAntecedents(goal-node)*
    and return the rule *consequent* ← *antecedents.*

Procedure CollectGeneralAntecedents(*node*) /* Collect the generalized version of the antecedents of *node*. */
  Let *result* be the empty set.
  For each direct *antecedent* of *node*,
    If it is operational or a call to a recurrence,
      Then conjunctively add it to *result*,
    Else if it is supported by a term that unifies with it, /* Found a potential recurrence. */
      Then conjunctively add *ProduceRecurrence(antecedent)* to *result*,
    Else if it is directly supported by the consequent of a rule,
      Then:Let *consequent* be the rule's consequent.
        Conjunctively add to *result* the equalities that must hold to unify *antecedent* and *consequent*.
        Conjunctively add *CollectGeneralAntecedents(consequent)*.
    Else return *false*. /* Reached a non-operational leaf node. */
  Return *result*.

Procedure ProduceRecurrence(*node*) /* Produce a BAGGER2 recurrence from the subexplanation headed by *node*. */
  Let *consequent* be the root of *node*.
  Let *antecedents* be the empty set.
  For each terminal and recursive *subproof* supporting *node*: /* Look at alternative ways of satisfying *node*. */
    Let *subconsequent* be the root of *subproof*.
    Let *disjunct* contain the equalities that must hold to unify *subconsequent* and *node*.
    Conjunctively add *CollectGeneralAntecedents(subconsequent)* to *disjunct*.
    Disjunctively add *disjunct* to *antecedents*.
  Construct the recurrence *consequent* ← *antecedents* and return a call to it.

consequent of the recurrence under construction. Satisfying these unifications insures that the general solution in the subexplanation applies to the recurrence's consequent. The function disjunctively combines the generalizations of the subexplanations and produces a recurrence.

Notice that, rather than only learning a single rule from an explanation, BAGGER2 also produces several useful subrules (the recurrences). The detection of recurrences provides a useful decomposition of explanations. Because recurrences are separate entities from the rule produced for the full explanation, they support transfer of the results learned during one task to the performance of another, provided the two tasks involve common subtasks. This separation also supports learning from multiple examples. If the system encounters a new method for satisfying the consequent of a recurrence, it can merge the new method with the previous disjuncts[2], as discussed further in Section 2.4.

Before BAGGER2 produces a new rule, it removes redundant antecedents and reorders the others to increase the efficiency of future retrievals. In recurrences, if an antecedent appears in every terminal disjunct, it can be removed from the recursive disjuncts, provided it is independent of the variables in the recurrence's consequent. Dropping them from the recursive disjuncts is valid because these antecedents will be checked when retrieval reaches a terminal disjunct.

Assuming that explanations are logical proofs, the BAGGER2 algorithm can be proved correct, as shown in the following theorem.

Theorem 1. The basic BAGGER2 algorithm is *sound*, in that the rules it learns will never derive anything that cannot be derived by the initial domain theory.

Proof. Consider the explanation in Figure 2. Basically, one must show that the algorithm properly generalizes the subexplanations and that the method for combining subexplanations preserves soundness. Assume for the moment that there are no potential recurrences other than P within Figure 2's explanation. Since the EGGS algorithm has been proven sound (Mooney, in press), BAGGER2 will correctly generalize each of the numbered subexplanations. The result from subexplanation 2 is handled the same by BAGGER2 and EGGS; it will not be considered further. Next, consider the recursive portion of the explanation. To derive the generalized goal, the uppermost P must be satisfied using the generalization of either a terminal or a recursive subproof. The soundness of a derivation using only a terminal proof follows directly from the proof for EGGS and the fact that the algorithm maintains the unifications necessary to insure that

[2] This may lead to poor performance if too many disjuncts are learned. The user can decide when a concept is sufficiently learned and tell the system to "freeze" all of its recurrences. After that, new recurrences will be constructed even if they have the same consequent as an existing one.

the generalized consequent of a subexplanation matches *P*. A derivation using a recursive proof must terminate by using the results of terminal proofs. Since these final derivation steps are sound, and soundness between steps is maintained by checking the necessary inter-step unifications, the soundness of recursive derivations follows inductively.

The case for embedded recurrences also follows inductively. After partitioning an explanation there are no embedded *P*'s in the subexplanations, since those at the leaves are converted to a recurrence call. Hence, each subdivision of the explanation reduces the number of potential recurrences. The deepest subexplanation contains only one recurrence, and BAGGER2 correctly generalizes it according to the argument in the previous paragraph. Encapsulating subexplanations are also correctly generalized, by induction. To see this, assume correctness for i levels of embedded recurrences. The argument in the previous paragraph can now be applied to demonstrate correctness for the level $i+1$.

2.3. A Circuit Implementation Example

As an example, consider the application of BAGGER2 in the domain of circuit implementation. Examples in this simple domain, commonly used to illustrate EBL techniques, clearly show the weaknesses of standard EBL methods and demonstrate how BAGGER2 overcomes them. Replication of structure is an important operation in the circuit domain.

The rules in Table 2 determine how to implement a circuit depending on the type of gates available. PROLOG notation is used, except that leading question marks indicate variables. The term *ImplementBy(?x, ?y)* means that circuit *?y* can be used to implement design *?x*. Assuming one can use only AND and NOT gates, Figure 3 shows a circuit whose implementation requires repeated application of DeMorgan's Law. The rules in the table can be used to provide an explanation of how to accomplish this task.

It is instructive to compare the behavior of EGGS and BAGGER2 on this problem. If EGGS is applied to the resulting explanation, the rule in Table 3 results. Notice that this rule not only requires a fixed number of inputs, but also a fixed topology, that of Figure 3. Clearly the explanation structure needs to be generalized. The result produced by BAGGER2 appears in Table 4. In this problem, the full explanation leads to a single recurrence, which involves four disjuncts. The first applies when only a single application of DeMorgan's Law is necessary, with AND gates being necessary if the resulting circuit is to be implemented. The remaining three disjuncts are recursive. The second and third disjuncts apply when one input is a wire, in which case the rule recurs on the other input. In the final disjunct, recursion is needed for both inputs.

The notation in Tables 3 and 4 merits some discussion. For instance, the capitalized *Or*'s, *And*'s, and *Nots*'s refer to gates in the circuit domain; they are not part of the rule description

Table 2. Initial rules for the domain of circuit implementation.

Rule	Description
ImplementBy(Not(Not(?x)),?y) $\leftarrow$ ImplementBy(?x,?y).	Can eliminate double negations.
ImplementBy(Not(And(?x,?y)),Nand(?a,?b)) $\leftarrow$ HaveNands $\wedge$ ImplementBy(?x,?a) $\wedge$ ImplementBy(?y,?b).	Can use *nand*'s to implement negated *and*'s.
ImplementBy(Not(?x),Nand(?y,1)) $\leftarrow$ HaveNands $\wedge$ ImplementBy(?x,?y).	Can use *nand*'s to implement *not*'s.
ImplementBy(And(?x,?y),Nand(Nand(?a,?b),1)) $\leftarrow$ HaveNands $\wedge$ ImplementBy(?x,?a) $\wedge$ ImplementBy(?y,?b).	Can use *nand*'s to implement *and*'s.
ImplementBy(Not(?x),Not(?y)) $\leftarrow$ HaveNots $\wedge$ ImplementBy(?x,?y),	Can use *not*'s to implement if they are available.
ImplementBy(Or(?x,?y),Or(?a,?b)) $\leftarrow$ HaveOrs $\wedge$ ImplementBy(?x,?a) $\wedge$ ImplementBy(?y,?b).	Can use *or*'s to implement if they are available.
ImplementBy(Or(?x,?y),Nand(?a,?b)) $\leftarrow$ HaveNots $\wedge$ HaveNands $\wedge$ ImplementBy(Not(?x),?a) $\wedge$ ImplementBy(Not(?y),?b).	A version of DeMorgan's Law.
ImplementBy(Not(Or(?x,?y)),And(?a,?b)) $\leftarrow$ HaveAnds $\wedge$ ImplementBy(Not(?x),?a) $\wedge$ ImplementBy(Not(?y),?b).	Alternative version of DeMorgan's Law.
ImplementBy(?wire,?wire) $\leftarrow$ Wire(?wire).	Wires are already implemented.

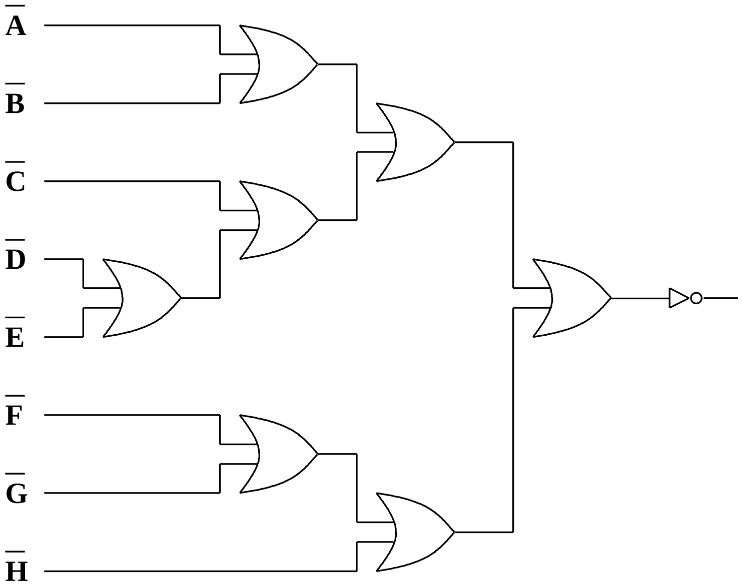


Figure 3. A sample circuit design to be implemented using only AND gates.

