Induction Over Explanations: A Method that Exploits Domain Knowledge to Learn from Examples

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)

April 4, 1988

Abstract

We introduce five criteria by which to judge the suitability of a method for solving the problem of learning concepts from examples: correctness (the correct concept should be identified), performance efficiency (the learned definition should be efficient to apply to the performance task), flexibility (the method should be able to learn a variety of different concepts), ease of engineering (the method should be easy to implement in new domains) and learning efficiency (the method should learn from few examples efficiently). We analyze two existing methods for learning from examples, similarity-based learning (SBL) and explanation-based learning (EBL), and find them inappropriate for solving an important sub-problem: learning *functional* concepts from examples. In SBL, the performance efficiency goal is incompatible with the other goals, because the representation best for performance is ineffective for learning. In EBL, it is difficult to satisfy the flexibility or correctness goals, because the concepts are identified from a single example and an inflexible generalization policy. We introduce a new method, called induction over explanations (IOE), that overcomes these difficulties. The method applies a domain theory to construct explanations from the training examples as in EBL, but forms the concept definition by employing an SBL generalization policy over the explanations. The concept definition is then compiled into a form efficient for the performance task. The method has the advantage that an explicit domain theory can be exploited to aid the learning process, the vocabulary engineering of representations is significantly reduced, and the correct concepts can be learned from few examples. We illustrate the method in an implemented system, called *Wyl2*, that learns concepts in a variety of domains including the concepts "skewer" and "knight-fork" in chess.

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

1 Introduction

This paper discusses methods for solving the problem of learning from examples. The problem is defined as follows:

Given:

- **Training examples:** A set of n training examples of the form
  $\langle TI_i, C, + \rangle$ positive examples,
  $\langle TI_j, C, - \rangle$ negative examples,
  with $1 \leq i, j \leq n$. Each TI is a description written in the environment language (EL).

Determine:

- **Intended Concept:** A performance program, P, that correctly labels new EL descriptions as being instances or non-instances of the concept C.

In this definition there are two minor departures from traditional statements of this problem. First, the goal of the learning system is to produce a performance program rather than a concept definition. Second, the representation used for the training examples is called the Environment Language (EL) to distinguish it from other representations that will be introduced below.

A learning method that solves this problem should meet the following five criteria:

Correctness: The method should identify the correct concept—that is, the one intended by the teacher or true in the environment.

Performance Efficiency: The method should construct a performance program that classifies future instances efficiently.

Flexibility: The method should be concept-independent—that is, it should be able to learn a wide range of possible concepts without internal modification. It should be possible to teach a different concept simply by changing the examples presented to the method.

Learning Efficiency: The method should require few training examples and small amounts of computational resources.

Ease of Engineering: The method should be easy to implement in new domains. It should not require delicate "vocabulary engineering" in order to meet the four previous criteria.

In this paper, these five criteria are applied to evaluate two general classes of learning methods: so-called "similarity-based learning (SBL)" methods (e.g., Michalski, 1980; Quinlan, 1982) and "explanation-based learning (EBL)" methods (e.g., Mitchell, Keller, & Kedar-Cabelli, 1986; DeJong & Mooney, 1985).

This evaluation focuses on the problem of learning *functional* concepts from *structural* examples. A concept definition is said to be functional if it is stated in terms of goals and behaviors. A training example is structural if it is stated in terms of instantaneous properties of objects or relationships among objects. For example, the concept of "knight fork" in chess is easily stated as a functional concept ("The knight simultaneously checks the king and the queen, the king is forced to move out of check, then the knight takes the queen"). The concept of an "arch" in the blocks world can be stated as a structural concept ("One block is on top of two upright bricks that do not touch").

The evaluation determines that neither learning method, when applied to learning functional concepts, meets all five criteria. SBL methods satisfy the performance efficiency and flexibility criteria, but they rarely identify the correct concept, they need too many training examples, and they require extensive vocabulary engineering. EBL methods are more efficient both during learning and during performance. However, they also rarely identify the correct concept, they are generally inflexible, and they also require significant vocabulary engineering.

