OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

A Study of Explanation-Based Methods for Inductive Learning

Nicholas S. Flann
Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

88-30-11

A Study of Explanation-Based Methods for Inductive Learning

Nicholas S. Flann
Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331
flann@cs.orst.edu

Running head: *INDUCTION OVER EXPLANATIONS*
(Submitted to the Machine Learning Journal)

December 14, 1988

Abstract

This paper formalizes a new learning from examples problem: identifying a correct concept definition from positive examples such that the concept is some specialization of a target concept defined by a domain theory. This paper describes an empirical study that evaluates three methods for solving this problem: explanation based generalization (EBG), multiple example explanation based generalization (mEBG), and a new method, induction over explanations (IOE). The study demonstrates that the two existing methods (EBG and mEBG) exhibit two shortcomings: (a) the methods rarely identify the correct definition, and (b) the methods are brittle—their success depends greatly on the choice of encoding of the domain theory rules. The study demonstrates that the new method, IOE, does not exhibit these shortcomings. The IOE method applies the domain theory to construct explanations from multiple training examples as in mEBG, but forms the concept definition by employing a similarity-based generalization policy over the explanations. The method has the advantage that an explicit domain theory can be exploited to aid the learning process, the dependence on the initial encoding of the domain theory is significantly reduced, and the correct concepts can be learned from few examples. The study evaluates the methods in an implemented system, called *Wyl2*, learning a variety of concepts in chess including "skewer" and "knight-fork."

Key words: Learning from examples, induction over explanations, explanation based learning, inductive learning, knowledge compilation, evaluation of learning methods.

1 Introduction

Explanation-based generalization (EBG) is usually presented as a method for improving the performance of a problem solving system without introducing new knowledge into the system (i.e., without performing *knowledge-level learning*; Dietterich, 1986). The problem solver begins with an inefficient, but correct, domain theory (DT) that defines a target concept (TC). The learning process consists of repeatedly accepting a training example (E), applying the domain theory to prove that E is an instance of the target concept, and then extracting the weakest preconditions of that proof to form an efficient "chunk" that provides an easy-to-evaluate sufficient condition for TC. This chunk can be used during future problem solving to quickly determine that E and many examples similar to E are instances of TC (see Mitchell, Keller, and Kedar-Cabelli 1986; DeJong and Mooney 1986).

According to this perspective, EBG is related to other methods of knowledge compilation such as partial evaluation (Preditis, 1988; Van Harmelen & Bundy, 1988) and test incorporation (Bennett and Dietterich, 1986) because it is simply converting the target concept into "operational" (i.e., efficient) form.

However, there is way in which the *same mechanism*—computing weakest preconditions—can be applied to acquire new knowledge. The method works as follows. Suppose that the training example E, in addition to being an instance of TC, is also an instance of another, more specific concept C. As before, the domain theory is applied to demonstrate that E is an instance of TC, and the weakest preconditions (call them WP) are extracted from the proof. But instead of just viewing WP as a sufficient condition for TC, we can also view WP as *necessary and sufficient conditions* for C. In short,

$$WP(E) \equiv C(E).$$

By making this assertion, the learning program is making an inductive leap and thus performing knowledge-level learning.

When viewed in this way, the purpose of explanation-based learning is not to translate TC into more efficient form, but instead to *identify the correct definition* of C. The target concept and the domain theory are acting in the role of a *semantic bias* by providing a good vocabulary in which to define C (i.e., the language in which the domain theory is expressed) and by dictating how the training example should be generalized (i.e., by computing the weakest preconditions of the proof).

A review of the literature reveals that many applications of EBG are best viewed from this second perspective. Consider, for example, the OCCAM system (Pazzani, 1988). In OCCAM, the domain theory defines several target concepts including the concept of "coercion" (i.e., achieving a goal by making a threat). In one example from (Pazzani, 1988), OCCAM is given a scenario where one country threatens to stop selling an essential product to another country. This scenario is simultaneously an example of "coercion" and also an example of the more specific concept "economic sanction." OCCAM applies its domain theory for "coercion" to obtain the weakest preconditions for the scenario to succeed and then assumes that these weakest preconditions define the "economic sanction" plan. This assumption is clearly an inductive leap, since OCCAM does not know that the weakest preconditions are a correct definition of "economic sanction."

Many other examples of this inductive application of weakest preconditions can be found in systems that learn control knowledge, such as LEX2 (Mitchell, Utgoff and Banerji, 1983) and Prodigy (Minton, 1988). The problem solver in these systems constructs a set of preference rules to decide which operator to apply to solve each problem or subproblem. The domain theory in these systems defines the target concept "operator succeeds" by stating that a successful operator is an operator that solves the problem or is the first step in a sequence of operators that solves the problem[1]. Training examples are constructed by applying a heuristic search to find a successful sequence of operators. Then, the first operator in this sequence, Op_1 is an instance of the concept "operator succeeds". However, the first operator in the sequence is also *assumed* to be an instance of the concept "best operator to apply." Using the domain theory for "operator succeeds", EBG constructs the weakest preconditions WP under which Op_1 will solve the given problem. Then, a new preference rule is created that states

> If the current problem satisfies WP
> Then the best operator to apply is Op_1.

The creation and application of this preference rule constitutes an inductive leap. To see this, consider how LEX2 handles the following operator:

$$\text{OP3: } \int cf(x)dx \Longrightarrow c\int f(x)dx$$

When LEX2 solves the problem $\int 5x^2dx$, it derives the weakest preconditions $WP : \int cx^r dx$ $(r \neq -1)$ for OP3 to be a successful operator. It then constructs the preference rule

> If the current problem matches $\int cx^r dx$ $(r \neq -1)$
> Then prefer OP3.

This preference rule recommends the wrong action when LEX2 attempts to solve the problem $\int 0x^4dx$, because the zero should instead be multiplied out to obtain $\int 0dx$. This error reveals that LEX2 has taken an inductive leap when it constructed the preference rule.[2]

Figure 1 formalizes this learning problem, which we call the theory-based concept specialization (TBCS) problem, because it involves the inductive specialization of a concept defined by a domain theory. We believe this learning problem is important, because it provides a strategy for incorporating domain knowledge into the inductive learning process. Hence, it addresses the important open problem in machine learning of how to exploit domain knowledge to guide inductive learning.

The reader may have noticed that Figure 1 does not mention the "operationality criterion," which plays a major role in the "non-inductive" applications of explanation-based generalization. This omission reflects the fact that in the TBCS problem, the focus is not

[1] Prodigy contains several other target concepts. This same argument applies, with minor modifications, to each of them.

[2] Of course, since this control error will not affect the answer that LEX2 eventually computes, the entire LEX2 system has not made an inductive leap or performed knowledge level learning. However, the subsystem of LEX2 that selects the best operator to apply *has* made such a leap, it is just not visible to an external observer. In other words, LEX2 has made a control error, but this does not cause it to produce the wrong solution to the integration problem.

Given

- A domain theory that defines a target concept, TC.
- A set of positive training examples of a concept C, where C is a specialization of TC.

Find

- A correct definition of C.

Figure 1: The theory-based concept specialization (TBCS) problem.

on improving problem-solving efficiency, but rather on identifying the correct definition for concept C. In most cases, once the correct definition is identified, a knowledge compilation step will be needed to convert it into a form that can be efficiently evaluated.

The purpose of this paper is to analyze and compare three related methods for solving this learning problem: explanation-based generalization (EBG), multiple-example explanation-based generalization (mEBG) (Kedar-Cabelli, 1985; Hirsh, 1988; Cohen, 1988; Pazzani, 1988), and induction over explanations (IOE), which is introduced in this paper. The paper demonstrates that IOE eliminates two shortcomings of EBG and mEBG. The first shortcoming is that the semantic bias implemented by EBG and mEBG is too strong, and this causes them to produce incorrect concept definitions very frequently. The second shortcoming is that EBG and mEBG are quite brittle—their success depends greatly on the choice of encoding of the domain theory rules. This makes it difficult to design a domain theory that produces correct specializations.

In this paper, we present empirical evidence to document these shortcomings, but they have been noted by several other researchers. For example, the SOAR group has found (Laird, 1986) that the encoding of the eight-puzzle problem space in SOAR critically influences the quality and generality of the chunks that are learned. In OCCAM, Pazzani found it necessary to augment the event schemas with additional features so that the learned definition of "economic sanction" included the constraint that the threatened country would pay more for the sanctioned product if purchased elsewhere (Pazzani, personal communication). Gupta (1988) also discusses these shortcomings.

To overcome these problems, IOE employs a weaker semantic bias. The weaker bias requires IOE to use more training examples than either EBG (which only uses a single example) or mEBG (which generally uses very few). However, we present experimental evidence and theoretical analysis to demonstrate that the number of training examples is still acceptably small.

We also present experimental evidence supporting the claim that IOE requires less "domain theory engineering" than either EBG or mEBG. This results from the fact that IOE's

weaker semantic bias makes it less sensitive to the form of the domain theory and more sensitive to the training examples that are presented.

This paper is organized as follows. Section 2 describes the EBG, mEBG, and IOE methods and illustrates them learning definitions of cups. Section 3 introduces four criteria by which to judge the effectiveness a method and describes an empirical study in the domain of chess that evaluates these methods according to our criteria. In Section 4 we analyze our results. Section 5 concludes with a summary of the major results and open problems for future research.

2 Explanation Based Methods

In this section we describe three methods for solving the theory-based concept specialization problem: explanation-based generalization (EBG), multiple-example explanation-based generalization (mEBG), and induction over explanations (IOE). Each of these methods, requires a domain theory, a target concept, and one or more training examples. To illustrate the methods, we will use the simple domain theory shown in Figure 2 and the three training examples shown in Figure 3.[3]

The domain theory defines the target concept `cup(Object)` as follows. A cup is any object that holds liquid, is stable, is liftable, and can be drunk from. To hold liquid, the sides and bottom of the object must be made of non-porous materials. To be stable, the bottom must be flat. To be liftable, the object must be made of light-weight materials and be graspable. There are two different ways to be graspable. One way is for the object to have a small, cylindrical shape and to be made from an insulating material. The other way is for the object to have a handle.

The three examples shown in Figure 3 are each positive examples of the cup concept. *Cup1* is a plastic cup without any handles. It is graspable because plastic is an insulating material. *Cup2* and *cup3* both have handles. Their main difference is that *cup2* has plastic sides and a metal bottom, whereas *cup3* is made entirely of china.

Below each cup is shown the symbolic description that is actually processed by the three methods. At the bottom of the figure, we present the explanation tree that is constructed for each cup. To make these trees understandable, we have given each rule in the domain theory a two-letter name, and these are shown in the explanation trees and in the domain theory.

2.1 Explanation-Based Generalization

Explanation-based generalization forms its concept definition from only one example. It proceeds as follows.

Step 1. Construct the explanation tree (proof tree) that explains why the example is an instance of the target concept.

Step 2. Compute the weakest preconditions WP such that the *same explanation* could be applied. For simple explanation trees of the type shown in Figure 3, WP is a

[3]This example is inspired by (Kedar-Cabelli, 1985).

Rule C

```
cup(Object):-
      hold_liquid(Object),
      stable(Object),
      liftable(Object),
      drinkfrom(Object).
```

Rule Hl

```
hold_liquid(Object):-
      sides(Object,S),
      made_from(S,Ms),
      non_porous(Ms),
      bottom(Object,B),
      made_from(B,Mb),
      non_porous(Mb).

non_porous(plastic).
non_porous(china).
non_porous(metal).
non_porous(styrofoam).
```

Rule Df

```
drinkfrom(Object):-
      has(Object,Ob1),
      concavity(Ob1),
      upward_pointing(Ob1).
```

Rule St

```
stable(Object):-
      bottom(Object,B),
      flat(B).
```

Rule Li

```
liftable(Object):-
      light_weight(Object),
      graspable(Object).
```

Rule Lw

```
light_weight(Object):-
      small(Object),
      sides(Object,S),
      made_from(S,Ms),
      light_material(Ms),
      bottom(Object,B),
      made_from(B,Mb),
      light_material(Mb).

light_material(plastic).
light_material(china).
light_material(metalsheet).
light_material(styrofoam).
```

Rule Gr1

```
graspable(Object):-
      small(Object),
      sides(Object,S),
      cylindrical(S),
      made_from(S,M),
      insulating_material(M).

insulating_material(styrofoam).
insulating_material(plastic).
```

Rule Gr2

```
graspable(Object):-
      small(Object),
      has(Object,O1),
      handle(O1).
```

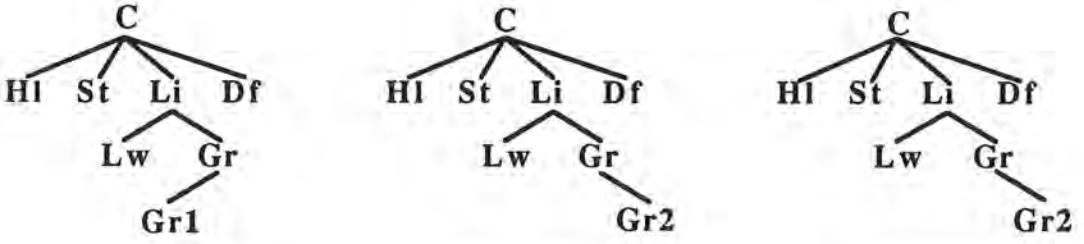
Figure 2: Cup Domain Theory

cup1 *cup2* *cup3*

```
sides(cup1,s1).
made_from(s1,plastic).
bottom(cup1,b1).
made_from(b1,plastic).
flat(b1).
has(cup1,c1).
concavity(c1).
upward_pointing(c1).
small(cup1).
cylindrical(s1).
```

```
sides(cup2,s2).
made_from(s2,plastic).
bottom(cup2,b2).
made_from(b2,metal).
flat(b2).
has(cup2,c2).
concavity(c2).
upward_pointing(c2).
small(cup2).
has(cup2,h2).
handle(h2).
```

```
sides(cup3,s3).
made_from(s3,china).
bottom(cup3,b3).
made_from(b3,china).
flat(b3).
has(cup3,c3).
concavity(c3).
upward_pointing(c3).
small(cup3).
has(cup3,h3).
handle(h3).
```



EBG(cup1) *EBG(cup2)* *EBG(cup3)*

Figure 3: Cup Examples

conjunction of the literals that appear as the leaves of the explanation tree. However, the terms that appear as the arguments in those literals must be carefully selected so that they are as general as possible and yet still guarantee that the consequent of each rule appearing in the tree will unify with the antecedent of the appropriate rules above it in the tree. This can be accomplished by constructing the explanation tree from the domain theory rules, this time performing only those unifications needed to reconstruct the tree itself and omitting any unifications with the training example. There are many refinements of this procedure. For example, in systems where unifications can be "undone," it suffices to undo all unifications between the domain theory and the training example. See Mooney & Bennett, 1986 and Kedar-Cabelli & McCarty, 1987 for more details.