Table 3. The rule acquired by EGGS for the circuit problem in Figure 3.

```
ImplementBy(Not(Or(Or(Or(Not(?G6),  Not(?G9)),
                      Or(Not(?G16), Or(Not(?G23), Not(?G26)))),
                   Or(Or(Not(?G37), Not(?G40)),
                      Not(?G43)))),
            And(And(And(?G6,  ?G9),
                    And(?G16, And(?G23, ?G26))),
                And(And(?G37, ?G40),
                    ?G43)))
    ←
HaveAnds   ∧  Wire(?G43) ∧  Wire(?G6)  ∧  Wire(?G9) ∧  Wire(?G16) ∧
Wire(?G37) ∧  Wire(?G40) ∧  Wire(?G23) ∧  Wire(?G26).
```

Table 4. The recursive rule acquired by BAGGER2 for the circuit problem in Figure 3.

To satisfy: ImplementBy(Not(Or(?v1, ?v2)), And(?v3, ?v4))

One of the following must hold:

?v1 matches Not(?v3) ∧ ?v2 matches Not(?v4) ∧ /* no more gates */
HaveAnds ∧ Wire(?v3) ∧ Wire(?v4).

or

?v1 matches Or(?e1, ?e2) ∧ ?v2 matches Not(?v4) ∧ /* no gates on right */
?v3 matches And(?e3, ?e4) ∧ Wire(?v4) ∧
recursively satisfy ImplementBy(Not(Or(?e1, ?e2)), And(?e3, ?e4)).

or

?v1 matches Not(?v3) ∧ ?v2 matches Or(?e1, ?e2) ∧ /* no gates on left */
?v4 matches And(?e3, ?e4) ∧ Wire(?v3) ∧
recursively satisfy ImplementBy(Not(Or(?e1, ?e2)), And(?e3, ?e4)).

or

?v1 matches Or(?e1, ?e2) ∧ ?v2 matches Or(?e3, ?e4) ∧ /* gates on both sides */
?v3 matches And(?e5, ?e6) ∧ ?v4 matches And(?e7, ?e8) ∧
recursively satisfy ImplementBy(Not(Or(?e1, ?e2)), And(?e5, ?e6)) ∧
recursively satisfy ImplementBy(Not(Or(?e3, ?e4)), And(?e7, ?e8)).

language. The *matches* operator unifies its two arguments. As an extension to PROLOG, direct calls to recurrences are permitted in order to satisfy a goal. Finally, BAGGER2 renames the variables in recurrences; variables starting with *V* appear in the consequent, whereas those starting with *E* are local variables.

The rule BAGGER2 learns is a general version of DeMorgan's Law; it converts the negation of an *N*-input OR gate into an *N*-input AND gate. It applies to a much larger class of problems than does the rule learned by EGGS. Notice that the acquired recurrence does not refer to any of the initial rules. Thus, it is self-contained and is topologically similar to a recursive LISP function. The consequent specifies the parameters, and the antecedents form something like a LISP conditional statement. This "function" is produced from a collection of simple declarative PROLOG-like rules, which are called explicitly rather than determining which rules in a large rule base unify with the current consequent. Hence, BAGGER2 provides a way to transform a simple, but inefficient, logic program into another program stated in a more efficient language. Shavlik and Maclin (1988) investigate the application of the approach to the problem of acquiring programs. Section 4 further discusses the relationship between BAGGER2 and automatic programming.

2.4. Learning from Multiple Examples

As mentioned earlier, BAGGER2 is capable of learning from multiple examples. Recurrences are disjunctive subrules, and hence are prime candidates for enhancement during additional learning. When generalization of a new explanation produces a basic recurrence that has the same consequent as a previous recurrence, the generalizer merges the disjuncts in the new recurrence with those of the existing one. As an illustration, assume that the training example in Figure 3 was simpler, and the second recursive case in Table 4 was not encountered. A future training example involving the missing case would be needed in order to completely learn Table 4's general version of DeMorgan's Law.

Commonly in explanation-based learning, additional examples lead to disjuncts in acquired rules (e.g., Cohen, 1988; Kedar-Cabelli, 1986; Minton, 1989). There are two important aspects of BAGGER2's approach to learning from multiple examples. One, the system need not learn disjuncts during the achievement of the same goal. For instance, a technique learned during the building of an arch can improve what was previously learned about building towers. Two, BAGGER2 does not require the explanation from which the existing recurrence arose, so there is no need to save old explanations. Hence, learning can proceed incrementally without requiring full memory of all past experiences.

Learning from multiple examples produces much the same effect as initially learning from a more complicated example. Since BAGGER2 breaks up explanations and merges together similar generalizations (see Figure 2), in terms of generalization it does not matter if the subexplanations come from one example or many. However, there is an advantage of learning from multiple examples in explanation-based learning. A complicated concept can be acquired by observing several simple versions of the concept. This can greatly reduce the load on the agent that produces the training examples and also simplifies the explanation process. Looking at this a different way, if a rule is initially incompletely learned because a training example is too simplistic, later experiences can refine the rule.

2.5. Problem Solving with Acquired Rules

Later sections of this article empirically compare the performance of EGGS and BAGGER2 in two sample domains. This section presents how they apply their learned rules to new problems.

The basic problem solver is a COMMON LISP implementation of PROLOG, which satisfies goals using depth-first backward chaining. Besides returning variable bindings, it returns a proof tree that explains how it satisfied the goal. BAGGER2's recurrences require a slight extension to PROLOG, namely the ability to explicitly call a specific rule. When it calls a particular rule to satisfy a given subgoal, the problem solver applies no other rules to that subgoal upon backtracking. To avoid wasted effort, when first calling a recurrence the solver checks those terms in the recurrence's terminal disjuncts that are independent of the variables in the recurrence's consequent. Such antecedents are unaffected by calls to the recurrence, and if they cannot be satisfied the problem solver does not call the recurrence.

Mooney (1989) has empirically shown, for depth-first problem solvers, that allowing arbitrary chaining of learned rules can lead to worse performance than not learning at all. As a result, the problem solver never combines learned rules when trying to solve a problem. To be successful, a learned rule must solve a problem completely by itself.

Situation calculus planning is used in a second sample domain (described later). One potential problem with situation calculus in a depth-first problem solver is the possibility of infinite regression. The solver can hypothesize an unbounded number of potential intermediate states when trying to achieve a goal. To avoid this, it places an upper bound on the number of actions that can appear in the situation calculus plans it produces. Specifically, in the blocks world domain the depth-first problem solver limits its search to plans involving no more actions than there are blocks in the current scene.

The EGGS and BAGGER2 systems index rules acquired during training according to the goal each rule achieves. When multiple rules achieve the same goal, the order they are applied during problem solving depends on the learner. EGGS organizes situation calculus rules according to the number of actions they specify. When facing a new problem, it first tries to apply the rules that involve one action; if this fails, it tries rules involving two actions, and so forth. Shavlik (in press) demonstrates the efficiency of this strategy, which is similar to *iterative deepening* (Korf, 1985); it works well because the effort required to produce a plan can depend exponentially on the number of actions involved. For rules involving the same number of actions or not expressed in situation calculus, EGGS tries rules in the order it learned them, while BAGGER2 tries the most recently learned rules first. Shavlik (in press) demonstrates these strategies are preferable for each system when problems are randomly generated, for the following reasons. Probabilistically, earlier rules result from more typical examples, while later rules result from examples that are less likely to occur again soon. For EGGS, it is best to first try the rules that result from the most probable situations. However, BAGGER2 often learns more from the more complicated, though less likely, later examples and the acquired rule usually covers the simpler, previous examples.

2.6. Empirical Studies of the Basic Algorithm

One may question the desirability of generalizing explanation structures. Such learning leads to more general rules, but because the resulting rules are more complicated, applying them entails more work. This question involves the relationship between the *operationality* and *generality* of acquired rules (DeJong & Mooney, 1986; Keller, 1988; Mitchell *et al.*, 1986;

Segre, 1987; Shavlik, DeJong, & Ross, 1987). Experiments reported in this section investigate whether it is better to learn a more general recursive rule or better to individually learn the subsumed rules as they are needed. A secondary question focuses on whether explanation-based learning is worthwhile at all. As more is learned, the process might slow down a problem solver that uses the learned concepts (Fikes *et al.*, 1972; Minton, 1988).

Using the circuit implementation rules, this study compares three systems: BAGGER2, EGGS, and NO-LEARN - a system that does not learn any new rules. All three systems use the backward-chaining problem solver described in the previous section.

The experimental hypothesis of the first experiment is that BAGGER2 requires less training than EGGS to acquire a concept; for a given amount of training it will be more likely to solve a new problem. The independent variables in this study are the generalization algorithm used and the amount of training, while the dependent variable is the percentage of a novel examples solved using only a learned rule. In this study, problems consist of randomly generated designs of eight-input OR gates using binary OR's. The final output is negated; Figure 3 contains a sample design. As described in Section 2.1, the task is to implement this gate using only NOT and binary AND gates. An experimental run proceeds as follows. First, ten randomly-generated test problems are produced and saved. Next, 250 randomly-generated circuit designs are produced, and the two learners implement them. If they use more than one rule to do so, they generalize the resulting explanation and save the result. Periodically their performance is measured on the ten test problems, during which learning is turned off. There are ten experimental runs, each with a different initial random number; hence each point in this study is the mean of 100 measurements.

Figure 4 presents the number of test problems solvable using only each system's acquired rules. Clearly, BAGGER2 needs significantly fewer training examples to learn the concept than does EGGS. The reason is that BAGGER2 learns a rule that encodes a general strategy for repeatedly applying DeMorgan's Law to implement any network of binary OR gates, while EGGS learns separate rules for each possible layout. EGGS can only apply its learned rules to new problems that have the same topology as a previously seen one. This illustrates one of the strengths of a system that restructures explanations; since it recognizes repeated portions of explanations and generalizes the number of repetitions, it can require less training to acquire a concept.