To combine the strengths and avoid the weaknesses of these two methods, we have developed a new method, called Induction Over Explanations (IOE). Like EBL, it employs a domain theory to construct explanations for each training example. These multiple explanations are then generalized using SBL methods to develop a generalized explanation that constitutes the concept definition. This generalized explanation is then compiled into an efficient recognition procedure using EBL-like methods.

This paper is organized as follows. Section 2 gives some examples of the problem of learning functional concepts from structural training examples. Section 3 then evaluates the suitability of SBL and EBL approaches for solving this learning problem. Section 4 presents the IOE method and describes the *Wyl2* program, an implementation of the method. Section 5 demonstrates, by empirical experiments with the *Wyl2* program, that it satisfies all five evaluation criteria. Section 6 concludes by discussing a number of issues.

This paper assumes that the reader is already familiar with SBL and EBL methods. For good examples of SBL methods, see Dietterich [1982]. For a good review of EBL methods, see Mitchell et al [1986], Kedar-Cabelli [1987], and Hirsh [1987].

2 Some Example Learning Problems

Before discussing the SBL and EBL methods, let us consider some instances of the kinds of learning problems we are trying to solve.

Figure 1 shows an example of the chess concept "knight-fork." The white knight on square e6 is simultaneously threatening the black queen 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 any of c8, d7 and e8). This will permit the knight to take the queen.

We desire a learning system that can learn the knight-fork concept from examples such as this one when they are presented in a simple, structural representation. Such a representation would describe the position in terms of

- the types and locations of the pieces,

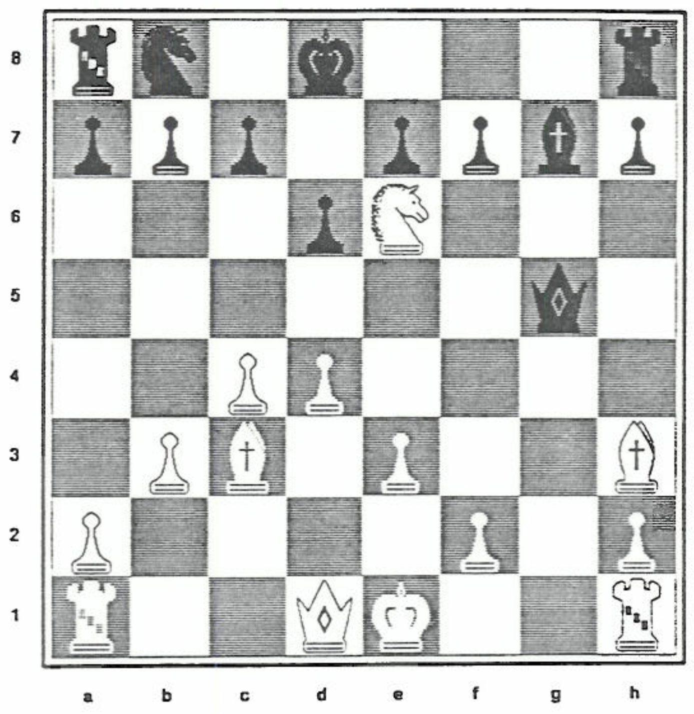


Figure 1: Knight-Fork: black to play, $TS2_1$ (`state1`)