To see how this method works, consider applying EBG to *Cup1* from Figure 3. Like all cups, this one holds liquids, is stable, light-weight, and can be drunk from. The interesting aspect of this particular cup is that it lacks a handle. However, it is still graspable because it is small, cylindrical, and the sides are made of an insulating material. It is these general properties that are identified by EBG, as a result of analyzing the explanation tree. Here are the weakest preconditions that it discovers:[4]

```
sides(Object,S),cylindrical(S),madefrom(S,Ms),
non_porous(Ms),light_material(Ms),insulating_material(Ms),
bottom(Object,B),flat(B),madefrom(B,Mb),non_porous(Mb),
light_material(Mb),small(Object),has(Object,Ob),
concavity(Ob),upwardpointing(Ob).
```

In the TBCS problem, these weakest preconditions form the definition for a new concept, C. This concept C is clearly a specialization of the target concept ("Cup"), because it describes only cups with cylindrical sides made of light, insulating material.

If different training examples are given to EBG, different weakest preconditions will be computed. To see this, consider applying EBG to *cup2*. This cup has plastic sides, a metal bottom, and a handle. Consequently, a different rule for `graspable` is applied from the domain theory, and EBG forms a concept definition that covers only cups that have handles and removes the restriction that the material must be insulating.

In these two cases, EBG forms different concept descriptions because each example requires different domain theory rules in its explanation. When learning from *cup1*, rule **Gr1** must be applied to prove that the object is graspable. When learning from *cup2*, rule **Gr2** must be applied to prove that the object is graspable. Because the domain theory rules appearing in the explanation determine the weakest preconditions computed by EBG, these two different explanations yield two different concept definitions.

This observation makes it possible to characterize the space of all concept definitions that EBG can discover. Let us say that an explanation tree is "complete" if it provides a proof connecting the target concept to the predicates that are provided in the training examples. For every distinct complete explanation tree that can be constructed from the

[4]For illustrative purposes we are assuming that ground atomic formulas in the domain theory (i.e., `non_porous`, `light_material` and `insulating_material`) are "compiled" knowledge (in the sense given in Braverman & Russell, 1988) and not included in the EBG definition.

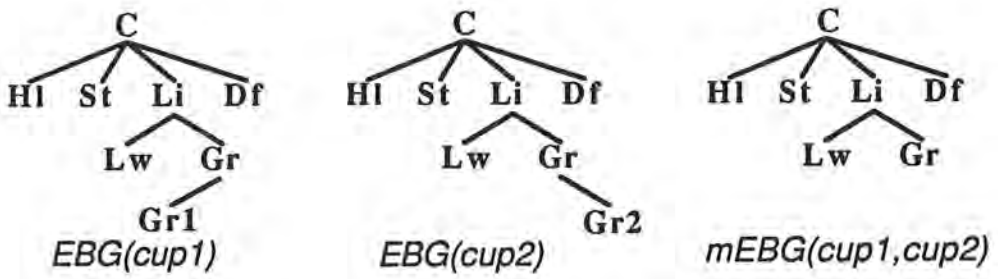


Figure 4: mEBG learning from *cup1* and *cup2*

domain theory, EBG will construct a different concept definition. In our illustrative cup domain theory, there are only two distinct complete explanation trees, so EBG can only construct two distinct concept definitions.

Utgoff (1986) and Haussler (1988) define the strength of an inductive bias in terms of the size of the space of hypotheses permitted by the bias. In the TBCS problem, the space of hypotheses permitted by the semantic bias is precisely the space of definitions that EBG can construct. Therefore, when we apply EBG to solve the TBCS problem, we are employing a very strong bias indeed!

2.2 Multiple Explanation-Based Generalization

The second method that we wish to consider is the multiple-example explanation-based generalization (mEBG) method described in Kedar-Cabelli (1985), Hirsh (1988), Pazzani (1988), and Cohen (1988). The method differs from EBG in that the final concept definition is identified from multiple examples, rather than from only a single example. The method, given two or more examples of the target concept, proceeds as follows.

Step 1. For each example, mEBG constructs an explanation tree that proves that the example is an instance of the target concept.

Step 2. It then compares these trees to find the largest subtree that is shared by all of the examples.[5]

Step 3. Finally, it applies the EBG generalization technique to this maximal common subtree to generalize the terms appearing in the tree and extract the weakest preconditions such that the tree remains a valid proof.

To illustrate the method, consider giving mEBG the first two training examples of cup: *cup1* and *cup2*. Figure 4 shows the (schematic) explanation trees for each example and the maximal common explanation tree computed in Step 2. Notice that because *cup1* lacks a

[5]An alternative view of this step (Cohen, 1988) is that mEBG forms a combined AND/OR proof tree, where OR nodes are introduced at each point in the tree where different domain theory rules were applied. This technique will be equivalent to the pruning technique described here only when enough training examples are presented to cause every applicable domain theory rule to be included in the OR-node.

handle, its explanation must involve rule **Gr1** (insulated, small cylinder), whereas *cup2* has a handle, so its explanation must involve rule **Gr2** (small with handle). Because the rules differ, neither is included in the common explanation tree. As a consequence, Step 3 of the mEBG procedure produces the following weakest precondition:

```
graspable(Object),sides(Object,S),madefrom(S,Ms),
non_porous(Ms),light_material(Ms),
bottom(Object,B),flat(B),madefrom(B,Mb),non_porous(Mb),
light_material(Mb),small(Object),has(Object,Ob),
concavity(Ob),upwardpointing(Ob).
```

The only difference between this definition and the one produced by EBG from *cup1* is that the conditions `cylindrical(S)` and `insulating_material(Ms)` have been deleted and replaced by the condition `graspable(Object)`. This change reflects the fact that the new definition is *uncommitted* about the way the graspable goal is satisfied. Whenever this definition is applied to new examples, it will be necessary to consult rules **Gr1** and **Gr2** to determine whether the `graspable(Object)` condition is satisfied.

The space of concept definitions that mEBG can produce strictly includes the space of definitions computed by EBG. Recall that EBG will produce a different concept definition for each distinct complete explanation tree that can be constructed in the domain theory. The mEBG method expands this set by also permitting *incomplete* explanation trees. It will construct a different concept definition for each distinct incomplete explanation tree.

Incomplete explanation trees need not relate the target concept to the predicates given in the training examples. Instead, the leaves of an incomplete tree can terminate at any point where there is a disjunction in the domain theory. The reason is that by presenting one training example for each branch of the disjunction, we can force the mEBG method to prune all rules at or below the disjunction when it constructs the maximal common subtree.

In the cup domain theory, there is only one disjunction (`graspable`), so the space of definitions that can be constructed by mEBG from this domain theory includes only 3 definitions: the two constructed by EBG and the third definition shown above.

2.3 Induction Over Explanations

Like mEBG, the induction over explanations (IOE) method is also a method for learning from multiple examples. The key difference is that IOE employs a different strategy for generalizing the maximal common explanation tree to obtain the concept definition. To simplify the notation, we describe IOE applied to only two training examples, TI_1 and TI_2:

Step 1. Apply the mEBG method to the two training examples TI_1 and TI_2. This produces a concept definition C_{mEBG}. Retain the original explanation trees for use in step 2.

Step 2. Match C_{mEBG} to the saved proof tree for each training example TI_i to produce a substitution, θ_i (where θ_i consists of a set of pairs $v_j = c_j$; each v_j is a variable in C_{mEBG}, and each c_j is a constant from TI_i or the domain theory).

Step 3. Form θ, a new substitution for C_{mEBG} as follows:

1. Set θ to $\emptyset$.
2. For each variable, v_j in C_{mEBG} put $v_j = c_j$ in θ, where c_j is computed as follows:
   (a) Lookup $v_j = c1_j \in \theta_1$, and $v_j = c2_j \in \theta_2$.
   (b) $c_j \leftarrow gen(c1_j, c2_j)$, where $gen(c1, c2)$ is defined as follows: If $c1 = c2$ then $gen(c1, c2) = c1$.
   Otherwise, if $c1 = f(t_{1,1}, t_{1,2}, \ldots, t_{1,k})$ and $c2 = f(t_{2,1}, t_{2,2}, \ldots, t_{2,k})$ (i.e., terms with the same initial function symbol f), then $gen(c1, c2) = f(gen(t_{1,1}, t_{2,1}), gen(t_{1,2}, t_{2,2}), \ldots, gen(t_{1,k}, t_{2,k}))$. (In other words, *gen* is applied recursively to each pair of corresponding arguments of f.)
   Otherwise, $gen(c1, c2) = \nu$, where ν is selected as follows. If there has never been a previous call to *gen* with these same arguments $c1$ and $c2$, then ν is a new variable that does not appear anywhere in θ_1, θ_2, or θ. The information that $gen(c1, c2) = \nu$ is stored in a table for future use. If there has been a previous call to *gen* with the same arguments, then this table is consulted, and the previously generated variable is returned as ν.

Step 4. Compute and return the $C_{mEBG}\theta$, the substitution applied to the mEBG definition.

In this procedure, IOE begins by computing the mEBG concept definition. However, it then specializes this mEBG concept definition by introducing additional constraints on the variables appearing in the definition. These new constraints are imposed by the substitution θ.

Two kinds of constraints are introduced. The first kind is a *constant constraint.* It is introduced whenever a given variable v appearing in the mEBG definition is always bound to the same constant c in *every* training example. When this occurs, IOE binds v to c in the concept description.

To illustrate this, suppose we apply IOE to the two training examples *cup1* and *cup2* (see Figure 3). In the mEBG definition computed by Step 1, the literal `madefrom(S,Ms)` appears, where `Ms` is the material making up the sides of the cup. Because both *cup1* and *cup2* have plastic sides, IOE will introduce the constraint `Ms = plastic`, so the final concept description will require that the cup sides be plastic.

The second kind of constraint introduced by IOE is an *equality constraint* that forces two or more variables appearing in the mEBG definition to be equal to identical terms. An equality constraint is introduced whenever IOE finds two (or more) different variables v_1 and v_2 in the mEBG definition that are always bound to the same term c_i in each training example TI_i. Notice that the term c_i can differ from one training example to the next. However, within each example TI_i, v_1 and v_2 are both bound to the same value c_i.

To illustrate this, suppose we apply IOE to training examples *cup1* and *cup3*. Recall that both the bottom and the sides of *cup1* are made of plastic. Similarly, the bottom and the sides of *cup3* are made of china. In the mEBG definition computed in Step 1, the two literals `madefrom(S,Ms)` and `madefrom(B,Mb)` appear, where `Ms` is the side material and `Mb` is the bottom material. In the first training example, `Ms` and `Mb` are both bound to `plastic`, while in the second training example, `Ms` and `Mb` are both bound to `china`. Therefore, IOE introduces a new variable `M` and changes the final definition so that it includes the two literals

Variable	Training Examples			Learning output	
v	*cup1*	*cup2*	*cup3*	*IOE(cup1,cup2)*	*IOE(cup1,cup3)*
`Object`	`cup1`	`cup2`	`cup3`	`Obj`	`Obj`
`S`	`side1`	`side2`	`side3`	`Side`	`Side`
`Ms`	`plastic`	`plastic`	`china`	`plastic`	`M`
`B`	`bottom1`	`bottom2`	`bottom3`	`Bottom`	`Bottom`
`Mb`	`plastic`	`metal`	`china`	`M`	`M`
`Ob1`	`con1`	`con2`	`con3`	`Con`	`Con`

Table 1: Table showing the results of applying IOE to *cup1, cup2* and *cup3*

`madefrom(S,M)` and `madefrom(B,M)`. This definition describes "homogeneous" cups, that is, cups made entirely of a single material.