Being able to apply learned rules to solve new problems is of little use if doing so requires a substantial amount of effort. The second study investigates the speed up achieved by using the explanation-based systems. This study's independent variables are the identity of the system and the size of the space of possible problems. The dependent variable is the average problem-solving time following learning. The primary hypothesis is that performing explanation-based learning is better than solving all problems by only resorting to the initial domain rules. The

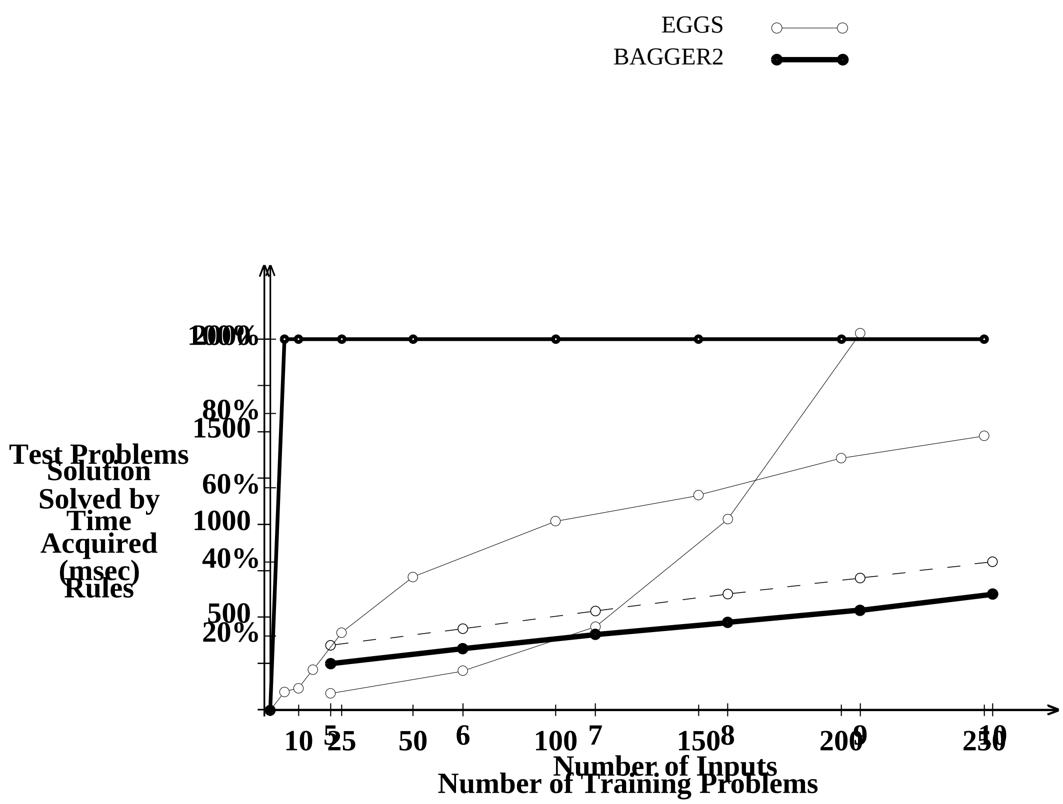
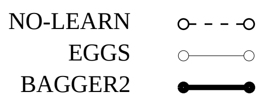


Figure 5. Mean solution time for circuit implementation as a function of problem size.

Figure 4. New circuit problems solved with acquired rules as a function of amount of training.



secondary hypothesis is that BAGGER2's learned rules produce solutions faster than EGGS's rules. It is possible that the added complexity of recursive rules leads to longer solution times. Although EGGS must learn more rules than BAGGER2 to cover a concept involving recursion, a problem solver may be able to more efficiently search through these non-recursive rules for a solution.

To examine the hypothesis, each learning system is trained on OR circuit problems of various sizes (from five to ten inputs).[3] For each problem size, 1000 random circuits constitute the training set. Following training, all three systems attempt to solve the same ten test problems, with learning turned off. This is repeated ten times, each time with different test problems, so again the plotted points represent the means of 100 measurements. EGGS organizes

[3] The number of different ways to implement an N-input OR gate with binary gates is $\frac{(2N-2)!}{N!\,(N-1)!}$ (Jacobson, 1951, page 18).

its rules according to the number of inputs involved (i.e., in six groups) and only checks possibly relevant rules during problem solving. For EGGS, only the time spent on problems solved by a learned rule is recorded. Both this and the assumption about rule organization favor EGGS over BAGGER2. Figure 5 contains the mean solution time on the test problems. Learning recursive rules is always better than not learning at all. However, there are several places where EGGS' performance curve crosses another system's curve. When there are a small number of possible problems, EGGS does better than BAGGER2, however it soon becomes worthwhile to learn recursive rules. Note that after awhile, it would be better to not learn at all than to use EGGS. The locations in Figure 5 where curves cross merit discussion.

Explanation-based generalizers produce composite rules that capture useful combinations of other rules. Using the resulting generalization, a problem solver can solve many problems more quickly than without it; the solver need not spend time rediscovering the successful combination. However, as a system acquires more new rules, its problem-solving performance can *decrease*, because it may spend substantial time trying to apply learned rules that appear promising, but ultimately fail (Minton, 1988). This explains why EGGS performs worse than NO-LEARN when the space of possible circuit designs is large; the solver must try too many promising but unfruitful rules before succeeding. Because BAGGER2 can capture a concept in a single rule, a solver need not waste time searching through a collection of closely related rules looking for an appropriate one; for this reason BAGGER2 helps avoid the negative effects of learning Minton recently reported.

The results in Figure 5 can be better understood by considering the difference between the rules EGGS learns and those BAGGER2 learns. One can view the rules EGGS learns as fixed-sized templates, each of which contains a fixed number of variables to bind. Problem solving with these rules involves matching their preconditions to the current task; if the problem is similar to a previous one, the solver can apply the generalized version of the previous solution. It does not produce new solutions, but only determines if it can reuse an old one. Conversely, BAGGER2 learns a technique for *generating* new solutions, in which an arbitrary number of variables can be bound. By analyzing an explanation, it extracts an algorithm reflected in a specific problem's solution. This generative capability means it can express an unbounded number of templates, while still maintaining the efficiencies obtained by performing explanation-based learning.

In summary, these studies demonstrate that a structure-generalizing EBL algorithm such as BAGGER2 can learn rules that are both more general and more efficient than the corresponding ones EGGS learns. The ability to scale to larger problems is an important property of any learning system; as the space of possible problems grows, BAGGER2's edge over EGGS increases.

3. Improving the Efficiency of the Rules BAGGER2 Learns

The basic BAGGER2 algorithm produces valid generalizations of specific solutions, but the results produced can be often made more efficient without any loss of generality. This section presents three techniques for improving the efficiency of rules learned by the basic approach. These extensions to the basic method are particularly useful for generalizing actions in plans. Often portions of plans are iterative; a set of actions is repeated until a subgoal is achieved. The presented techniques can improve the efficiency of inherently iterative tasks; the blocks-world task of building towers of various heights illustrates them. An appropriate plan is to stack clear blocks until reaching the desired height. Unlike the circuit implementation solution, this is inherently an iterative strategy, in the sense that an action (or set of actions) is repeated until achieving some target. In addition, the circuit domain primarily involves repeated components, while building towers primarily involves repeated actions.

After describing the representation used for the tower-building task, this section uses tower building to illustrate the extensions. A second empirical study evaluates the extensions and compares BAGGER2, EGGS, and NO-LEARN in the blocks world.

3.1. A Tower Building Task

The situation calculus (McCarthy, 1963) is used in this section's example to reason about actions, in the style of Green (1969). In this formalism, predicates and functions whose values may change over time possess an extra argument that indicates the situation in which they are being evaluated. For example, rather than using the predicate *On(x,y)* to indicate that *x* is on *y*, the predicate *On(x,y,s)* is used to indicate that *x* is on *y* in situation *s*. In this framework, operators are represented as functions that map from one situation to another situation. For instance, the term *Do(Transfer(A,B,s0))* represents the situation that results from the initial situation upon transferring block *A* to block *B*. Problem solving involves transforming situations until a path from the initial situation to the goal situation is found.

As an example, consider the following situation calculus rule.

$$On(?x,\ ?y,\ Do(Transfer(?x,\ ?y),\ ?s)) \ \leftarrow\ AchievableState(Do(Transfer(?x,\ ?y),\ ?s)).$$

This rule formalizes one effect of a transfer. It says that if one can legally achieve the situation represented by the term *Do(Transfer(?x,?y),?s))*, then in this situation *?x* is on *?y*. A separate rule defines legal transfers. Tables 5 and 6 contain the rules for the tower-building domain.

Two types of inference rules are used: *inter-situational* rules, which specify attributes that a new situation will have after application of a particular operator, and *intra-situational* rules, which can embellish a problem solver's knowledge of a situation by specifying additional

Table 5. Some initial rules for the blocks domain.