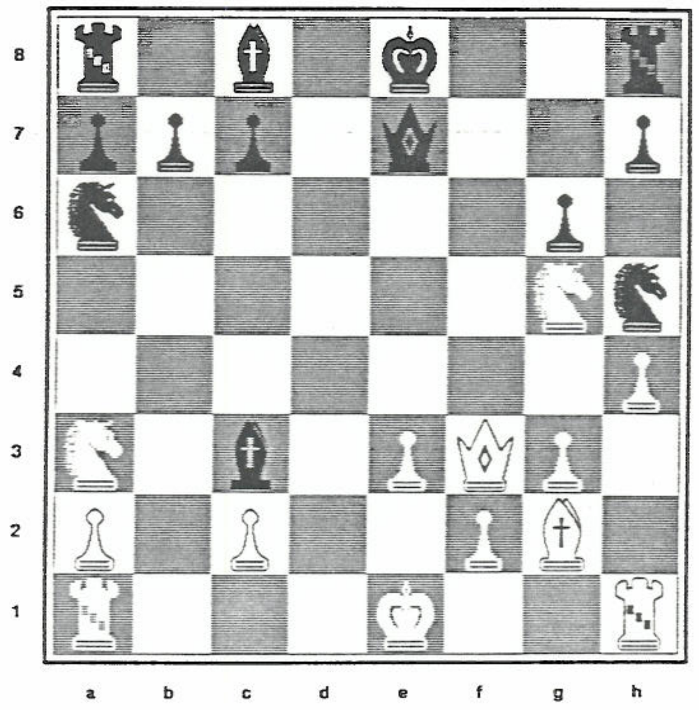


Figure 2: Bishop-Fork: white to play, $TS1_2$

- the name of the concept, in this case "knight-fork".

The knight fork concept is most easily defined functionally, in terms of the goals of the players and the available moves. It is difficult to describe in structural terms exactly what configurations of pieces constitute a knight-fork—particularly, because one must exclude situations in which the knight can be taken by some opposing piece.

In addition to learning knight-fork, we want the learning system to be able to learn other functional concepts from the same domain. Figure 2 shows a structural example for another concept: bishop-fork. In this case, the black bishop on `c3` simultaneously threatens the white king on `e1` and the white rook on `a1`. To respond, the white side must move the king out of check to any of `d1,d2,e2,` or `f1`. The black bishop will then take the rook on `a1`.

Both Figures 1 and 2 are also examples of the functional concept of "fork against the king" in which one piece simultaneously threatens the opponent king and some other piece without itself being threatened. Suppose we present both of these examples to the learning system at once. In that case, we would like it to learn a definition for "fork against the king." Furthermore, if we presented additional examples of forks not involving the king, we would like the learning system to construct a definition for "general-fork," in which one piece simultaneously threatens any two opponent pieces without itself being threatened.

It should be clear from these examples what is meant by the "flexibility" criterion given in Section 1. We want to be able to change the set of examples given to the learning system and have the resulting concept change as well.

3 Applying SBL and EBL to Learn Functional Concepts

Now that we have seen the kind of learning problems to be solved, let us consider applying the existing SBL and EBL methods to them.

3.1 Similarity-Based Learning

All methods for learning from examples accept a set of training examples and produce a concept description consistent with those examples. Similarity-based learning methods, construed narrowly, are those methods in which unseen examples are classified as positive (or negative, respectively) if they are syntactically similar (different) in specified ways to the positive training examples. The concept description specifies the ways in which the unseen examples must resemble the positive training examples. Typically, this is accomplished by representing the concept as a collection of generalized instances (or some equivalent data structure) in which some aspects have been omitted. ID3, for example, represents the concept as a decision tree that tests only the relevant features. Slightly more sophisticated methods (e.g., AQ11, version space algorithm) can replace some aspects of the training examples with terms of intermediate generality (e.g., the integers 2 and 4 might be replaced by "even integer" in the concept description). Internal disjunctions (e.g., replacing 2 by "2 or 4") are also supported in the AQ family of systems.

The algorithms employed by these learning systems work by comparing the positive examples to each other and retaining the similarities and by comparing the positive examples to negative examples and retaining the differences. The point to emphasize is that these comparisons take place in the environment language or in some language closely related to it (e.g., by internal disjunction or simple generalization trees for features). Hence, similarities and differences that cannot be easily expressed in the *EL* will not be discovered by these methods.

When applied to learning functional concepts, SBL methods are unlikely to find the correct concept definition. This is because the similarities underlying training examples of such concepts as knight fork have very little to do with structural properties of the chess position (e.g., number of pieces, the particular locations of the pieces, etc.).