Both kinds of constraints are computed by the *gen* procedure. Table 1 shows how this works in more detail. The first column of the table lists the six variables that appear in C_{mEBG}. The set of columns labelled "Training Examples" shows the substitutions θ_i for each of the three training examples, *cup1*, *cup2*, and *cup3*. The final two columns show the results of applying IOE either to *cup1* and *cup2* or to *cup1* and *cup3*.

Consider the last column of the table, where we are computing $IOE(cup1, cup3)$. During Step 3, the *gen* procedure will be applied one row at a time. First, IOE computes *gen*(`cup1`,`cup3`). Because the two constants differ, IOE creates a new variable, `Obj`, and adds the pair `Object = Obj` to the final substitution θ.

The second call to *gen* is with arguments `side1` and `side3`. A new variable `Side` is created, and the pair `S = Side` is added to θ.

The third call to *gen* is the one that introduces an equality constraint. The arguments are `plastic` and `china`. As before, a new variable, `M` is created, and the pair `Ms = M` is added to θ. As always, however, the information that *gen*(`plastic`,`china`) =`M`, is stored away for future reference. When IOE reaches the fifth line of Table 1 (the line for `Mb`), it will consult the stored information and add the pair `Mb = M` to θ, thus imposing the equality constraint.

The layout of Table 1 reveals an important way of thinking about the IOE procedure. We can think of each variable appearing in C_{mEBG} (the left-most column) as a simple "feature." Each substitution θ_i can be viewed as a feature vector. According to this perspective, the generalization procedure of Step 3 is the well-known algorithm for computing the maximally-specific conjunctive generalization of a collection of feature vectors by turning constants to variables. The only subtlety is the technique for introducing equality constraints. We call this technique the "no-coincidences" bias, because it assumes that if the pair (`plastic`, `china`) appears in more than one row in the table, this must indicate an equality constraint rather than just a coincidence.[6]

Given that IOE is applying such a simple inductive generalization procedure, what is the source of power of the method? The answer is that unlike traditional inductive learning techniques, IOE is not attempting to find patterns in the training examples as they are originally presented to the learning system. Instead, it applies the domain theory and the

[6]Subsequent to implementing IOE, we discovered that Plotkin (1970) had already developed and formalized this no-coincidences bias.

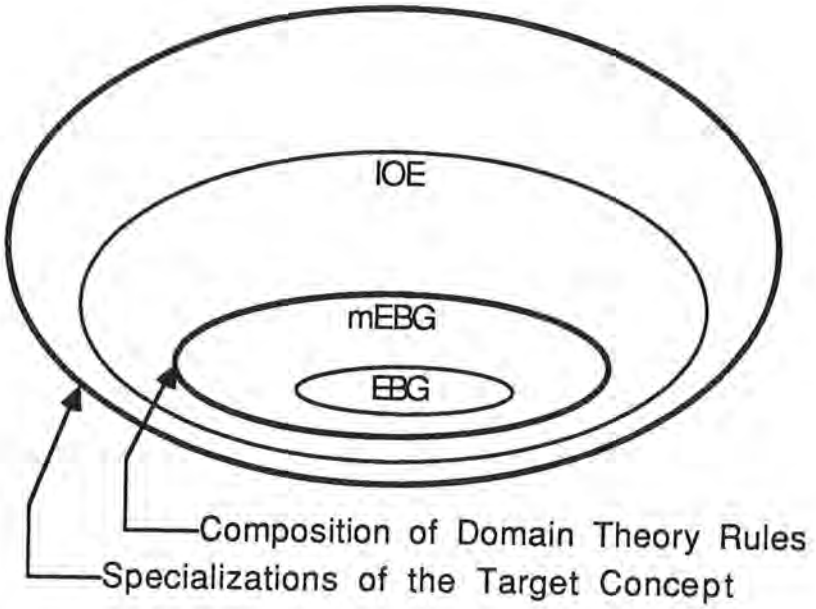


Figure 5: Concept spaces of EBG, mEBG and IOE

target concept to identify a useful set of features for inductive generalization (this point has been made by Pazzani, 1988). In IOE, the substitutions θ_i re-express the training examples in terms of these useful features. In later sections of the paper we show that it is this ability to reformulate the training examples in terms of relevant features, rather than the particular generalization strategies employed, that is the source of power underlying all explanation-based learning methods.

What is the space of concept definitions that can be constructed by IOE? For every possible definition constructed by mEBG, IOE can generate any legal substitution, where the substitution constrains the definition through a combination of constant constraints and equality constraints. In the cup domain, this use of constraints allows IOE to generate 68 different definitions of cups including the two above: cups that have plastic sides and cups that are homogeneous. Among these 68 definitions are the three definitions that mEBG discovers. These are produced when the training examples have different combinations of values in every row of Table 1 so that *gen* produces a distinct variable for each row.

To summarize, we illustrate the three spaces in Figure 5. Note that all three methods implement a semantic bias, that is, form a definition C that is some specialization of the initial target concept TC. EBG builds the smallest space of specializations, each corresponding to a distinct complete explanation tree. The mEBG method offers a larger space of specializations that includes the space of EBG, because it considers incomplete explanation trees as well. IOE offers a much larger space, because it is able to specialize each mEBG definition in many different ways, depending on the configuration of constants appearing in the training examples.

3 Comparative Study

In this section we describe an empirical evaluation of the three methods EBG, mEBG, and IOE applied to the theory-based concept specialization (TBCS) problem. The section begins by defining four evaluation criteria. This is followed by an outline of the experiments and a description of the test domain and the test domain theories. The section concludes with the results of the experiments.

3.1 Evaluation Criteria

Because the TBCS problem is a problem of learning from examples, it is appropriate to consider the two traditional criteria for such learning methods: correctness (the method should find the correct concept C) and learning efficiency (the method should require a small number of training examples and few computational resources).

In addition to these two criteria, we want to consider other criteria that assess the ease with which each learning method can be applied to new problems and to new domains. To do this, we borrow from software engineering the idea of the "life cycle" of a program. In particular, let us make a distinction between the *design phase* of a learning system and the *learning phase* of the system. During the design phase of a TBCS system, a learning method is chosen and implemented, a vocabulary and target concept TC are selected, and a domain theory is written for TC in terms of this vocabulary. During the learning phase, a collection of training examples for some specialized concept C is presented to the system, and a definition for C is constructed.

This simple life cycle model suggests two additional criteria for evaluating learning methods: ease of engineering (during the design phase) and flexibility (during the learning phase). A method is easy to engineer if it is not very sensitive to the vocabulary and the exact form of the domain theory. This makes it easier to design the vocabulary and write the domain theory. A method is flexible to the extent that the user can change the unknown concept C, provide a new set of training examples, and still find a correct definition for C without having to change the domain theory, the vocabulary, or the target concept.

The four criteria employed in this study can be summarized as follows:

Ease of Engineering (design phase): The learning system should be easy to construct. It should not require the careful design of the domain theory or the careful choice of the target concept in order to be effective during the learning phase.

Learning Efficiency (learning phase): The learning system should require a small number examples and few computational resources during learning.

Correctness (learning phase): The learning system should construct a correct definition for the unknown concept C.

Flexibility (learning phase): The learning system should be adaptable—that is, once designed, it should be able to learn a wide range of possible concepts during the learning phase without the need for redesign.

3.2 Experiment Outline

To evaluate each of these criteria, we designed a series of experiments as follows. To test correctness and flexibility, we chose a domain in which there were several closely related concepts. Using a fixed domain theory, we attempted to get each learning method to learn every one of these concepts. Methods that learned each concept correctly scored well on the criteria of correctness and flexibility.

To test learning efficiency, we measured the number of training examples required by each method to attain a given level of correctness. This comparison was only possible with concepts that were correctly learned by all three methods.

Finally, to test ease of engineering, we performed all of the above experiments using two different domain theories: one developed by Flann and one developed independently by Russell (1985). If the results obtained from a method vary significantly depending on which domain theory is used, then the method is judged to be difficult to engineer.

The following subsections describe the test domain, the two domain theories, and our implementation of the three methods.

3.2.1 The test domain: chess

The domain of chess was selected because it provides an excellent testbed for comparing different solutions to the TBCS problem. Chess provides many interesting concept definitions that are significant to problem solving, and it has a small, complete domain theory. Figure 6 illustrates four concepts of interest in chess: knight-fork, sliding-fork, skewer, and check-with-bad-exchange.

Figure 6(a) is an example of knight-fork. The white knight on square e6[7] is simultaneously threatening the black bishop on g5 and the black king on d8. No black piece can take the white knight, so the king will be forced to move out of check (to c8 or d7). This will permit the knight to take the bishop. Figure 6(b) is a different kind of fork. The white bishop on c6 is simultaneously threatening the black rook on a8 and checking the king on e8. We call this fork a "sliding-fork", since the threatening piece can move through multiple squares. Figure 6(c) is an example of a "skewer." The black rook is checking the king on e4 who is forced to move out of check and expose the queen on b4 to capture. Notice that the captured piece is behind the king. Figure 6(d) shows a further variation, where different pieces are used in the check and capture. The black rook on c6 checks the white king on c2, while the black bishop on c8 threatens the knight on g4.

Notice that in each case, the king is in check and the side to move suffers a bad exchange of material in two ply. Hence, all of these concepts are specializations of the concept check-with-bad-exchange, where a piece $P1$ checks the king and forces it to move out of check, which allows a piece $P2$ to capture an opponent's piece Po. Figure 7 illustrates the relationships among the various different concepts. In a fork, the same piece ($P1 = P2$) is used both to threaten the king and make the capture. A skewer is a further specialization of a fork, because the captured piece Po is hidden behind the king.

[7] Each square on the board is denoted by a unique name. The first letter denotes the column, with the left-most column being a and the right-most column being h. The second number denotes the row, with the bottom row being 1 and the top row being 8.

a) Knight-Fork: black to play

b) Sliding-Fork: black to play

c) Skewer: white to play

d) Check-With-Bad-Exchange: white to play

Figure 6: Examples of related chess concepts

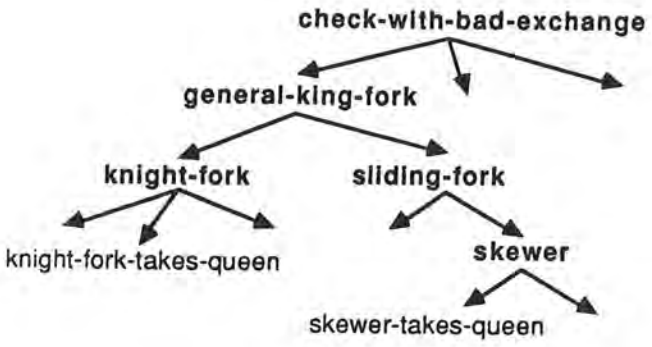


Figure 7: Related chess concepts

These specializations of check-with-bad-exchange are interesting because they exhibit important tactical properties. For example, because Po in a skewer hides behind the king, the threatening piece $P1$ is not itself threatened by Po. This allows a weak, unprotected piece $P1$, such as a bishop or rook, to capture a powerful piece Po such as the queen. The concept knight-fork is interesting because the check threat cannot be blocked by the opponent—to avoid the loss of Po, it is necessary to capture the knight. In contrast, in a sliding-fork the check threat may be blocked, allowing the opponent the potential to mitigate the loss of Po through an exchange.

For the purpose of our experiments, we chose the five concepts skewer, sliding-fork, knight-fork, general-king-fork, and check-with-bad-exchange to challenge the learning methods. The target concept for each of these is the concept of check-with-bad-exchange.

3.2.2 The domain theories

The two domain theories employed in the experiment were both designed to be very general, easily understood encodings of the legal moves of chess. In particular, the domain theory written by Flann (referred to as DT_{flann}) employs a very general representation of the chess board as a collection of 64 independent squares. Rather than introducing the notions of rows, columns, and diagonals, DT_{flann} simply states how the squares are connected to one another along the eight directions of the compass. The complete domain theory is given in Appendix 8.2.

The domain theory written by Russell (denoted $DT_{russell}$) was originally developed as an exercise in logic programming. It appeared as an appendix to "The Compleat Guide to MRS" (Russell, 1985).[8] In $DT_{russell}$, squares are represented by column (x) and row (y) coordinates, and moves are computed using vectors. The domain theory includes definitions of discovered check, of pinned piece, and of moving the king out of danger. In many ways, $DT_{russell}$ is more "engineered" than DT_{flann}, because it employs more special case analysis in its rules (such as how to move a half-pinned-piece). The complete domain theory is given in Appendix 8.3.