Rule	Description
AchievableState(S0).	The initial state is always achievable. A state is achievable if legal application of some sequence of operators can lead to it.
AchievableState(Do(Transfer(?x,?y),?s)) ← AchievableState(?s) ∧ Liftable(?x,?s) ∧ FreeSpace(?y,?s) ∧ ?x ≠ ?y.	If the top of an object is clear in some achievable state and there is free space on another object, then the first object can be moved from its present location to the new location. However, an object cannot be moved onto itself.
Tower(?topObj,?yMin,?yMax,?xMin,?xMax,?s) ← AchievableState(?s) ∧ Clear(?topObj,?s) ∧ Xpos(?topObj,?xPos,?s) ∧ ?xPos ≥ ?xMin ∧ ?xPos ≤ ?xMax ∧ Ypos(?topObj,?yPos,?s) ∧ ?yPos ≥ ?yMin ∧ ?yPos ≤ ?yMax.	A tower exists if a clear block is located within this region in a valid state.
On(?x,?y,Do(Transfer(?x,?y),?s)) ← AchievableState(Do(Transfer(?x,?y),?s)).	After an object is moved, it is on the destination object.
On(?a,?b,Do(Transfer(?x,?y),?s)) ← On(?a,?b,?s) ∧ ?a ≠ ?x.	If not moved, an object stays where it is (frame axiom).
Clear(?x,Do(Transfer(?x,?y),?s)) ← AchievableState(Do(Transfer(?x,?y),?s)).	After an object is moved, it is clear.
Clear(?z,Do(Transfer(?x,?y),?s)) ← On(?x,?z,?s) ∧ Block(?z) ∧ ?z ≠ ?y.	After an object is moved, the previously supporting object is clear, if it is a block and the moved object is not placed back on top of it.
Clear(?a,Do(Transfer(?x,?y),?s)) ← Clear(?a,?s) ∧ ?a ≠ ?y.	If nothing is placed on it, an object stays clear.

conclusions that can be drawn within that situation. Reasoning within a situation is assumed to be *operational*, while reasoning between situations is not, because hypothesizing intermediary states can proceed without bound. The learning task is to produce new rules whose preconditions can be checked by reasoning only about the initial situation, ignoring the details of all the intermediate situations traversed on the way to the goal situation. Because intra-situational reasoning is considered operational, subproofs that only involve intra-situational rules are ignored during generalization.

Situation calculus is appealing because it expresses planning in a deductive framework. All problem-solving knowledge is explicit and inspectable by the learner. For example, in Green's formulation frame axioms are explicit rules, unlike in a STRIPS planner. Because situation calculus is deductive, the circuit domain's problem solver can be used.

Table 6. Remaining initial rules for the blocks domain.

Rule	Description
Xpos(?x,?xpos,Do(Transfer(?x,?y),?s)) $\leftarrow$ Xpos(?y,?xpos,?s).	After a transfer, the object moved is centered (in the *X*-direction) on the object upon which it is placed.
Ypos(?x,?ypos2,Do(Transfer(?x,?y),?s)) $\leftarrow$ Ypos(?y,?ypos,?s) $\wedge$ Height(?x,?hx) $\wedge$?x $\neq$?y $\wedge$?ypos2 = (?hx + ?ypos).	After a transfer, the *Y*-position of the object moved is determined by adding its height to the *Y*-position of the object upon which it is placed.
Xpos(?a,?xpos,Do(Transfer(?x,?y),?s)) $\leftarrow$ Xpos(?a,?xpos,?s) $\wedge$?a $\neq$?x.	All blocks, other than the one moved, remain in the same *X*-position after a transfer (frame axiom).
Ypos(?a,?xpos,Do(Transfer(?x,?y),?s)) $\leftarrow$ Ypos(?a,?xpos,?s) $\wedge$?a $\neq$?x.	All blocks, other than the one moved, remain in the same *Y*-position after a transfer (frame axiom).
FreeSpace(?x,?s) $\leftarrow$ Clear(?x,?s) $\wedge$ FlatTop(?x).	If an object is clear and has a flat top, then space is available.
Liftable(?x,?s) $\leftarrow$ Clear(?x,?s) $\wedge$ Block(?x).	A block is liftable if it is clear.
FlatTop(?x) $\leftarrow$ Box(?x).	Boxes have flat tops.
FlatTop(?x) $\leftarrow$ Table(?x).	Tables have flat tops.

In the blocks world, an initial situation is created by randomly generating *N* blocks on a table. All possible configurations, from all *N* stacked to all *N* directly on the table, are possible. Once it places the blocks, the problem generator selects a goal height, centered above a second table. The goal height randomly ranges from one to *N* block heights, and the goal is to produce a plan such that a block is at this location. The goal is expressed by the following conjunctive rule:

$$\begin{aligned}&\text{Tower}(?\text{topObj}, ?\text{yMin}, ?\text{yMax}, ?\text{xMin}, ?\text{xMax}, ?\text{s})\\ &\quad\leftarrow\\ &\quad \text{AchievableState}(?\text{s}) \wedge \text{Clear}(?\text{topObj}, ?\text{s}) \wedge\\ &\quad \text{Xpos}(?\text{topObj}, ?\text{xPos}, ?\text{s}) \wedge ?\text{xPos} \geq ?\text{xMin} \wedge ?\text{xPos} \leq ?\text{xMax} \wedge\\ &\quad \text{Ypos}(?\text{topObj}, ?\text{yPos}, ?\text{s}) \wedge ?\text{yPos} \geq ?\text{yMin} \wedge ?\text{yPos} \leq ?\text{yMax}.\end{aligned}$$

This rule says that one has a tower at a given location in some state, provided there is a clear block in this region and the state can be legally reached. As previously mentioned, the problem solver limits plans to at most *N* actions to prevent unbounded back chaining. Using the rules appearing in Tables 5 and 6, the solver can produce a plan for satisfying the goal, starting from the specific situation in Figure 6. This plan stacks the three blocks that are clear in the initial situation (Blocks *B*, *C*, and *E*) to build the desired tower.

By generalizing the resulting explanation, the basic BAGGER2 algorithm produces the rule in Table 7. Notice that, unlike the rule that EGGS would produce, it is not limited to cases in which *three* blocks need be moved. Also notice that the recurrences only test predicates that are either situation-independent or that refer to the initial situation, *s0*. The main rule says that to build a tower, one should first find an achievable situation, then see if the final block moved is at the goal location. This *generate-and-test* approach leads to wasted work, as demonstrated later; it

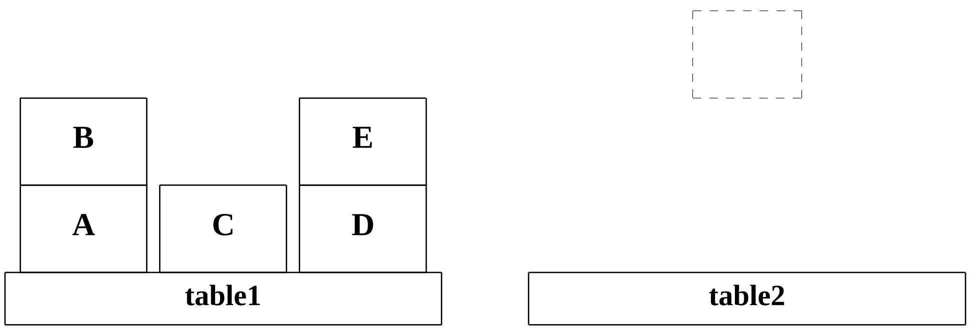


Figure 6. The initial state for a sample tower-building task.

Table 7. Rules learned by the basic BAGGER2 algorithm in the blocks world.

Rule Tower-2: /* Build a tower of arbitrary height. */

Tower(?v1, ?v2, ?v3, ?v4, ?v5, Do(Transfer(?v1, ?v6), ?v7))
$\leftarrow$
call recurrence AchievableState-1 to satisfy AchievableState(Do(Transfer(?v1, ?v6), ?v7)) $\wedge$
call recurrence Xpos-1 to satisfy Xpos(?v1, ?v8, Do(Transfer(?v1, ?v6), ?v7)) $\wedge$
?v8 $\geq$?v4 $\wedge$?v8 $\leq$?v5 $\wedge$
call recurrence Ypos-1 to satisfy Ypos(?v1, ?v9, Do(Transfer(?v1, ?v6), ?v7)) $\wedge$
?v9 $\geq$?v2 $\wedge$?v9 $\leq$?v3.

Recurrence AchievableState-1: /* Reach legal states by moving clear blocks. */

To satisfy: AchievableState(Do(Transfer(?v1, ?v2), ?v3))

One of the following must hold:

?v3 matches s0 $\wedge$ Liftable(?v1, s0) $\wedge$ FreeSpace(?v2, s0) $\wedge$?v1 $\neq$?v2.

or

?v3 matches Do(Transfer(?v2, ?e1), ?e2) $\wedge$ recursively satisfy AchievableState(?v3) $\wedge$
call recurrence Clear-1 to satisfy Clear(?v1, ?v3) $\wedge$ Block(?v1) $\wedge$ FlatTop(?v2) $\wedge$?v1 $\neq$?v2.

Recurrence Xpos-1: /* Determine the *x*-position of the last block moved by finding the first supporting object. */

To satisfy: Xpos(?v1, ?v2, Do(Transfer(?v1, ?v3), ?v4))

One of the following must hold:

?v4 matches s0 $\wedge$ Xpos(?v3, ?v2, s0).

or

?v4 matches Do(Transfer(?v3, ?e1), ?e2) $\wedge$ recursively satisfy Xpos(?v3, ?v2, ?v4).

Recurrence Ypos-1: /* Determine the *y*-position of a block by summing heights. */

To satisfy: Ypos(?v1, ?v2, Do(Transfer(?v1, ?v3), ?v4))

One of the following must hold:

?v4 matches s0 $\wedge$ Ypos(?v3, ?e2, s0) $\wedge$ Height(?v1, ?e1) $\wedge$?v1 $\neq$?v3 $\wedge$?v2 = ?e1 + ?e2.

or

?v4 matches Do(Transfer(?v3, ?e1), ?e2) $\wedge$ recursively satisfy Ypos(?v3, ?e4, ?v4) $\wedge$
Height(?v1, ?e3) $\wedge$?v1 $\neq$?v3 $\wedge$?v2 = ?e3 + ?e4.

Recurrence Clear-1: /* See if a block is clear due to nothing being placed on it. */

To satisfy: Clear(?v1, Do(Transfer(?v2, ?v3), ?v4))

One of the following must hold:

?v4 matches s0 $\wedge$ Clear(?v1, s0) $\wedge$?v1 $\neq$?v3.

or

?v4 matches Do(Transfer(?e1, ?e2), ?e3) $\wedge$ recursively satisfy Clear(?v1, ?v4) $\wedge$?v1 $\neq$?v3.

would be better if the problem solver started the tower at an acceptable horizontal position and checked the tower's height *while* producing a valid final state. The next section describes how one can produce a rule that embodies this strategy.

3.2. Three Extensions to the Basic Algorithm

The extended BAGGER2 algorithm carries out three additional processing steps; each of these can improve the efficiency of the acquired rule, as Section 3.3 demonstrates. Table 8 contains the results of applying the extensions of BAGGER2 to the rules in Table 7. This section describes the three extensions and explains how they produce the result in Table 8.

The first of these applies when multiple calls to recurrences appear in the antecedents of a rule (as in the first rule of Table 7). If these recurrences can be ascertained to occur together in a compatible way, they can be merged, and the generalizer can replace the multiple calls by a single call to the merged recurrence. One class of compatible recurrences contains those that traverse through the same sequence of situations.; each individual recurrence places constraints on an acceptable traversal. Rather than satisfying these constraints successively for each recurrence, the problem solver can satisfy them simultaneously. When this extension merges several recurrences together, it names the new recurrence by concatenating the names of the individual recurrences, and basically produces the union of the individual recurrences. In a sense, this extension further restructures an explanation; it merges independent portions of an explanation structure. A specific solution may involve several independent recursive tasks, while the acquired rule may address these tasks concurrently.

In Table 7's acquired rule for tower building, the calls to *AchievableState-1*, *Xpos-1*, and *Ypos-1* all involve the same final state. Since in situation calculus the final state is defined in terms of a path starting at the initial state, all three recurrences must traverse the same sequence of states and, hence, they can be merged together. Merging requires making copies of the recurrences involved, using the fact that all of the recurrences traverse the same sequence of situations to properly rename variables, and then producing all possible combinations of recursive disjuncts and terminating disjuncts. In the table, the recurrences involved each only have one recursive and one terminating disjunct, so there is only one way to combine them, but in general combination can be explosive. For example, if two recurrences each have three terminating disjuncts, the combined recurrence will have up to nine distinct terminating disjuncts.

The second step determines "unchanging" variables; its value arises in combination with the third step, as will be seen momentarily. Determining the variables that remain unchanged is easy; a variable in a recurrence's consequent is unchanged if, in all possible recursive calls, it appears in the same position. For readability, variables that remain unchanged are renamed to

Table 8. Rules learned by the extended BAGGER2 algorithm in the blocks world.

Rule Tower-3: /* Build a tower of arbitrary height. */

Tower(?v1, ?v2, ?v3, ?v4, ?v5, Do(Transfer(?v1, ?v6), ?v7))
$\leftarrow$
call recurrence AchievableState-Xpos-Ypos-1 to satisfy
AND(AchievableState(Do(Transfer(?v1, ?v6), ?v7)),
Xpos(?v1, ?g1, Do(Transfer(?v1, ?v6), ?v7)),
Ypos(?v1, ?v9, Do(Transfer(?v1, ?v6), ?v7)),
?g1 ≥ ?v4, ?g1 ≤ ?v5) ∧
?v9 ≥ ?v2 ∧ ?v9 ≤ ?v3.

Forward Recurrence AchievableState-Xpos-Ypos-1: /* Legally stack blocks and keep track of their positions. */

To satisfy: AND(AchievableState(Do(Transfer(?v1, ?v3), ?v4)),
Xpos(?v1, ?g1, Do(Transfer(?v1, ?v3), ?v4)),
Ypos(?v1, ?v2, Do(Transfer(?v1, ?v3), ?v4)),
?g1 ≥ ?g2, ?g1 ≤ ?g3)

One of the following must hold:

?v4 matches s0 ∧ Liftable(?v1, s0) ∧ Height(?v1, ?e1) ∧ FreeSpace(?v3, s0) ∧ ?v1≠ ?v3 ∧
Xpos(?v3, ?g1, s0) ∧ ?g1 ≥ ?g2 ∧ ?g1 ≤ ?g3 ∧ Ypos(?v3, ?e2, s0) ∧ ?v2 = ?e1 + ?e2.

or

?v4 matches Do(Transfer(?v3, ?e1), ?e2) ∧
recursively satisfy AND(AchievableState(Do(Transfer(?v3, ?e1), ?e2)),
Xpos(?v1, ?g1, Do(Transfer(?v3, ?e1), ?e2)),
Ypos(?v1, ?e4, Do(Transfer(?v3, ?e1), ?e2)),
?g1 ≥ ?g2, ?g1 ≤ ?g3) ∧
call recurrence Clear-1 to satisfy Clear(?v1, ?v4) ∧
Block(?v1) ∧ FlatTop(?v3) ∧ ?v1 ≠ ?v3 ∧ Height(?v1, ?e3) ∧ ?v2 = ?e3 + ?e4.

Forward Recurrence Clear-1: /* See if a block is clear due to nothing being placed on it. */

To satisfy: Clear(?g1, Do(Transfer(?v2, ?v3), ?v4))

One of the following must hold:

?v4 matches s0 ∧ Clear(?g1, s0) ∧ ?g1 ≠ ?v3.

or

?v4 matches Do(Transfer(?e1, ?e2), ?e3) ∧ recursively satisfy Clear(?g1, ?v4) ∧ ?v1 ≠ ?v3.

start with *g*, indicating they are “global” variables.

The final step determines if a linear recurrence should be satisfied forward or backward. If a recurrence only involves a linear chain of recursive calls, it may be more efficient to satisfy the recurrence by starting at the initial state and working forward until reaching the desired final state. BAGGER2 heuristically chooses the direction to satisfy recurrences. It selects working forward from the initial state only when any unchanging variables are present, since these variables specify constraints on the initial action to be performed.

When the problem solver is to satisfy a recurrence from the initial state forward, some terms in the conjunct that calls the recurrence may be pushed down into the recurrence's terminating disjuncts, as these extra terms further constrain the initial action. The terms pushed down are those that only involve variables that are unchanging or independent of the recurrence. This is done for the inequalities involving *?v8* in Table 7's tower-building rule. It is preferable to satisfy these constraints early, rather than after the recurrence produces a candidate tower. The *x*-location of the first block moved determines the *x*-location of the tower, so the inequalities eliminate time wasted investigating improperly placed towers. Here an extension also restructures an explanation; it moves preconditions from the end of a linear chain of rule applications down to the first step in the sequence.

The result in Table 8 is essentially an iterative plan. It is a notational variant on the result produced by the original BAGGER (Shavlik, in press). Blocks are stacked until a tower of the desired height is produced; at each step in the iteration the problem solver must choose a block to move. The plan does not require the use of any other inter-situational rule in the rule base. There may be many ways to build towers or to verify that a block is clear, but the solver expends no resources trying out these portions of the search space. Attention during problem solving is tightly focused; any testing done outside of the acquired plan only involves checking properties of the initial state.

For a further illustration of learning from multiple examples, consider again the rule in Table 8. This rule only supports stacking blocks that are clear in the initial state, which is reasonable given that it is produced from a solution which stacks three initially clear blocks upon one another. One can also move some initially obstructed block *x* if the block on top of it has been moved and no other blocks have been placed on block *x*. If after learning the rule in Table 8, BAGGER2 observes a solution where Figure 6's blocks are moved in reverse alphabetical order, the disjunction below will replace the call to *Clear-1*.

call recurrence Clear-1 to satisfy Clear(?v1, ?v4).

or

call recurrence On-1 to satisfy On(?v3, ?v1, ?e2) $\wedge$?v1 $\neq$?e1.

It specifies constraints on the next block to be stacked. This block (*?v1*) must either be originally clear or it must have originally supported the block moved in the previous step (*?v3*), provided *?v3* was not placed on *?v1*. A call to the appropriate recurrence insures that the relevant relationship in the initial state still holds, given the plan constructed so far.[4]

There are two technical points concerning the interaction of learning from multiple examples and the three extensions to the basic BAGGER2 method. One, the extensions can alter the consequent of the recurrence, which complicates determining that a new recurrence's disjuncts support the same conclusion as that of an old one. Two, the addition of a disjunct can invalidate the applicability of an extension. For these reasons, modified recurrences maintain records of the basic recurrences from which they were produced. The generalizer only adds new disjuncts to pre-existing basic recurrences; it then reconstructs all of the modified recurrences that depend on the basic recurrences.

3.3. Empirical Studies of the Extended Algorithm

This section empirically ascertains the value of the three extensions to the basic algorithm, and compares BAGGER2, EGGS, and NO-LEARN in a second domain.

The first study investigates the hypothesis that the extensions improve BAGGER2's performance. Using the blocks world domain, it compares the basic and extended algorithms, as well as the three partial extensions that result from dropping one technique. These partial extensions provide information of the individual contributions of the three refinements. Each configuration generalizes the solution where the blocks in Figure 6 are stacked in reverse alphabetical order on the second table. The problem generator then produces 100 random configurations of five blocks on one table, and the goal is to build a five-block tower on a second table. Finally, each configuration solves the 100 test problems, and its mean solution time is recorded. Table 9 contains the results, which demonstrate the value of the three extensions.

Table 9. Evaluation of the extensions to the basic BAGGER2 algorithm.

System	*Mean Solution Time*
Basic BAGGER2	53.7 sec.
Extended BAGGER2	3.8 sec.
without merging related recurrences	7.9 sec.
without marking unchanging variables	5.9 sec.
without selecting problem-solving direction	14.8 sec.

[4] The recurrence *On-1* is analogous to *Clear-1* and, hence, is not shown.