3.2.3 The *Wyl2* implementation

The three methods EBG, mEBG, and IOE have been implemented in a learning system called *Wyl2*. Like PrologEBG (Kedar-Cabelli & McCarty, 1987), *Wyl2* is an extended meta-interpreter for Prolog. In addition to the usual Prolog meta-logical operations (such as cut), *Wyl2* also includes special forms of universal and existential quantification, which are required to express the adversarial search tree in chess. In (Flann, 1988b), we describe the logical language employed in *Wyl2* and the modifications to the generalization step required to deal with universal quantification.[9]

Wyl2 actually contains two different implementations of EBG. The first, which we call EBG-, is the simple 13-line Prolog-EBG algorithm given in (Kedar-Cabelli & McCarty, 1987).

[8] Russell's domain theory was changed slightly for this test. First, the code was translated from MRS to Prolog. Second, the legal move generator was changed to use the `Op` notation (see Appendix 8.1), and frame axioms were written for the primitive board predicates. Finally, a definition of *in-check* was written, since it was missing from the original theory.

[9] *Wyl2* is written in Quintus Prolog. Contact the authors for distribution information.

It creates the complete explanation tree and then computes its weakest preconditions as described above. The other implementation, which we call EBG*, incorporates two techniques designed to improve its performance in the chess domain. First, it applies generalization-to-n techniques (Shavlik & DeJong, 1987) to generalize over the distance that pieces may move. This allows EBG* to generalize over the distance between the king and a piece that is checking it, or more generally, between any two pieces where one piece is threatening the other.

The second addition is that EBG* automatically prunes branches from the explanation tree to allow it to generalize over the type of piece being moved.[10]

3.3 Experimental Results

3.3.1 Correctness and Flexibility

Table 2 shows the concepts that were learned by each learning method when presented with examples of each of the five test concepts from the chess domain.

To test EBG- and EBG*, we performed 20 learning trials for each of the five test concepts. In each trial, we selected one training example at random from the space of positive examples of the desired test concept and gave it to the algorithm. EBG* learns the same concept in every trial, regardless of the desired concept or the domain theory. EBG-, in contrast, learns different concepts depending on the specific configuration of pieces in the training example. In particular, the distances between the starting and ending squares of each move are fixed constants in the concepts learned by EBG-, because it is unable to generalize to n. This means that all threats and checks are also constrained to fixed distances, since these are potential moves. In Table 2, we have summarized the different concepts learned by EBG- by indicating that they are "fixed-distance" versions of **check-with-bad-exchange**, **check-with-bad-exchange-by-knight**, and **check-with-bad-exchange-by-sliding-piece**. Even allowing for the "fixed-distance" problem, only one of these definitions was learned correctly.

For mEBG and IOE, we provided the methods with 50 positive examples randomly chosen from the space of positive examples of the desired concept.

Notice that of the four methods, only IOE correctly learns each of the five test concepts. All of the other methods essentially overgeneralize to the **check-with-bad-exchange** concept. None of the definitions learned by EBG-, EBG*, or mEBG includes the equality constraint that the same piece that checks the king must also be the piece that makes the capture, which is needed to express everything except **cwbe**.

Because none of the methods except IOE learn the desired test concepts, none of them perform well according to the correctness and flexibility criteria.

Notice also that the results obtained from EBG- and mEBG vary depending on the domain theory employed. This is evidence that these methods are sensitive to the form of the domain theory.

3.3.2 Learning Efficiency

[10] See the two domain theories in the Appendix (Page 34 and Page 39) for details of exactly which predicates where chosen for prunning.

Correct Concept	Learned Concept							
	EBG-		EBG*		mEBG		IOE	
	DT_{flann}	$DT_{russell}$	DT_{flann}	$DT_{russell}$	DT_{flann}	$DT_{russell}$	DT_{flann}	$DT_{russell}$
knight-fork	cwbe-kn*	cwbe-kn*	cwbe	cwbe	cwbe-1	cwbe-kn	knight-fork	knight-fork
skewer	cwbe*	cwbe-s*	cwbe	cwbe	cwbe	cwbe-s	skewer	skewer
sliding-fork	cwbe*	cwbe-s*	cwbe	cwbe	cwbe	cwbe-s	sliding-fork	sliding-fork
general-king-fork	cwbe*	cwbe*	cwbe	cwbe	cwbe	cwbe	gkf	gkf
cwbe	cwbe*	cwbe*	cwbe	cwbe	cwbe	cwbe	cwbe	cwbe

Table 2: Concepts learned by different methods (Key: cwbe: check-with-bad-exchange; cwbe-s: cwbe with sliding pieces only; cwbe-kn: cwbe with knight only; cwbc-1: cwbc with single length check and capture; gkf: general-king-fork). Concepts marked * restrict the direction of the moves or the exact length of the moves in the definitions

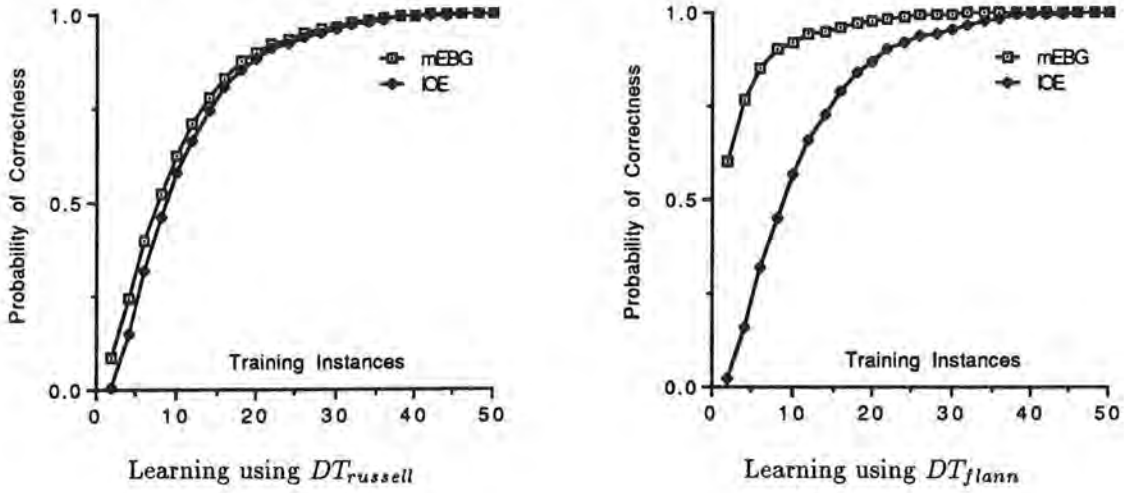
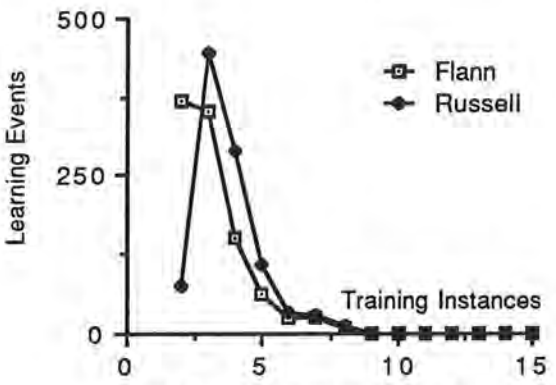


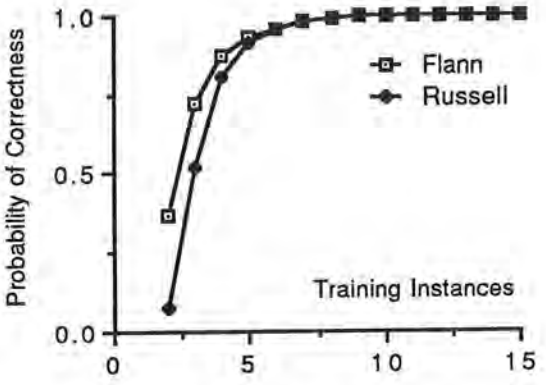
Figure 8: A comparison of the learning efficiency of IOE, mEBG and EBG* learning check-with-bad-exchange from randomly chosen examples.

Number of Training Instances	Learning Method					
	EBG*		mEBG		IOE	
	DT_{flann}	$DT_{russell}$	DT_{flann}	$DT_{russell}$	DT_{flann}	$DT_{russell}$
Average	1	1	4.1	10.2	11.8	11.7
To 90% correct	1	1	8	21	22	22
To 99% correct	1	1	25	37	38	38

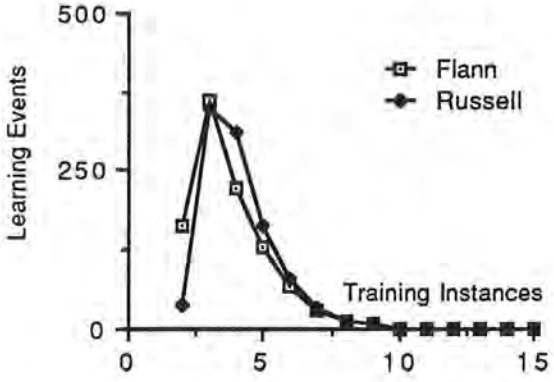
Table 3: A summary of the learning efficiency for check-with-bad-exchange exhibited by different learning methods and domain theories



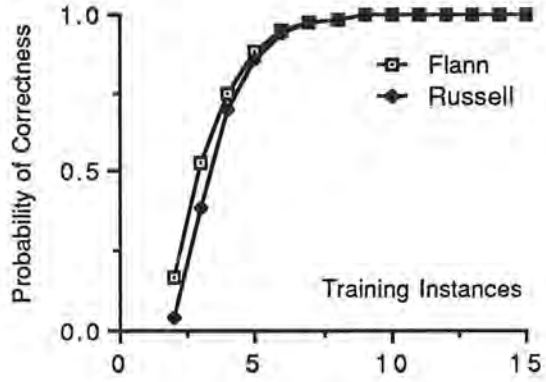
Knight-Fork: Histogram of learning correct concepts



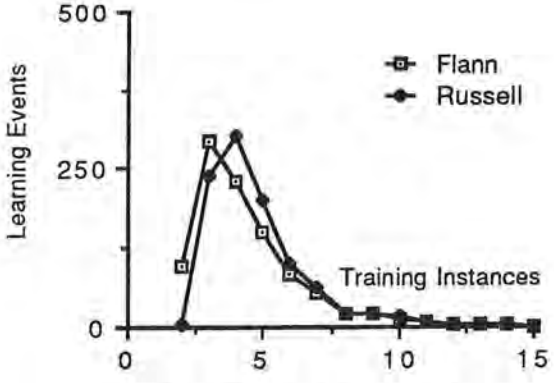
Knight-Fork: Learning curve



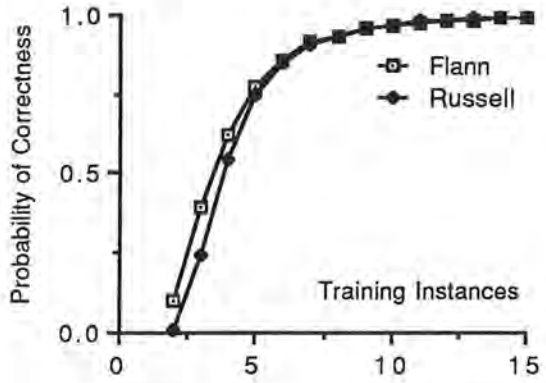
Skewer: Histogram of learning correct concepts



Skewer: Learning curve



Sliding-Fork: Histogram of learning correct concepts



Sliding-Fork: Learning curve

Figure 9: An evaluation of IOE learning efficiency. The curves on the left are histograms showing the number of trials (out of 1000) in which exactly m randomly-selected training examples were needed to correctly learn the concept. The curves on the right show the resulting learning curves giving the probability of correctness as a function of the number of training examples.

Concept Learned	Number of Training Instances					
	Average		To 90% Correct		To 99% Correct	
	DT_{flann}	$DT_{russell}$	DT_{flann}	$DT_{russell}$	DT_{flann}	$DT_{russell}$
knight-fork	3.2	3.8	5	5	8	8
skewer	3.7	4.1	6	6	9	9
sliding-fork	4.8	4.9	7	7	14	14
general-king-fork	11.0	11.3	21	22	35	35
check-with-bad-exchange	11.8	11.7	22	22	38	38

Table 4: Summary of learning efficiency exhibited by IOE in *Wyl2*

Learning efficiency has two aspects: (a) the number of training examples required and (b) the computational cost. All of the methods have low computational costs, so our evaluation focuses on the number of training examples needed.

To compare the learning efficiency of the three methods, we selected the one concept that they were all able to learn, **check-with-bad-exchange**, and performed 1000 learning trials with each method. In a learning trial, the learning method (EBG*, mEBG, or IOE) is repeatedly given a randomly-selected positive example of the desired concept and then tested to see whether it has learned the concept correctly. When the concept is correctly learned, the trial terminates, and the number of training examples used is recorded.