In this experiment the three extensions provide a speed up of more than ten over the basic approach. The bottom rows in Table 9 indicate the individual contributions of the three extensions. Deciding that the problem solver should satisfy recurrences from the initial state forward provides the largest contribution. Tower building is a task for which planning naturally proceeds from the first action forward; select the starting position, then choose movable blocks and stack them. When it starts by choosing the last block to stack, the problem solver may perform substantial work before realizing that, given the bound on the number of actions, no plan exists where this block is moved last. Conversely, some plans more naturally proceed from the last action backward. The task of clearing a block is one example; move the block on top after first clearing it. The second largest contribution to efficiency is produced by merging recurrences that traverse through the same situations; while producing legal states, it is worthwhile to also record the positions of the blocks moved. Finally, marking variables that remain constant throughout a recurrence produces additional speedup. In the stacking problem, this allows BAGGER2 to move the requirements on the starting *x*-position into the recurrence. This insures that the problem solver only considers properly located bases for the tower. While the relative contributions of the three extensions heavily depend on the task of tower building, this study indicates the value of reorganizing a collection of basic recurrences.

The remaining two studies compare BAGGER2, EGGS, and NO-LEARN. The methodology for these experiments is basically the same as that used in the circuit implementation experiment. However, when the learners cannot solve a training problem using their acquired rules, they are provided a solution to the problem (consisting of a sequence of transfers), which they then explain and generalize. Having the learners solve large problems using only the initial domain theory rules is intractable, as the performance of NO-LEARN shows.

The second study re-addresses the hypothesis that BAGGER2 requires less training than EGGS does, this time using the blocks world. As in the circuit domain, the amount of training and the learning algorithm are the independent variables, while the study measures the percentage of novel problems solved using a learned rule. It follows the same methodology as the first circuit implementation study; again there are 250 randomly-generated training problems in each of the ten experimental runs. During each run, the two learners periodically attempt to solve the same ten test problems, during which learning is turned off; hence each point is the mean of 100 measurements. A given run uses the same ten test problems throughout, but each run has its own test set. For all problems five blocks are randomly dropped over a table (see Figure 6), and the goal height is randomly chosen to be from one to five blocks.

Figure 8 presents the results of this study. Similar to the circuit implementation experiment, BAGGER2 requires less training than does EGGS to acquire completely the ability to build towers. EGGS learns templates that describe situations in which it can build a tower of a

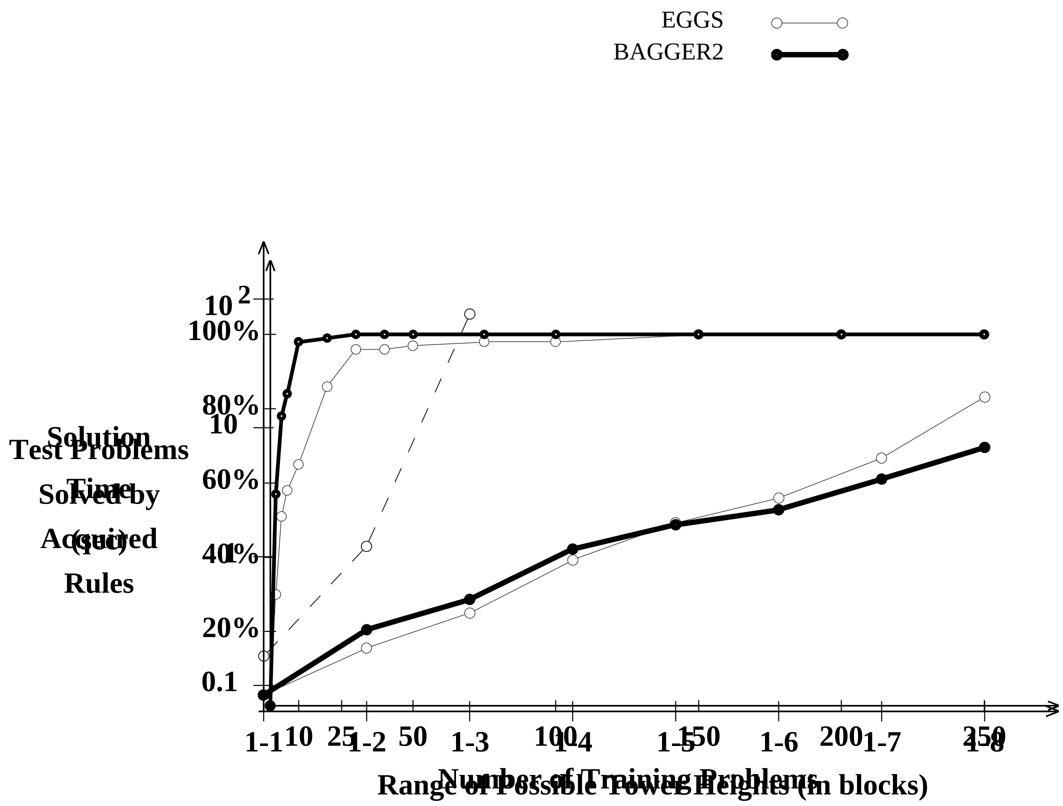
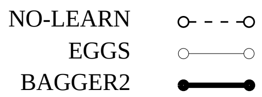


Figure 7. Percentage of new tower problems solved as a function of amount of training.

Figure 8. Mean solution time for tower building as a function of problem complexity.



fixed height; plans for stacking three clear blocks, inverting a five-block stack at one position on to another, using the top two blocks of two existing towers to build a four-block stack, etc. As in the circuit domain, it must receive many training examples before it encounters enough inherently different configurations to capture the concept. (On average, EGGS learns 14.6 rules per run.) BAGGER2 instead learns a strategy for building towers; stack clear blocks until the goal is met. It needs to learn it can move a block initially clear or one made clear by previously moving the block it supports. Once it observes solutions involving these subtasks, it can build towers of any height. (Usually BAGGER2 learns one tower-building rule per run; occasionally the first training example does not support generalization to N and several rules are learned.)

The third study in this section re-addresses the hypotheses that explanation-based learning speeds up a problem solver and that BAGGER2 outperforms EGGS. As before, the independent variables are the identity of the system used and the size of the space of possible problems, while the dependent variable is the average problem-solving time following learning. Incrementing the upper limit on possible tower heights increases the number of possible problems; at each point the systems solve problems requiring stacking from one to N blocks, for some fixed N, in a scene containing N blocks. In this experiment, 500 training examples are presented at each point to insure that both learning systems sufficiently learn how to build towers. The results are averaged over five experimental runs, and in each run performance is measured on 20 different random problems, producing a total of 100 measurements per point. For a given problem size, all systems receive the same training examples and solve the same test problems. When N is above five, after training EGGS occasionally cannot solve a problem using its acquired rules; however, only the time spent on problems solved is recorded. Figure 8 presents the performance of the two learning systems, along with that of the system that does not learn, on the tower-building task as a function of the range of possible tower heights. Notice that times are plotted on a logarithmic scale, where exponentially increasing functions produce straight lines. As with circuit implementation, as the problem complexity increases, BAGGER2 begins to outperform EGGS. For both learning systems, one can clearly see the advantage over not learning at all.

In this study the relative performance of EGGS and BAGGER2 is qualitatively the same as in the circuit implementation study; for small ranges of possible problems, collecting and matching fixed-sized templates is effective, but it is better to learn a general solution strategy when there are many possible configurations. Unlike the circuit domain, Figure 8 provides no indication that explanation-based learning is detrimental in this domain. The negative effects of learning depend on the relationship between the basic problem-solving complexity of a set of domain rules and the number of qualitatively different composite rules required in that domain. Explanation-based learning may prove particularly beneficial in domains where the number of compositive rules a problem solver needs is small compared to the number of possible combinations of the basic domain rules. Analytical studies of explanation-based learning may provide important insight on this issue.

4. Related Work

In addition to BAGGER2 and its predecessor BAGGER (Shavlik, in press; Shavlik & DeJong, 1987), which only learns iterative concepts, several other explanation-based approaches to generalizing number have been proposed. This section briefly presents these other approaches and compares them to BAGGER2. Also, some related projects involving similarity-based learning and automatic programming are discussed.

For instance, Cohen's (1988) ADEPT system generalizes number by constructing a finite-state control mechanism that deterministically directs the construction of proofs similar to the one used to justify the specific example. His approach can acquire recursive and disjunctive concepts, as well as learn from multiple examples. However, in order to eliminate backtracking when applying the learned rule, his system assumes that operational terms can be matched by no more than one fact in the database. By disallowing backtracking, ADEPT improves efficiency at the cost of some expressiveness. This means that, unlike BAGGER2, it cannot learn how to build towers in a situation where Blocks A, C, and E are initially clear and then apply its learned strategy in a situation where Blocks B, D, and F are initially clear. Another difference from BAGGER2 is that, to learn from multiple examples, ADEPT requires the previous examples be present in their entirety. Cohen's method also differs from other explanation-based algorithms in that it does not eliminate "internal nodes" of the explanation during generalization. In other approaches, only the leaves of the operationalized explanation appear in the acquired rule's antecedents. In Cohen's approach, every inference rule used in the original explanation is explicitly incorporated into the final result. Each rule may again be applied when satisfying the acquired rule. Finally, unlike BAGGER2, Cohen's system does not separately extract subconcepts from portions of an explanation.

On another front, Prieditis (1986) developed a system which learns macro-operators that represent linear sequences of repeated STRIPS-like operators. His approach analyzes the constraints imposed by the unbounded connection of the precondition, add, and delete lists of the operators deemed to be of interest. This produces an iterative macro-operator that accommodates an indefinite number of repeated operator inter-connections. Although his approach produces iterative rules, it can learn neither recursive nor disjunctive rules. Also, unlike Prieditis' approach, BAGGER2 allows arbitrary inference rules to intervene between applications of the predicate whose number of appearances is being generalized.

A third system, Cheng and Carbonell's FERMI (1986), recognizes cyclic patterns using empirical methods, and generalizes the detected repeated pattern using explanation-based learning techniques. A major strength of the system is the incorporation of conditionals within the learned macro-operator. However, unlike the techniques implemented in BAGGER2, the rules acquired by FERMI are not fully based on an explanation-based analysis of an example, and so are not guaranteed to always work. For example, the system learns a strategy for solving a set of linear algebraic equations, but none of the preconditions of the strategy check that the equations are linearly independent. Thus, the learned strategy will appear applicable to the problem of determining x and y from the equations $3x + y = 5$ and $6x + 2y = 10$, and after a significant amount of work, it will terminate unsuccessfully. Shell and Carbonell (1989) present improvements that increase the efficiency of the macro-operators FERMI learns.

Shavlik and DeJong (1985, in press) developed the first explanation-based learning system that generalized number. Their PHYSICS 101 system acquires the knowledge that momentum is conserved for any *N* objects from an example involving the collision of a fixed number of balls. It differs from the above approaches, including BAGGER2, in that the need for generalizing number is motivated by an analytic justification of an example's solution and general domain knowledge. In the momentum problem, information about number, localized in a single physics formula, leads to a global restructuring of a specific solution's explanation. However, PHYSICS 101 is designed to reason about the use of mathematical formulae, and its generalization algorithm takes great advantage of the properties of algebraic cancellation (e.g., $x - x = 0$). To constitute a broad solution of the generalization to *N* problem, an approach must also handle non-mathematical domains.

A related task is generalizing the *organization* of the nodes in the explanation, rather than generalizing their *number*. Mooney (1988) presents an approach along these lines. His method, which is limited to domains expressed in the STRIPS formalism, determines the minimal set of constraints on the order of a plan's actions. Without this knowledge, the actions in the generalized plan must occur in the same order as in the training example. Strictly speaking, his approach does not alter the explanation structure. Rather, it produces the most general partial ordering of the plan's actions that maintains all connections between preconditions and effects in the original example. Though his technique can not handle it, Mooney discusses an example where the generalization of operator order requires alteration of the explanation.

The problem of generalizing to *N* has also been addressed within the paradigm of empirical or similarity-based learning (e.g., Andreae, 1984; Dietterich & Michalski, 1984, Sammut & Banerji, 1986). Like BAGGER2, the MARVIN system of Sammut and Banerji (1986) uses Horn clauses to represent concepts. The recursive concept *column*, which is a stack of objects, is one of the objects it learns to recognize. It learns by inductively generalizing training instances; these generalizations are corroborated by generating new examples and asking its teacher if they are a member of the concept being taught. A major difference between MARVIN's and BAGGER2's approaches is that due to the former's inductive nature, it can incorrectly learn a concept. BAGGER2's explanation-based concepts are immediately deductively supported by the domain theory; hence confirmation and revision are unnecessary. Also, MARVIN strives to learn abstract recognition rules, while BAGGER2's goal is to acquire efficiently applicable (operational) ones — it already possesses a general description of a tower.

As previously discussed, BAGGER2 recurrences are essentially recursive programs. Unlike the simple template-matching rules EGGS learns, BAGGER's rules can produce solutions of various sizes. Some research in automatic programming shares many characteristics with this approach, namely that involving program synthesis from examples (e.g., Bauer, 1979; Biermann, 1978; Kodratoff, 1979; Summers, 1977). In these approaches, sequences of input/output pairs for recursive functions provide information on the control structure of the algorithm being specified. A major problem with input/output pairs is that for complex operations the amount of search needed to find the proper algorithm is prohibitive. Automatic programming systems that use examples must search for a consistent hypothesis because, unlike an explanation-based system, they do not have the information that specifies the dependencies between successive recursive calls. In this sense, they are similar to similarity-based learning algorithms; both must make unjustified generalizations, unlike those an explanation-based system makes.

5. Some Open Research Issues

Although the BAGGER2 system has taken important steps towards the solution to the "generalization to N" problem, the research is still incomplete. From the vantage point of the current results, several avenues of future research are apparent.

A major weakness of the rules learned by BAGGER2 is that a problem solver can expend much useless effort when they fail. For example, assume the task at hand is to find enough heavy rocks in a storehouse to serve as ballast for a ship. An acquired rule may first add the weights in some order, find that the total weight of all the rocks in the room is insufficient, and then try another ordering for adding the weights. The system should be capable of realizing that the actions in an acquired rule produce the same result regardless of their order. This could be accomplished by reasoning about the semantics of the system's predicate calculus functions and predicates. Properties such as symmetry, transitivity, and reflexivity may help determine constraints on order independence. Programmers use PROLOG's *cut* operator (Clocksin & Mellish, 1984) to indicate where backtracking will be a waste of time, and it may prove fruitful to have a learning system decide where to place cuts in the rules it acquires.

A related area of future research involves determining the most efficient ordering of conjunctive goals (Smith & Genesereth, 1985) in recursive rules. Consider an acquired iterative rule that builds towers of a desired height, subject to the constraint that no block can be placed upon a narrower block. The goal of building such towers is conjunctive: the correct height must be achieved and the width of the stacked blocks must be monotonically non-increasing. The optimal ordering can be found by selecting the blocks subject only to the height requirement and then sorting them by size to determine their position in the tower. This strategy works because it guarantees a non-increasing ordering of widths on any set of blocks, so that no additional block-selection constraints are imposed by this conjunct. The system should ultimately detect and exploit this kind of decomposability to improve the efficiency of the new rules.

BAGGER2 uses a relatively simple technique for parsing explanations (c.f., Figure 2). The class of recursive concepts this technique recognizes needs to be characterized, and BAGGER2 should be extended to cover a wider range of recursive rule applications. Techniques for detecting recursive patterns developed in automatic programming research may be applicable to this task (Smith, 1984). However, such approaches can introduce backtracking search into the generalization process, thereby leading to problems of intractability. Also, if they allow multiple parses of an explanation, techniques for choosing the best parse may be required.

Often an explanation will not be a tree, as has been assumed in this article, and some portions of the explanations will be shared. These shared portions can arise when one action satisfies preconditions for several subsequent actions. The BAGGER2 algorithm can handle shared subexplanations be replicating them. Although this will lead to a more general concept, the efficiency of sharing is lost. The problem with shared nodes in a system that generalizes explanation structures involves synchronization. In one recurrence the shared node can be encountered on the ith cycle, while in another it may be the jth cycle. This complicates the identification of variables that should be equated. One solution involves having recurrences check for shared nodes during problem solving. If a node is marked, then there is no need to continue the recurrence. Instead, the current term can be unified with the term that did the marking.

Another extension would simplify loops during learning. Often a repeated process has a closed form solution. For example, summing the first N integers produces $\frac{N(N+1)}{2}$, and there is no need to compute the intermediate partial summations. A *recurrence relation* is a recursive method for computing a sequence of numbers. Many recurrences can be solved to produce efficient ways to determine the nth result in a sequence. It is this property that motivates the requirement that BAGGER2's preconditions be expressed solely in terms of the initial state. The system would be more efficient if it could produce, whenever possible, number-generalized rules in closed form. For instance, if BAGGER2 observes the summation of four numbers it will not produce the efficient result mentioned above; instead it will produce a rule that performs the intermediate summations. One possible extension is to create a library of templates for soluble recurrences, then match them against explanations (Shell & Carbonell, 1989). However, a more direct approach, such as Weld's (1986) *aggregation* technique, may be more fruitful; aggregation creates a description of a continuous process from a series of discrete events.

A major weakness of current EBL algorithms that generalize explanation structures is that they do not generalize the structure of the *goal*. The examples studied do not require this type of generalization. For example, the goal of having a block at a given height should not be generalized to having N blocks at M heights. Instead, the number of blocks stacked should be generalized so that a given block can be placed at any height. However, if the specific plan involved finding the average of five numbers in an array, the general plan acquired should support the determination of the average of any size array. One approach to this issue is to develop methods for determining general versions of specific goals, then construct the explanation for the general goal, using the specific problem's explanation for guidance.

Another area of future research is to investigate how BAGGER2 and related systems might acquire accessory inter-situational rules, such as frame axioms, to complement their acquired rules. Currently, each of the learned inference rules specifies how to achieve a goal that involves some arbitrary aggregation of objects by applying some number of operators. These rules are useful in directly achieving goals that match the consequent, but they do not effectively improve BAGGER2's backward-chaining problem-solving ability. This is because the current system does not construct new frame axioms for the rules it learns.[5] There are several methods of acquiring such accessory rules. One technique would construct them directly by combining the accessory rules of operators that make up the acquired rule. However, the number of accessory rules for initial operators may be so large as to make this intractable. Another, potentially more attractive, approach is to treat the domain theory as intractable. Since new accessory rules can be derived from existing knowledge of initial operators, the approach taken by Chien (1989) might be used to acquire the unstated but derivable accessory rules when they are needed. Chien's system makes simplifying assumptions during plan understanding in order to keep the task tractable. Failure in later applying a learned plan leads to in-depth investigation of the assumptions and then refinement of the plan.

There is also a need for research on the generalization to *N* problem in the context of imperfect domain models (Mitchell, Keller, & Kedar-Cabelli, 1986; Rajamoney & DeJong, 1987). In any real-world domain, a computer system's model can only approximate reality. Furthermore, the complexity of problem solving prohibits any semblance of completeness. Thus far, BAGGER2 has relied on a correct domain model. Also, it has not addressed issues of intractability, other than using an outside agent to provide sample solutions when the internal construction of solutions is intractable. One relevant form of theory imperfection occurs when the effects of an operator are not precisely specified. In this case, small errors may accumulate when repeatedly applying the operator. An approach to this problem is to monitor the actions in recursive plans, seeing how well their effects match the system's expectations. When the system detects significant divergence, it can limit the recursive plan to some maximum length or correct the plan to accommodate the uncertainty in the operators.

---

[5] This problem is not specific to systems that generalize explanation structures. Standard EBL algorithms must also face it when dealing with situation calculus.