Table 3 shows that in general, IOE requires more examples than mEBG. mEBG and IOE both require many more examples than EBG* (which of course requires only one example). Interestingly, the efficiency of mEBG varies depending on the domain theory being used. For DT_{flann}, mEBG is much more efficient than it is for $DT_{russell}$. This is evidence that mEBG is more sensitive to the form of the domain theory than IOE. To visualize this sensitivity, examine Figure 8, which shows learning curves for mEBG and IOE on both domain theories. Notice that for $DT_{russell}$, the methods are virtually identical, whereas for DT_{flann}, the two methods differ significantly.

To produce these curves, we first construct a histogram showing the number of learning trials (out of the 1000 trials) in which exactly m training examples were required to correctly learn the concept. Then, we integrate the histogram to compute the number of trials n that required less-than-or-equal-to m training examples to obtain correct performance. The quantity $n/1000$ gives the probability that after processing m training examples the concept has been correctly learned. Note that these learning curves do *not* indicate the percentage of training examples that the partially-learned concept definitions would classify correctly. From statistical tests the learning curves are accurate to $\pm 3\%$ with 95% confidence.[11]

All of the efficiency results presented so far were measured only for the most general concept, **check-with-bad-exchange**. For IOE, we can also evaluate learning efficiency on the other test concepts. We summarize the learning efficiency of IOE in Table 4 and show histograms and learning curves for **knight-fork**, **skewer**, and **sliding-fork** in Figure 9. A difference in efficiency between $DT_{russell}$ and DT_{flann} is observed in all cases, although its magnitude varies. IOE consistently learns faster when using DT_{flann} than when using $DT_{russell}$, although the difference amounts to less than one training example. This difference is much smaller than

[11] Confidence limits are calculated from formulae given in (Spiegel, 1975, page 196).

the difference in performance observed when mEBG was applied to the two domain theories.

3.3.3 Ease of Engineering

The results on correctness show that all of the methods except IOE would require engineering of the domain theory in order to obtain the correct results. This is because for the two domain theories that we tested, EBG-, EBG*, and mEBG did not learn the desired concepts. Because additional training examples will not change the outcomes produced by these methods, this means that the only way to obtain correct performance would be to modify the domain theories.

To a lesser extent, the correctness and efficiency data also demonstrate the sensitivity of EBG-, EBG*, and mEBG to the exact form of the domain theory, because different results are obtained depending on which domain theory is employed.

IOE on the other hand, is able to learn the correct concepts in all cases, and its efficiency does not vary significantly from one domain theory to the other. Therefore, IOE appears to require much less engineering of the domain theory to obtain correct results.

3.4 Summary

Correctness and Flexibility From the results in Table 3 IOE exhibits higher correctness than the other methods. IOE learned the correct concept each time, while EBG-, EBG*, and mEBG learned the correct concept in only one case.

Learning Efficiency The results summarized in Table 3 and Table 4 demonstrate that IOE has a slightly lower learning efficiency than mEBG. The difference is small however, with IOE usually requiring a few additional training examples. The results in Table 4 show that IOE requires few training examples to converge to correctness. Whether the number required would be unreasonable depends upon the application. However, we can claim that the number required is only a small fraction of the total possible. For example, there are approximately 13×10^3 **knight-forks**, and only 8 randomly selected examples are needed to achieve 99% correctness; there are approximately 13×10^4 possible **sliding-forks**, and only 14 randomly selected examples are needed to achieve 99% correctness.

Ease of Engineering The results suggest that IOE is easier to engineer than either mEBG or EBG. IOE performed well with domain theories that were not specially designed for this learning problem or the method.

4 Analysis

This section attempts to explain the experimental results by reconsidering how each of the three learning methods works. First, we address the three issues of correctness, flexibility and ease of engineering. Then, we look at the question of learning efficiency in IOE.

4.1 Correctness, Flexibility, and Ease of Engineering

4.1.1 EBG and mEBG

Let us begin by considering why EBG and mEBG perform so poorly on the correctness criterion. The answer is simple: only one of the five test concept definitions is included in the hypothesis spaces generated by EBG and mEBG (namely, check-with-bad-exchange). Hence, it is not surprising that the remaining four concepts are not learned correctly.

Recall that a concept definition can be produced by EBG or mEBG only if it can be defined as the weakest preconditions of a (possibly incomplete) proof tree for the target concept. The space of possible proof trees can be generated by constructing all AND-trees involving the rules from the domain theory. If we look at DT_{flann}, there is only one rule for computing the legal moves of a piece. Consequently, the weakest preconditions of any proof tree containing this rule will generalize over *any type* of piece. This prevents EBG and mEBG from discovering that the checking piece must be a knight in a knight fork or a sliding piece in a sliding fork.

Furthermore, EBG and mEBG will not introduce an equality constraint unless there is a single rule somewhere in the proof tree that forces two variables to be equal. Neither domain theory includes such a rule, since there is nothing in the rules of chess that would require such a constraint. This prevents EBG and mEBG from discovering that in any fork or skewer, the piece that checks the king must be the same piece that takes the queen (or other valuable piece).

This analysis also explains why mEBG learns different concepts with $DT_{russell}$ than it does with DT_{flann}. In $DT_{russell}$, there are two different rules for determining the legal moves of a piece: one rule for knights and one rule for sliding pieces. Hence, when presented with examples of knight-fork, mEBG does include the constraint that the checking piece must be a knight. And when presented examples of sliding-fork, mEBG does include the constraint that the checking piece must be a sliding piece. However, the learned concepts cwbe-kn and cwbe-s are still incorrect because the equality constraint is missing.

This analysis also shows how correctness can be obtained from EBG and mEBG. All that is needed is to re-design the domain theory so that the rules in the theory introduce the necessary constraints. For example, we could introduce a separate rule to generate legal moves for pieces that have been checking the king during the preceding move. This would introduce the equality constraint that we need for learning forks and skewers.

Such an approach is unacceptable however, because it virtually requires us to know what concepts we are trying to learn before we design the domain theory. Furthermore, it makes the learning system completely inflexible, since the domain theory must be redesigned for each concept.

Hence, we see that for EBG and mEBG, we can obtain correctness only by sacrificing ease of engineering and flexibility. On the other hand, if we do not carefully design the domain theory, it is unlikely to contain the constraints needed to learn the correct concept definitions. For EBG and mEBG there is a direct tradeoff between correctness and ease of engineering.

Feature describes:	DT_{flann} Features		$DT_{russell}$ Features	
Checking Piece	P_{check}	Playing piece	T_{check}	Type
	Sq_{check}	Square	X_{check}	X coordinate
	Ty_{check}	Type	Y_{check}	Y coordinate
Check Threat	Dir_{check}	Direction	Δx_{check}	X vector
	L_{check}	Max Length	Δy_{check}	Y vector
Capturing Piece	$P_{capture}$	Playing piece	$T_{capture}$	Type
	$Sq_{capture}$	Square	$X_{capture}$	X coordinate
	$Ty_{capture}$	Type	$Y_{capture}$	Y coordinate
Taking Move	$Dir_{capture}$	Direction	$\Delta x_{capture}$	X vector
	$L_{capture}$	Max Length	$\Delta y_{capture}$	Y vector

Table 5: Selected features of the skewer definition for both domain theories

4.1.2 IOE

Why does the IOE method score so well on the correctness criterion? The answer is simple: the space of concept definitions produced by IOE is much larger than the space produced by EBG and mEBG, and it includes all five test concepts.

The more important question to explore is why IOE learns the correct concept when using two quite different domain theories. To answer this question, recall that IOE operates by taking the concept definition produced by mEBG (C_{mEBG}) and using it to define a vector of features. Each feature corresponds to a distinct variable in C_{mEBG}. Every training example TI_i can be translated into this feature-vector representation by computing the substitution θ_i that is required to match C_{mEBG} to TI_i. IOE then computes a generalized substitution θ by computing the maximally-specific common generalization of the θ_i's.

The insensitivity of IOE to the exact form of the domain theory results from two factors. First, regardless of the domain theory, C_{mEBG} provides a useful vector of features for representing the desired constant and equality constraints. Second, the specific constant and equality constraints introduced by IOE are determined by the training examples rather than by the domain theory. In other words, IOE is more sensitive to the training examples and less sensitive to the domain theory.

To illustrate these factors, consider how the skewer concept is learned using DT_{flann} and $DT_{russell}$. Let us focus on two of the key constraints in skewer: the threatening piece must be a sliding piece and the direction of the check threat must be the same as the direction of the capture move. Both of these properties are correctly represented in IOE even though each domain theory represents piece types and direction differently (see Table 5).

First, let us consider the how the piece type constraint is represented. In $DT_{russell}$, sliding pieces and knight pieces have separate move rules, making it easy to enforce the sliding piece constraint. In DT_{flann}, it is not so clear how this constraint can be represented since a single rule is used for all pieces. However, one property of a piece type is the maximum number of squares that the piece can move through (Max Length in Table 5). Sliding pieces can move through a maximum of 7 squares, while a knight can only move through a maximum of 1 square. Since this property is a variable in the skewer explanations, it easy to restrict the moves to only sliding pieces. IOE employs two constant constraints that constrain the

variables L_{check} and $L_{capture}$ to be 7.

Let us now consider how the direction constraint is represented. In DT_{flann}, the direction of a move or check is defined as a single variable, Dir, that can take eight different values[12] corresponding to the points of a compass. In $DT_{russell}$, the direction of a move or a check is represented as a vector employing two variables: one defining the x component Δx, and the other defining the y component, Δy. It is easy to represent the desired equality constraint under either encoding. In DT_{flann} IOE simply includes a single identity constraint that binds Dir_{check} (the direction in the check) to the same variable as $Dir_{capture}$ (the direction of the capture). In $DT_{russell}$, IOE includes two identity constraints, one constraining Δx_{check} to equal $\Delta x_{capture}$, the other constraining Δy_{check} to equal $\Delta y_{capture}$.

Another key constraint for skewers and forks (that the capturing and checking piece be the same) is similarly represented in either encoding. In DT_{flann}, a location is encoded as a single atom Sq, so a single identity constraint is needed; in $DT_{russell}$, a location of a piece is encoded as an X, Y coordinate pair, so two identity constraints are needed.

These examples demonstrate that for DT_{flann} and $DT_{russell}$, the features definable from C_{mEBG} provide a good set of features in either case. In general, any domain theory that is capable of representing squares, piece types, and directions will provide a good set of features for use by IOE.

The second factor that allows IOE to be insensitive to the domain theory is that the specific constant and equality constraints introduced by IOE are derived from the training examples rather than from the domain theory. If all of the training examples exhibit the same constant value for a particular feature, then IOE will retain that feature in its final substitution θ. If two features are always equal to each other in the training examples, then IOE will force the two features to be equal to each other in θ. Because these regularities are independent of the domain theory, they allow IOE to succeed regardless of the way the domain theory was encoded.

In summary, unlike EBG and mEBG, IOE does not exhibit a tradeoff between correctness and ease of engineering. A further consequence of this is that IOE is more flexible than EBG and mEBG. By changing the training examples given to IOE at learning time, we can determine which constant and equality constraints are created and imposed on C_{mEBG}.

4.2 Learning Efficiency

The experiments of Section 3 demonstrated that of the three methods, IOE requires the most training examples, and therefore scores the worst on the learning efficiency criterion. This raises the critical question of how many training examples in general are required by IOE.

To answer this question, let us develop a simple mathematical model of the IOE generalization process and derive an expression that gives the learning efficiency of the algorithm.

To model IOE we make some simplifying assumptions. First, we ignore the computation of C_{mEBG} and focus only on the process of computing θ from the training instance substitutions θ_i. Second, we ignore the derivation of equality constraints and consider only the decision to replace a constant by a variable in θ. Third, we assume that all of the features

[12]Knight moves actually add another eight directions for a total of sixteen.

in C_{mEBG} are independent and take the same number of possible values.

Under these assumptions, a training instance is a simple vector of feature values. Let k be the number of features and d be the size of the domain (i.e., the number of possible values) of each feature. This gives us an instance space of size d^k. A concept is a conjunction of k features, each set to either $*$ (don't care) or a constant. This gives us a concept space of size $(d+1)^k$.

IOE will retain a constant value for a feature if all of the training examples share the same constant value for that feature. Let us call this a "coincidence". Consider learning a concept definition that contains r $*$–valued features from a set S of examples uniformly drawn from the example space (the set of all possible positive examples). One way to look at this learning problem is to think of the set S of examples as exhibiting $k-r$ intended coincidences and many unintended coincidences among the remaining r features. The goal of learning is to detect the intended coincidences and eliminate any unintended coincidences by setting each of the r features to $*$. How many examples will this require?