A final area of research concerns termination. One weakness of systems that generalize explanation structures is that they may fall into infinite loops. Although the *halting problem* is undecidable in general, one can prove termination in restricted circumstances (Manna, 1974). Systems that generalize number need to incorporate techniques for proving termination. BAGGER2 contains a partial solution to this problem. If a recurrence involves "unchanging" variables, before calling the recurrence the problem solver checks those terms which involve these variables and which also appear in all of the recurrence's terminating disjuncts. If it cannot satisfy these terms, the problem solver does not call the recurrence. These checks reduce the chance of unbounded recursive calls, but they do not guarantee termination. A less appealing, but safe, solution is to place resource bounds on the algorithms that apply number-generalized rules, potentially excluding successful applications.

6. Conclusion

Explanation-based learning systems must generalize number if they are to fully extract general concepts inherent in the solutions to specific examples. This article has presented and proven correct a general approach for generalizing to *N*. The BAGGER2 algorithm learns recursive and iterative concepts, integrates results from multiple examples, and extracts useful subconcepts during generalization. On problems for which learning a recursive rule is not appropriate, the system produces the same result as Mooney's EGGS algorithm, a standard EBL technique. Applying the learned recursive rules only requires a minor extension to a PROLOG-like system, namely, the ability to explicitly call a specific rule. This lets the problem solver focus its attention on a small subset of a large rule base.

The empirical studies reported in Sections 2 and 3 demonstrate that generalizing the structure of explanations helps avoid the negative effects of learning (Fikes *et al.*, 1972; Minton, 1988). These experiments tentatively indicate that BAGGER2 produces substantial performance improvements over standard explanation-based methods and problem solvers that do not learn. In two sample domains, it learns rules that are both more general and more efficient than those learned by a standard EBL system, and its advantage grows as the complexity of the task increases. Its strength arises from its ability to extract a general algorithm from the solution to a specific problem. In one sample domain it learns a general version of a DeMorgan's Law upon observing the repeated application of a two-gate version. In a second domain it learns how to build towers by observing three blocks being stacked. The standard explanation-based system learns rules that state how to apply DeMorgan's Law exactly seven times or how to stack three blocks. Because it does not generalize the number of repetitions, such a system must learn many separate rules. Searching through this large collection of rules greatly reduces the gains learning can produce; in the circuit domain experiments, the standard EBL system performs worse than not learning at all. BAGGER2's rules encapsulate focused traversals through the basic domain rules and in the experiments never lead to performance worse than achieved by only using the basic rules.

Generalizing to *N* is an important property that is currently lacking in most explanation-based systems. This research contributes to the theory and practice of explanation-based learning by developing and testing methods for extending the structure of explanations during generalization. As such, it brings the field of machine learning closer to its goal of being able to acquire all of the knowledge inherent in the solution to a specific problem.

ACKNOWLEDGEMENTS

This work was supported by a grant from the University of Wisconsin Graduate School. Discussions with Jerry DeJong, Ray Mooney, Pat Langley, Yves Kodratoff, Eric Gutstein, Rich Maclin, and Geoff Towell substantially improved the research and its reporting. Their comments and suggestions, plus those of three anonymous reviewers, are greatly appreciated.

References

Ahn, W., Mooney, R. J., Brewer, W. F., & DeJong, G. F. (1987). Schema acquisition from one example: Psychological evidence for explanation-based learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (pp. 50-57). Seattle, WA: Lawrence Erlbaum.

Anderson, J. R. (1986). Knowledge compilation: The general learning mechanism. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Andreae, P. M. (1984). *Justified generalization: Acquiring procedures from example.* Doctoral dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Bauer, M. (1979). Programming by examples. *Artificial Intelligence, 12,* 1-21.

Biermann, A. W. (1978). The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics, 8,* 585-600.

Cheng, P., & Carbonell, J. G. (1986). The FERMI system: Inducing iterative macro-operators from experience. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 490-495). Philadelphia, PA: Morgan Kaufmann.

Chien, S. A. (1989). Using and refining simplifications: Explanation-based learning of plans in intractable domains. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 590-595). Detroit, MI: Morgan Kaufmann.

Clocksin, W. F., & Mellish, C. S. (1984). *Programming in PROLOG.* Berlin: Springer Verlag.

Cohen, W. W. (1987). A technique for generalizing number in explanation-based learning. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 256-269). Ann Arbor, MI: Morgan Kaufmann.

Cohen, W. W. (1989). *Solution path caching mechanisms which provably improve performance* (Technical Report DCS-TR-254). New Brunswick, NJ: Rutgers University, Department of Computer Science.

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

Dietterich, T. G., & Michalski, R. S. (1984). Discovering patterns in sequences of objects. *Artificial Intelligence, 25,* 257-294.

Ellman, T. (1985). Generalizing logic circuit designs by analyzing proofs of correctness. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 643-646). Los Angeles, CA: Morgan Kaufmann.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence, 3,* 251-288.

Green, C. C. (1969). Application of theorem proving to problem solving. *Proceedings of the First International Joint Conference on Artificial Intelligence* (pp. 219-239). Washington, D.C.: Morgan Kaufmann.

Hirsh, H. (1987). Explanation-based generalization in a logic-programming environment. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 221-227). Milan, Italy: Morgan Kaufmann.

Jacobson, N. (1951). *Lectures in abstract algebra* (Vol. 1). Princeton, NJ: Von Nostrand.

Kedar-Cabelli, S. T. (1986). Purpose-directed analogy: A summary of current research. In T. M. Mitchell, J. G. Carbonell, & R. S. Michalski, (Eds.), *Machine learning: A guide to current research.* Hingham, MA: Kluwer.

Kedar-Cabelli, S. T., & McCarty, L. T. (1987). Explanation-based generalization as resolution theorem proving. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 383-389). Irvine, CA: Morgan Kaufmann.

Keller, R. M. (1987). Defining operationality for explanation-based learning. *Proceedings of the National Conference on Artificial Intelligence* (pp. 482-487). Seattle, WA: Morgan Kaufmann.

Kodratoff, Y. (1979). A class of functions synthesized from a finite number of examples and a LISP program scheme. *International Journal of Computer and Information Sciences, 8,* 489-521.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence, 27,* 97-109.

Laird, J. E. , Rosenbloom, P. S., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning, 1,* 11-46.

Manna, Z. (1974). *Mathematical theory of computation.* New York: McGraw-Hill.

McCarthy, J. (1963). *Situations, actions, and causal laws* (Memorandum). Stanford, CA: Stanford University, Department of Computer Science. (Reprinted in M. Minsky, (Ed.), *Semantic information processing.*, 1968, Cambridge, MA: MIT Press.)

Minton, S. N. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564-569). St. Paul, MN: Morgan Kaufmann.

Minton, S. N. (1989). *Learning effective search control knowledge: An explanation-based approach.* Hingham, MA: Kluwer.

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

Mitchell, T. M., Mahadevan, S., & Steinberg, L. I. (1985). LEAP: A learning apprentice for VLSI design. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 573-580). Los Angeles, CA: Morgan Kaufmann.

Mooney, R. J. (1988). Generalizing the order of operators in macro-operators. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 270-283). Ann Arbor, MI: Morgan Kaufmann.

Mooney, R. J. (1989). The effect of rule use on the utility of explanation-based learning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 725-730). Detroit, MI: Morgan Kaufmann.

Mooney, R. J. (in press). *A general explanation-based learning mechanism and its application to narrative understanding.* London: Pitman.

Mooney, R. J., & Bennett, S. W. (1986). A domain independent explanation-based generalizer. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 551-555). Philadelphia, PA: Morgan Kaufmann.

Prieditis, A. E. (1986). Discovery of algorithms from weak methods. *Proceedings of the International Meeting on Advances in Learning* (pp. 37-52). Les Arcs, Switzerland.

Rajamoney, S., & DeJong, G. F. (1987). The classification, detection, and handling of imperfect theory problems. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 205-207). Milan, Italy: Morgan Kaufmann.

Riddle, P. J. (1989). *Automating shifts of problem representation.* Doctoral dissertation, Department of Computer Science, Rutgers University, New Brunswick, NJ.

Sammut, C. B., & Banerji, R. B. (1986). Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Segre, A. M. (1987). On the operationality/generality trade-off in explanation-based learning. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 242-248). Milan, Italy: Morgan Kaufmann.

Shavlik, J. W. (in press). *Extending explanation-based learning by generalizing the structure of explanations.* London: Pitman.

Shavlik, J.W., & DeJong, G. F. (1985). Building a computer model of learning classical mechanics. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society* (pp. 351-355). Irvine, CA: Morgan Kaufmann.

Shavlik, J.W., & DeJong, G. F. (1987). BAGGER: An EBL system that extends and generalizes explanations. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 516-520). Seattle, WA: Morgan Kaufmann.

Shavlik, J.W., & DeJong, G. F. (in press). Learning in mathematically-based domains: Understanding and generalizing obstacle cancellations. *Artificial Intelligence*.

Shavlik, J.W., DeJong, G. F., & Ross, B. H. (1987). Acquiring special case schemata in explanation-based learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (pp. 351-355). Seattle, WA: Morgan Kaufmann.

Shavlik, J. W., & Maclin, R. (1988). An approach to acquiring algorithms by observing expert behavior. *Proceedings of the AAAI-88 Workshop on Automating Software Design* (pp. 172-181). St. Paul, MN.

Shell, P., & Carbonell, J. G. (1989). Towards a general framework for composing disjunctive and iterative macro-operators. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 596-602). Detroit, MI: Morgan Kaufmann.

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

Smith, D. R. (1984). The synthesis of LISP programs from examples: A survey. In A. Biermann, G. Guiho, and Y. Kodratoff, (Eds.), *Automatic program construction techniques.* New York: MacMillan.

Summers, P. D. (1977). A methodology for LISP program construction from examples. *Journal of the Association for Computing Machinery, 24,* 161-175.

Weld, D. S. (1986). The use of aggregation in casual simulation. *Artificial Intelligence, 30,* 1-34.