Consider the case when $r = 1$, then the probability that this feature is set to $*$ after m ($m \geq 2$) training examples is,

$$1 - (1/d)^{m-1}$$

since $(1/d)^{m-1}$ is the probability that the feature value observed in the first training example will remain unchanged in the subsequent $m-1$ training examples. In the worst case we have $r = k$ features. The probability that all k features have changed after m training examples is,

$$(1 - p^{m-1})^k$$

where $p = 1/d$. Let δ be the probability that after m examples we do not have a correct concept definition, then

$$1 - \delta = (1 - p^{m-1})^k.$$

Solving this expression for m gives

$$m = 1 + \log_p(1 - (1-\delta)^{\frac{1}{k}}). \tag{1}$$

This expression quantifies the learning efficiency of IOE.

To visualize this result, we graph values for m with $\delta = 0.1$ and $\delta = 0.01$ against different values for k and p in Figure 10. The theory shows that IOE scales well to larger concept definition sizes and that the main limiting factor is the parameter p. If we consider only the case where $p = 1/d$, then the worst case is with binary valued features.

This theory explains two characteristics of the empirical results for learning efficiency: (a) why IOE consistently performs worse on $DT_{russell}$ than on DT_{flann} and (b) why general-king-fork and check-with-bad-exchange require so many training examples to learn.

IOE performs worse using Russell's domain theory than it does using Flann's because the variables in a $DT_{russell}$ definition have domains that are smaller than those in a DT_{flann} definition. For example, as we have seen above, the locations of playing pieces in $DT_{russell}$ are encoded as X, Y vectors using two features each with $d = 8$ values. In DT_{flann}, on the other hand, location is encoded as a single feature having $d = 64$ values. This gives $DT_{russell}$ a larger value for p and therefore a slower learning rate.

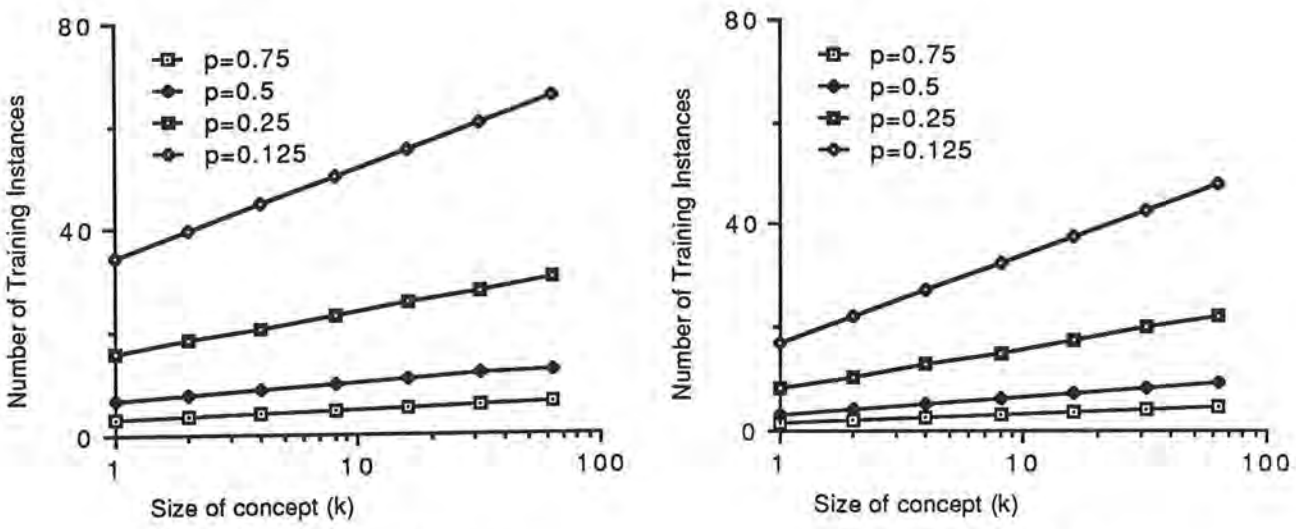


Figure 10: Learning Efficiency of IOE. The graph on the left shows the number of training examples needed to achieve 99% correctness as a function of the size of the concept description k and the probability of a feature taking different values, p. The graph on the right is similarity set up for a correctness of 90%.

The concepts general-king-fork and check-with-bad-exchange are much harder to learn than the other concepts. For example, check-with-bad-exchange requires 38 training examples to attain correctness (with probability 99%) compared to only 8 for knight-fork. To successfully learn these two concepts, IOE must generalize the "type of checking and capturing piece" to be any type. To do this, IOE must see training examples where the checking and capturing piece is a knight and training examples where it is a non-knight. A single example is not sufficient because both domain theories draw a distinction between knights and sliding pieces.

It takes IOE a long time to see the required examples because in the space of all general-king-forks, only 1 in 10 have a knight as the checking and capturing piece. This gives a p value of 0.82, which indicates that many training examples will need to be considered before this unintended coincidence is eliminated.[13]

Note that this would not be a problem if our model described probably approximately correct (PAC) learning (Valiant, 1984) rather than probably correct learning. The distribution of examples is not a problem for PAC learning, because the correctness of a concept is evaluated on the same distribution that is used for learning. In our model, a definition is considered correct only when it has completely converged. That is why general-king-fork is hard to learn, because the concept is 0% correct until both a sliding piece and a knight have been observed. In the PAC framework general-king-fork is not hard to learn, since the definition initially identified, sliding-fork, is already 90% correct. Hence, our model is more

[13]The value $p = 0.82$ is the probability that two successive training examples will give the same value for the type of the piece. The approximate probabilities of observing a knight and a sliding piece are .1 and .9 respectively. The probability of seeing two successive identical piece types is $0.1 \times 0.1 + 0.9 \times 0.9 = 0.82$

demanding than the PAC model.

In summary, the number of training examples required for IOE is not excessive and scales well with problem size. The form of the domain theory (in particular, the number of values of each feature) can influence learning speed. IOE works best when each feature has many different values and the distribution of the different values is uniform. Highly skewed distributions for features with very few values can lead to much longer learning times. However, if a probably-approximately correct concept definition is acceptable, then many fewer training examples are required.

5 Concluding Remarks

In this section we consider the implications of our results for on-going research on the problem of learning from examples. We conclude with a discussion of the problems and opportunities suggested by the IOE method.

5.1 Implications for Learning from Examples

The IOE, EBG, and mEBG methods illustrate a new approach to the problem of learning from examples. To appreciate the advantages of this new approach, let us briefly review the more traditional methods for learning from examples.

Traditional approaches (e.g., Quinlan, 1982, 1983; Michalski, 1983; Mitchell, 1982) suffer from two major problems. First, they require a large number of training examples to identify the correct concept definition. Second, they do not provide an easy way to incorporate domain knowledge into the learning process.

In the remainder of this subsection, we will argue that the three methods discussed in this paper, particularly IOE, overcome these two problems in many situations. Let us consider each problem in turn.

The number of examples required by traditional methods seems quite large when compared to the number of examples required by people to learn the same concepts. Consider the task of learning the knight-fork concept. Most people can be taught this concept with a handful of well-chosen examples. In contrast, ID3 requires 3327 examples to learn this concept with 90% accuracy.[14] What is the cause of this disparity?

Recent theoretical work (e.g., Ehrenfeucht, Haussler, Kearns & Valiant, 1988) shows that there is a fundamental tradeoff between the number of examples required for learning and the size of the space of possible concepts (the hypothesis space). More precisely, Ehrenfeucht et al. (1988) prove that any learning algorithm that considers an hypothesis space whose Vapnik-Chervonenkis dimension is d, must examine at least $\Omega(\frac{1}{\epsilon}\ln\frac{1}{\delta}+\frac{d}{\epsilon})$ training examples in order to guarantee that the hypothesis selected by the algorithm has error less than ϵ with

[14]The training examples were represented as 18 boolean features encoding the location of the knight, the king, and the queen. Each location specified a "square number" between 0 and 63 encoded in binary (6 bits). There are only 1672 positive examples of the concept. An additional 1659 negative examples were randomly generated and provided to the program. This representation is so bad, that training on 1670 positive and 1657 negative examples, ID3 is only able to predict the remaining 2 positive and 2 negative examples with probability 0.91.

probability greater than $1 - \delta$. Hence, the fact that people require many fewer examples than ID3 suggests that they are considering a much smaller hypothesis space than ID3.[15]

What determines the size of the hypothesis spaces considered by traditional inductive learning algorithms? Virtually all traditional algorithms represent their hypotheses as combinations of the features in which the training examples are represented. This is called the *single-representation trick* in Dietterich, London, Clarkson & Dromey (1982). Different algorithms can be characterized by the different ways they provide for combining the given features. ID3 combines the features in a decision tree. The AQ and version space approaches use the logical connectives of AND and OR. Perceptrons employ linear combinations of weights. Bayesian algorithms (e.g., STAGGER (Schlimmer, 1987); AutoClass (Cheeseman et. al., 1988)) employ products of probability distributions over the values of each feature.

These various combination methods each permit the learning algorithms to construct a combinatorial number of different hypotheses, and therefore they result in very large hypothesis spaces.

This suggests that the solution to the problem is to be found by considering hypothesis spaces that are not defined by combinatorial generators over the given features. This is exactly what EBG, mEBG, and IOE provide. Rather than considering all possible combinations of the given features, these methods apply the domain theory to derive a set of new features and to constrain the ways those features can be generalized. In particular, mEBG identifies a conjunction of important features, and IOE is only permitted to introduce constant and equality constraints. No new logical connectives or other combining operators are permitted. The result is that IOE can learn the knight-fork concept from two well-chosen examples (when the examples exhibit only the correct coincidences) or from 8 randomly chosen examples.

Let us now consider the second shortcoming of traditional inductive learning methods: their inability to incorporate domain knowledge easily into the learning process. For these methods, domain knowledge enters in only two ways: through the features used to represent the training examples and through the choice of feature combination methods. Neither of these ways is easy to use.

For example, when Quinlan (1983) attempted to teach ID3 the concept lost-in-2-ply for the chess endgame king-and-knight vs king-and-rook, he found that simple features describing only the type and location of each piece were inadequate. He therefore spent several months developing a set of "high-level" features that included terms such as "rook-and-king-in-same-row" and "knight-can-move-out-of-danger". With these features, ID3 succeeded in learning the concept. However, the lesson from this experience is that there is a tradeoff between the correctness of traditional methods such as ID3 and the amount of "vocabulary engineering" required to develop a set of good features. This tradeoff significantly reduces the usefulness of ID3 as a general-purpose learning method.

The other alternative for encoding domain knowledge—changing the set of combination "operators" employed by the learning algorithm—is relatively unexplored (although Seshu, Rendell and Tcheng, 1988, present some preliminary work in this area). We suspect, however,

[15]The Vapnik-Chervonenkis dimension generally increases as the size of the hypothesis space increases, although there are exceptions for hypothesis spaces with ordered (e.g., numerical) features. See Haussler (1988) for more details.

that it will be equally difficult to anticipate the relationship between combination methods and domain characteristics, and consequently, we doubt that this will provide a convenient method for incorporating domain knowledge.

The explanation-based algorithms discussed in this paper provide a more convenient and explicit method for incorporating domain knowledge. The user can construct a domain theory for a concept more general than the concepts that will be learned and then the explanation-based methods can consult this domain theory to obtain their "semantic bias". Our experiment with IOE shows that the exact form of this domain theory is not critical. However, it is true that the domain theory must be complete and correct. An important direction for future research is to find ways to exploit incomplete domain theories to provide semantic biases.

In conclusion, the explanation-based methods described in this paper directly address two major shortcomings of traditional methods for learning from examples: the need for a large number of training examples and the need for extensive "vocabulary engineering."

5.2 Open Problems and Future Research

The formulation of the theory-based concept specialization problem given in Section 1 explicitly separates the problem of learning the *correct* definition of a concept from the problem of applying that definition in some performance task. In particular, the definitions produced by EBG, mEBG, and IOE are not guaranteed to be efficiently evaluable. Some kind of knowledge-compilation process is needed to convert these definitions into efficient form.

Many systems (e.g., SOAR, Prodigy) apply some form of knowledge compilation to the results of EBG. Among the techniques employed are simplification, partial evaluation, enumeration of cases, and compaction. It turns out that in the chess domain, these techniques are not sufficient, because the learned concept still includes an embedded universal quantifier (and therefore, still requires a search to evaluate). In Flann (1988), we propose a reformulation technique that removes universal quantification by inventing new terms. We show how through reformulation, the cost of evaluating the knight-fork definition can be reduced by two orders of magnitude to approximately 2×10^3 logical inferences. In general, the successful application of explanation-based techniques to the TBCS problem will required further development of knowledge compilation methods.

It is interesting to note that the problem of applying learned concepts efficiently contributed to Quinlan's difficulty in engineering a good set of terms for lost-in-2-ply. Because ID3 uses the single representation trick, not only must it find the correct definition by combining the given features, but this definition must also be efficiently evaluable. This conjunction of correctness and efficiency constraints is difficult to satisfy, and it explains why Quinlan required so much time to design a successful vocabulary. By separating the problem of learning a correct concept from the problem of applying that concept, explanation-based methods address this aspect of the "vocabulary engineering" problem as well.

One other important direction for future research is to investigate alternative generalization strategies for IOE. The main insight of IOE is that the features identified by EBG and mEBG provide a good vocabulary in which to perform inductive generalization. There are many generalization methods besides those investigated in this paper. In particular, other constraints—besides equality and constant constraints—could be introduced (e.g., in-

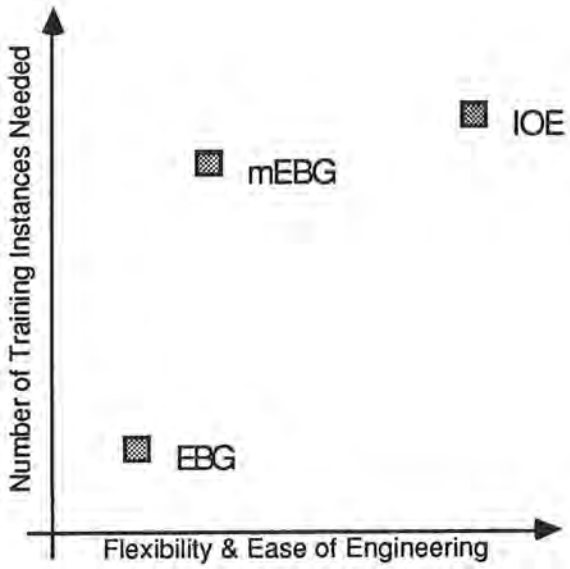


Figure 11: Trade-off among the three explanation based methods

equality constraints, climbing-generalization tree, numerical intervals, etc.). These other generalization strategies might violate the property that the learned concept is always a strict specialization of the target concept, but this is probably not important.

In conclusion, Figure 11 summarizes the results of this paper. There is a tradeoff between learning efficiency on the one hand and flexibility, ease of engineering, and correctness on the other. The method of induction over explanations represents an interesting point along this tradeoff, because it offers significantly improved correctness, flexibility, and ease of engineering while not requiring substantially more training examples than mEBG. The analysis presented in this paper suggests that there are many other points along this tradeoff curve waiting to be identified and studied.

6 Acknowledgements

Discussions with Haym Hirsh, Rich Keller and Mike Pazzani helped clarify the ideas in this paper. Caroline Koff, Giuseppe Cerbone, Jim Holloway, and Ritchey Ruff provided helpful comments on earlier drafts of this paper.

This research was partially supported by the National Science Foundation under grant numbers IST-8519926, IRI-86-57316 (Presidential Young Investigator Award), CCR-87-16748, and gifts from Tektronix and Sun Microsystems.

7 References

Bennett, J. S. and Dietterich, T. G., (1986). The test incorporation hypothesis and the weak methods. Technical Report Number 86-30-4, Department of Computer Science, Oregon State University, OR.

Braverman, M. and Russell, S., (1988). Boundaries of operationality. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 221–234). Ann Arbor, MI:

Morgan Kaufmann.

Cheeseman, P., Kelly, J., Self, M., Strutz, J., Taylor W., and Freeman D. (1988). AutoClass: A bayesian classification system. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 54–64). Ann Arbor, MI: Morgan Kaufmann.

Cohen, W., (1988). Generalizing number and learning from multiple examples in explanation based learning. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 256–269). Ann Arbor, MI: Morgan Kaufmann.

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

Dietterich, T. G. (1986). Learning at the knowledge level. *Machine Learning* 1(3) 287–316.

Dietterich, T. G., London, B., Clarkson, K., and Dromey, G., (1982) Learning and inductive inference. In P. R. Cohen and E. A. Feigenbaum (Eds.), *The handbook of artificial intelligence,* (Vol. 3). Los Altos, CA: Kaumann.

Ehrenfeucht, A., Haussler, D., Kearns, M., and Valiant, L. (1988). A general lower bound on the number of examples needed for learning. *COLT 88: The first workshop on computational learning theory* (pp. 110–120). MIT.

Flann, N. S. (1988). Improving problem solving performance by example guided reformulation of knowledge. *Proceedings of the First International Workshop in Change of Representation and Inductive Bias* (pp. 14–34). (also Technical Report Number 88-30-04, Department of Computer Science, Oregon State University, OR).

Genesereth, M. R. and Nilsson, N. J. (1987). *The logical foundations of artificial intelligence.* Los Altos, CA: Morgan Kaufmann.

Gupta, A. (1988). Significance of the explanation language in EBL. *Proceedings of the American Association of Artificial Intelligence Spring Symposium on Explanation Based Learning* (pp. 73-77). Stanford University.

Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence* 36(2) 177-222.

Hirsh, H. (1987). Explanation-based generalization in a logic-programming environment. *Proceeding of the 10th International Joint Conference on Artificial Intelligence* (pp. 221–227). Milan, Italy: Morgan Kaufmann.

Hirsh, H. (1988). Knowledge as bias. *Proceedings of the First International Workshop in Change of Representation and Inductive Bias,* (pp. 186–192).

Kedar-Cabelli, S. T. (1985). Purpose-directed analogy: a summary of current research. *Proceedings of the Third International Conference on Machine Learning* (pp. 80–83).

Kedar-Cabelli, S. T. and 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.

Laird, J. E., (1986). SOAR user's manual, Xerox Palo Alto Research Center, Palo Alto, CA.

Michalski, R. S., (1983). A theory and methodology of inductive learning. In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), *Machine learning, an artificial intelligence approach* (Vol. I). Palo Alto, CA: Tioga Press.

Minton, S. (1988). *Learning efficient search control knowledge: an explanation-based approach*, Ph.D. Thesis, Carnegie Mellon University.

Mitchell, T. (1982). Generalization as search. *Artificial Intelligence* 18(2) 203-226.

Mitchell, T., Utgoff, P. and Banerji, R. (1983). Learning by experimentation: aquiring and refining problem-solving heuristics. In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), *Machine learning: an artificial intelligence approach*, (Vol. I), Palo Alto, CA: Tioga Press.

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

Mooney, R. J., and 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.

Pazzani, M. J. (1988). *Learning causal relationships: an intergration of empirical and explanation based learning methods.* Ph.D. Thesis, University of California, Los Angeles.

Plotkin, G. D. (1970). A note on inductive generalization. *Machine Intelligence* 5 153-163.

Prieditis, A. E., (1988). Environment-guided program transformation. *Proceedings of the American Association of Artificial Intelligence Spring Symposium on Explanation Based Learning* (pp. 201-210). Stanford University.

Quinlan, J. R., (1982). Semi-autonomous acquisition of pattern-based knowledge, *Machine Intelligence,* 10.

Quinlan, J. R., (1983). Learning efficient classification procedures and their application to chess end games. In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Vol, I). Palo Alto, CA: Tioga Press.

Russell, S., (1985). The compleat guide to MRS, Knowledge Systems Laboratory, Report Number KSL-85-13, Stanford University, CA.

Schlimmer, J., (1987). Learning and representation change. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 511–515). Seattle, WA: Morgan Kaufmann.

Seshu, R., Rendell, L. and Tcheng, D. (1988). Managing constructive induction using sub-component assessment and multiple-objective optimization. *Proceedings of the First International Workshop in Change of Representation and Inductive Bias*, (pp. 293–305).

Shavlik, J. and DeJong, G. (1987). An explanation-based approach to generalizing number. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 236–238). Milan, Italy: Morgan Kaufmann.

Spiegel, R., (1975). Schaum's outline series: Theory and problems of probability and statistics, McGraw-Hill Book Company.

Utgoff, P. E. (1986). A shift in bias for inductive concept learning. In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), *Machine Learning, an Artificial Intelligence Approach* (Vol. II). Los Altos, CA: Morgan Kaufmann.

Van Harmelen, F. and Bundy, A. (1988). Explanation based generalization = partial evaluation. *Artificial Intelligence* 36(3) 401–412.

8 Appendix

In the appendix we include an annotated version of both the chess domain theories used in the empirical study. We first give the definition of target concept, check-with-bad-exchange, since both theories use the same definition. Next we give the domain theory written by Flann, then the domain theory written by Russell.

8.1 Target Concept Definition

A state `State` is an example of check-with-bad-exchange if the side to move `Side` is in check (`in_check`) and for all legal moves available, there exists a legal move for the opponent (`Otherside`) that results in a bad exchange, where a bad exchange is defined as a sequence of two moves where a valuable piece is exchanged for a piece of lower value (possibly nothing).

```
check_with_bad_exchange(State,Side,Otherside):-
    in_check(State,Side),
    forall(NewState,
           legal_move(State,Newstate1,Side),
           exists(legal_move(Newstate1,Newstate2,Otherside),
                  bad_exchange(Newstate2))).
```

Note the target concept definition uses two special literals of the form `forall`(V, P_1, P_2) and `exists`(P_3, P_4) (where P_1 is a literal, P_2, P_3 and P_4 are conjunctions of literals, and V is a list of universally quantified variables in P_1). The expression `forall`(V, P_1, P_2) is true when, for all possible solutions of P_1 (and legal bindings for V), P_2 is also true. The expression `exists`(P_3, P_4) is true when there exists a solution to P_3 such that P_4 is true.

We use a situation calculus approach to representing states and operators where the initial state is named and subsequent states are represented as operators applied to the

initial state. This is achieved by using the operator function op described in (Genesereth & Nilsson, 1987). If the input state to `legal_move` is `S`, then the new state is bound to `do(Move,S)`, where `Move` is the op function that takes four arguments: the source square, the destination square, the the piece moved, and the piece taken (may be `empty` if no piece is taken). This can be seen in the definition of `bad_exchange`. A state `State` is an example of bad exchange if the two previous moves were `Move1` followed by `Move2`, `Move1` captured a piece `Piecetaken1`, `Move2` captured a piece `Piecetaken2`, and `Piecetaken1` is a more valuable piece than `Piecetaken2`.

```
bad_exchange(State):-
    State=do(Move2,do(Move1,S)),
    Move1=op(From1,To1,Piecemoved1,Piecetaken1),
    Move2=op(From2,To2,Piecemoved2,Piecetaken2),
    type(Piecetaken1,Type1),
    type(Piecetaken2,Type2),
    morevaluable(Type1,Type2).
```

8.2 Flann's Chess Domain Theory

Each initial board state is denoted by a constant such as `state1`. The squares on the board are also denoted by constants, for example, `a8`, `b2`, and so on. Finally, the pieces are each given names such as `wr1` for the white rook and `bk1` for the black king. Empty squares are represented by an imaginary piece called `empty` (essentially a null value). With this representation, a board configuration is represented by 64 assertions. For example, the board configuration in Figure 6(a) includes the following facts:

```
square(state1,h1,wr1).
square(state1,a4,empty).
square(state1,d8,bk1).
```

In addition, the structure of the chess board is represented as the topology of the squares as follows:

```
connected(a7,a8,n).
connected(a7,b7,e).
connected(a7,b8,ne).
```

The constants `n`, `e`, `ne`, and so on represent the eight directions of the compass points. In all, 372 `connected` assertions are needed.

Additional information about each of the pieces is needed in order to define the legal moves. In particular, we identify certain pieces (e.g., `wn1`, `wr1`, and so on) as all being white pieces. Similarly, we define groups of pieces (e.g., `wn1`, `bn2`, and so on) as being of the same type, `knight`:

```
side(wn1,white).
type(wn1,knight).
side(bq1,black).
type(bq1,queen).
```

We also include the fact that `black` and `white` are opposite sides:

```
opside(black,white).
opside(white,black).
```

Using these definitions, it is possible to define legal moves for each piece. We begin by stating, for each piece, the direction and maximum number of moves it can make. For most pieces this is easy. For example, the rules for rooks are:

```
legaldirection(Side,rook,n,7).
legaldirection(Side,rook,e,7).
legaldirection(Side,rook,s,7).
legaldirection(Side,rook,w,7).
```

For knights, this simple technique does not work. To define knight moves, we first define a special kind of connectivity between squares. For example, squares `a1` and `b3` are connected by the "direction" `nne` defined by the rule

```
connected(S1,S2,nne):-
    connected(S1,Sa,n),
    connected(Sa,Sb,n),
    connected(Sb,S2,e).
```

The legal move directions for knights are then defined trivially by rules such as

```
legaldirection(Side,knight,nne,1).
legaldirection(Side,knight,nnw,1).
```

Several rules are required in order to define legal moves. First, we state that a legal move is a move such that after taking it, you are not in check:

```
legal_move(State,Newstate,Side):-
    move(State,Newstate,Side),
    not(in_check(Newstate,Side)).
```

Where `move` is defined as follows:

```
move(State,do(op(From,To,Piecem,Piecet),State),Side1):-
    opside(Side1,Side2),
    side(Piecem,Side1),
    type(Piecem,Type),
    square(State,From,Piecem),
    legaldirection(Side1,Type,Direct,Count),
    connected(From,Next,Direct),
    movedirection(State,Count,Direct,Next,To,Piecet,Type2,Side2).
```

This rule checks to see that the piece to move, `Piecem`, is located on the source square `From`; that `Piecet` is located on the destination square `To`; and that the indicated direction and number of squares is legal for the kind of piece being moved. In particular, the

movedirection predicate (given below) recursively decrements the Count as it traverses connected squares in the indicated direction. It checks that all intervening squares are empty. Notice that because knight moves are defined to have length 1, there are no intervening squares. This is how we encode the fact that knights can jump over intervening pieces.

The movedirection predicate terminates when the count is 0, with an empty square or with a take move:

```
movedirection(State,Count,Direct,Next,To,Piecet,Type2,Side2):-
    Count=0,
    !,
    fail.
movedirection(State,Count,Direct,Next,To,Piecet,Type2,Side2):-
    To=Next,
    Piecet=empty,
    Type2=empty,
    square(State,To,empty).
movedirection(State,Count,Direct,Next,To,Piecet,Type2,Side2):-
    To=Next,
    square(State,To,Piecet),
    type(Piecet,Type2),
    side(Piecet,Side2),
    !.
```

The recursive case decrements the count and checks the next square in the same direction.

```
movedirection(State,Count,Direct,Next,To,Piecet,Type2,Side2):-
    square(State,Next,empty),
    Ncount is Count - 1,
    connected(Next,NextNext,Direct),
    movedirection(State,Ncount,Direct,NextNext,To,Piecet,Type2,Side2).
```

The in_check rule is very like the move rule above. A check is defined as a possible take move of the king by the opponent.

```
in_check(State,Side1):-
    opside(Side1,Side2),
    type(Piecek,king),
    side(Piecek,Side1),
    square(State,Kingsq,Piecek),
    side(Piecetaking,Side2),
    type(Piecetaking,Typet),
    square(State,From,Piecetaking),
    legaldirection(Side2,Typet,Direct,Count),
    connected(From,Next,Direct),
    movedirection(State,Count,Direct,Next,Kingsq,Piecek,king,Side1).
```

In addition to these basic rules, frame axioms are included to indicate that pieces not explicitly moved are not affected. These are easy to write using the op notation:

```
square(do(op(F,T,Pm,Pt),S),T,Pm):-!,
    square(S,T,Pt).
square(do(op(F,T,Pm,Pt),S),F,empty):-!,
    square(S,F,Pm).
square(do(op(F,T,Pm,Pt),S),Sq,P):-
    Sq\==F,Sq\==T,square(S,Sq,P).
```

Now we have completed the description of Flann's domain theory we include the pruning information that is used by the EBG* method. The following predicates are pruned from the EBG- definition to form the EBG* definition: `connected` and `movedirection`. Pruning the `connected` predicate avoids the problem that the exact knight direction is incorporated within the definitions. Pruning the `movedirection` predicate avoids the problem with retaining the exact length of check and move threats.

8.3 Russell's Chess Domain Theory

Each board state is denoted by a constant such as `state1`. The squares on the board are represented as two coordinates, the first giving the column of the square, the second giving the row of the square. The playing piece on a square is represented as the type and side of the piece. Empty squares are represented by a piece with type and side `empty` (essentially a null value). With this representation, a board configuration is represented by 64 assertions. For example, the board configuration in Figure 6a) includes the following facts:

```
on(state1,white,rook,8,1).
on(state1,empty,empty,1,4).
on(state1,black,king,4,8).
```

Included in the domain theory is the fact that `white` and `black` are opposites:

```
opponent(white,black).
opponent(black,white).
```

In order to define the legal moves for each piece we include the directions (represented as a column vector and a row vector) in which the piece types can move:

```
movevector(rook,S, 1, 0).
movevector(rook,S, 0, 1).
movevector(rook,S,-1, 0).
movevector(rook,S, 0,-1).
movevector(knight,S, 1, 2).
movevector(knight,S, 2, 1).
```

Also included are definitions of those piece types that can move through multiple squares:

```
multipiece(bishop).
multipiece(rook).
multipiece(queen).
```

Several rules are needed to define legal moves. First we include a rule that defines the op notation for legal moves.

```
legal_move(State,Newstate,BW):-
    Newstate = do(op(s(Cf,Rf),s(Ct,Rt),p(Pm,BW),p(Pt,St)),State),
    legalmove(State,BW,Cf,Rf,Ct,Rt,Pm,Pt,St).
```

A legal move is defined in terms of the side to move BW, the from square (Cf,Rf), the to square (Ct,Rt), the type of piece moved Pm, and the type Pt of piece taken and side St of the piece taken. Three rules define three different cases of legal moves: the first rule covers the case when we are not in check and the king is not moved; the second rule covers the case where the king is moved; the third rule covers the case where the king is in check and generates moves that remove the check threat.

```
legalmove(State,BW,Cf,Rf,Ct,Rt,Pm,Pt,St):-
    not(in_check(State,BW)),
    move(State,BW,Cf,Rf,Ct,Rt,Pm,Pt,St),
    Pm\==king,
    not(discoveredcheck(State,BW,Cf,Rf,Ct,Rt)).
legalmove(State,BW,Cf,Rf,Ct,Rt,Pm,Pt,St):-
    on(State,BW,king,Cf,Rf),
    move(State,BW,Cf,Rf,Ct,Rt,Pm,Pt,St),
    opponent(BW,WB),
    not(attacking(State,WB,Ct,Rt)),
    Cv is Ct - Cf,
    Rv is Rt - Rf,
    not(multiattacksalong(State,WB,Cf,Rf,Cv,Rv)).
legalmove(State,BW,Cf,Rf,Ct,Rt,Pm,Pt,St):-
    in_check(State,BW),
    opponent(BW,WB),
    on(State,BW,king,Kcol,Krow),
    attacks(State,WB,P,Pcol,Prow,Kcol,Krow),
    escapescheck(State,Pcol,Prow,BW,Cf,Rf,Ct,Rt,Pm,Pt,St).
```

In check is defined as an attack on the king by the opponent:

```
in_check(State,BW):-
    opponent(BW,WB),
    on(State,WB,Piece,C1,R1),
    attacks(State,WB,Piece,C1,R1,C2,R2),
    on(State,BW,king,C2,R2).
```

A move escapes check if it is not a king move, does not result in a discovered check and stops the check threat to the king:

```
escapescheck(State,Pcol,Prow,BW,Col,Row,NewCol,NewRow,Pm,Pt,St):-
    move(State,BW,Col,Row,NewCol,NewRow,Pm,Pt,St),
```

```
    not(on(State,BW,king,Col,Row)),
    not(discoveredcheck(State,BW,Col,Ro NewCol,NewRow)),
    on(State,BW,king,Kcol,Krow),
    stopcheck(Pcol,Prow,NewCol,NewRow,Kcol,Krow).
```

A move stops a check if its destination square is between the check threat and the king (i.e., it blocks the check) or if its destination square is the same as the checking piece's square (i.e., it takes the checking piece):

```
stopcheck(Pcol,Prow,NewCol,NewRow,Kcol,Krow):-
    between(NewCol,NewRow,Kcol,Krow,Pcol,Prow).
stopcheck(Pcol,Prow,NewCol,NewRow,Kcol,Krow):-
    NewCol=Pcol,
    NewRow=Prow.
```

A move originates from a square occupied by a piece of the side to move and terminates in a square that can be attacked by that piece:

```
move(State,BW,C1,R1,C2,R2,Pm,Pt,St):-
    on(State,BW,Pm,C1,R1),
    attacks(State,BW,Pm,C1,R1,C2,R2),
    opponent(BW,WB),
    destinationsquare(State,WB,C2,R2,Pt,St).
```

A destination square of a move must either be occupied by a piece of the opposite side or empty:

```
destinationsquare(State,WB,C2,R2,Pt,St):-
    on(State,WB,Pt,C2,R2),
    St=WB,
    !.
destinationsquare(State,WB,C2,R2,Pt,St):-
    Pt=empty,
    St=empty,
    on(State,empty,empty,C2,R2).
```

A move results in a discovered check if the moving piece is pinned and the move is in a direction that is not parallel to the direction of the pin threat:

```
discoveredcheck(State,BW,Col,Row,NewCol,NewRow):-
    pinned(State,BW,Col,Row,Pcol,Prow),
    Cv1 is NewCol - Col,
    Rv1 is NewRow - Row,
    Cv2 is Pcol - Col,
    Rv2 is Prow - Row,
    not(parallel(Cv1,Rv1,Cv2,Rv2)).
```

A piece is pinned if it lies along a line of attack of the king by a multipiece of the opposite side:

```
pinned(State,BW,Col,Row,Pcol,Prow):-
    on(State,BW,king,Kcol,Krow),
    unitvector(Col,Row,Kcol,Krow,Cv,Rv),
    Ncol is Col + Cv,
    Nrow is Row + Rv,
    route(State,Ncol,Nrow,Kcol,Krow,Cv,Rv),
    opponent(BW,WB),
    attacksalong(State,WB,Piece,Pcol,Prow,Col,Row,Cv,Rv),
    multipiece(Piece).
```

The different ways that one piece can attack another are defined below:

```
multiattacksalong(State,WB,Col,Row,Cv,Rv):-
    attacksalong(State,WB,Piece,Pcol,Prow,Col,Row,Cv,Rv),
    multipiece(Piece).
attacks(State,BW,P,C1,R1,C2,R2):-
    attacksalong(State,BW,P,C1,R1,C2,R2,Cv,Rv).
attacking(State,BW,C2,R2):-
    attacks(State,BW,P,C1,R1,C2,R2).
attacksdirectly(State,BW,Piece,Col,Row,NewCol,NewRow,Cv,Rv):-
    on(State,BW,Piece,Col,Row),
    movevector(Piece,BW,Cv,Rv),
    nextto(Col,Row,Cv,Rv,NewCol,NewRow).
attacksalong(State,BW,Piece,Col,Row,NewCol,NewRow,Cv,Rv):-
    attacksdirectly(State,BW,Piece,Col,Row,NewCol,NewRow,Cv,Rv),
    not(multipiece(Piece)).
attacksalong(State,BW,Piece,Col,Row,NewCol,NewRow,Cv,Rv):-
    on(State,BW,Piece,Col,Row),
    multipiece(Piece),
    attacksdirectly(State,BW,Piece,Col,Row,Col2,Row2,Cv,Rv),
    route(State,Col2,Row2,NewCol,NewRow,Cv,Rv).
```

Route recursively traverses the board in a direction defined by the vector `Cv,Rv` checking that all intermediate squares are empty:

```
route(State,Col,Row,NewCol,NewRow,Cv,Rv):-
    Col=NewCol,
    Row=NewRow.
route(State,Col,Row,NewCol,NewRow,Cv,Rv):-
    on(State,empty,empty,Col,Row),
    nextto(Col,Row,Cv,Rv,Col2,Row2),
    route(State,Col2,Row2,NewCol,NewRow,Cv,Rv).
```

The following rules compute vector arithmetic needed by the previous rules:

```
nextto(Col,Row,Cv,Rv,NewCol,NewRow):-
    NewCol is Col + Cv,
```

```
    NewCol > 0,
    NewCol < 9,
    NewRow is Row + Rv,
    NewRow > 0,
    NewRow < 9.
parallel(Cv1,Rv1,Cv2,Rv2):-
    0 is Cv1 * Rv2 - Cv2 * Rv1.
unitvector(Col1,Row1,Col2,Row2,Cv,Rv):-
    Mcv is Col2 - Col1,
    Mrv is Row2 - Row1,
    sign(Mcv,Cv),
    sign(Mrv,Rv).
sign(Mv,V):-
    Mv > 0,
    V = 1.
sign(Mv,V):-
    Mv < 0,
    V = -1.
sign(Mv,V):-
    Mv = 0,
    V = 0.
between(Xc,Xr,Yc,Yr,Zc,Zr):-
    Cv1 is Zc - Xc,
    Rv1 is Zr - Xr,
    Cv2 is Yc - Xc,
    Rv2 is Yr - Xr,
    parallel(Cv1,Rv1,Cv2,Rv2),
    Dot is Cv1 * Cv2 + Rv1 * Rv2,
    Dot < 0.
```

In addition to these basic rules, frame axioms are included to indicate that squares not involved in moves remain unchanged:

```
on(State,Sm,Tm,Ct,Rt):-
    State=do(op(s(Cf,Rf),s(Ct,Rt),p(Tm,Sm),p(Tt,St)),NS),
    on(NS,St,Tt,Ct,Rt).
on(State,S,T,Cf,Rf):-
    State=do(op(s(Cf,Rf),s(Ct,Rt),p(Tm,Sm),Pt),NS),
    S=empty,
    T=empty,
    on(NS,Sm,Tm,Cf,Rf).
on(State,S,T,C,R):-
    State=do(op(s(Cf,Rf),s(Ct,Rt),Pm,Pt),NS),
    s(C,R)=s(Cf,Rf),
    s(C,R)=s(Ct,Rt),
    on(NS,S,T,C,R).
```

Now we have completed the description of Russell's domain theory we include the pruning information that is used by the EBG* method. The following predicate is pruned from the EBG- definition to form the EBG* definition: `attacks`. Pruning the `attacks` predicate avoids the two problems with the EBG- definition: retaining the piece type (either sliding or not sliding) and retaining the exact length of check and move threats.