To illustrate this point, let us consider a straightforward, epistemologically adequate structural representation for chess positions. Each board state can be 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 can be represented by 64 assertions. For example, the board configuration in Figure 1 includes the following facts:

```
square(state1,h1,wr1).
square(state1,d5,empty).
```

In addition, the structure of the chess board must be represented. A basic representation that captures the topology of the squares is the following:

```
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.

What kinds of similarities can be captured in this representation? We can easily describe all chess boards containing known pieces at unknown locations (e.g., all boards with the black king, `square(State,Loc,bk1)`. We can describe unknown pieces at known locations (e.g., `square(State,c4,Piece)`. We can describe a collection of four adjacent squares that lie in a row, column, or diagonal: `connected(Sq1,Sq2,Dir)`, `connected(Sq2,Sq3,Dir)`, `connected(Sq3,Sq4,Dir)`. However, it is quite difficult to represent the set of chess boards in which "For every legal move available to the king to escape a knight check, there is a legal response in which the knight captures the queen." It is not a matter of expressiveness: *any* collection of chess board positions can be expressed in the structural vocabulary. The problem is that extremely lengthy disjunctions may be needed. The heavy use of disjunction reveals that the structural vocabulary is not capturing much similarity or commonality among the various knight-fork positions.

When SBL methods are given training examples in this structural form, it is very unlikely that they will find the correct definition. Instead, they will tend to find simpler, accidental structural similarities among the examples that have nothing to do with the desired functional concept.

One approach to solving this problem is to change the environment language in which training examples are presented. In Quinlan's [1983] work, for example, he was attempting to teach the ID3 system a functional concept ("lost-in-2-ply") using structural examples. He found it necessary to introduce higher level structural and functional terms such as "rook-and-king-in-same-row" and "knight-can-move-out-of-danger" in order to get the system to discover the correct concept. Three man-weeks of effort were put into developing these specific vocabulary terms (and several more months of work were expended developing previous vocabularies for related learning problems). We call this process "vocabulary engineering," because it involves the careful design of a vocabulary such that the learning program will succeed.

In spite of this vocabulary engineering, ID3 still required many training examples—at least 83 examples were required to learn the correct concept. Furthermore, when Quinlan then tackled the related concept of "lost-in-3-ply," it was found that this vocabulary was inadequate. Two months of additional vocabulary engineering were needed to develop a suitable representation for learning this new concept. As Quinlan says,

> "The work on lost n-ply relations supports the view that almost all the effort (for a non chess-player at least) must be devoted to finding attributes that are adequate for the classification problem being tackled." [1982, pp 477-478].

Let us now evaluate SBL methods according to our five criteria:

Correctness: Without vocabulary engineering, it is unlikely that the correct concept definition will be learned. With vocabulary engineering, correctness is guaranteed by design.

Performance Efficiency: Because the concept definition is represented in structural terms, it can be efficiently evaluated to classify unseen examples. Furthermore, even when Quinlan included some functional features in his representation, the time required to classify unseen examples was less than for a comparable table-lookup algorithm (Quinlan, 1982) and far less than for a game-tree search.

Flexibility: Without vocabulary engineering, SBL methods are very flexible. The flexibility derives from the fact that SBL methods implement a preference relation (also called a "bias") over the space of possible concept definitions. They tend to prefer the syntactically simplest concept description available. ID3, for example, will not add a new test to the decision tree unless it is required to discriminate between positive and negative examples. By presenting additional training examples, the teacher can force ID3 to discard one decision tree and construct a different, more complex, one.

With vocabulary engineering, as Quinlan discovered, some flexibility is lost, because the *EL* was designed to make it easy to learn one particular concept.

Learning Efficiency: The computation time required by most SBL methods is very minimal. ID3, for example, can process thousands of training examples per minute. However, the number of training examples required is rather large, especially when one considers that many fewer examples are needed to teach a person the same concepts.

Ease of Engineering: When learning functional concepts, time-consuming vocabulary engineering is needed. Hence, these methods are not easy to engineer.

3.2 Explanation-Based Learning

Explanation-based learning methods are given, in addition to the training examples expressed in the environment language, a domain theory (expressed in a theory language, TL) and a target concept (TC). The TL is a superset of the EL that also includes the TC. The domain theory includes axioms that connect training examples in the EL to the target concept, TC. Using these axioms, the learning system can determine whether the TC is true for any given training example. If the TC is true of an example, the example and the axioms can be applied to prove this fact.

EBL methods learn from positive examples. Recall that each positive example has the form $\langle TI_i, +, C\rangle$. The learning process proceeds in three steps. First, a proof (called an explanation) is constructed that the target concept TC is true for TI_i. This proof can be viewed as a tree with an instance of the TC at the root and facts from TI_i and the domain theory at the leaves. Next, this proof is generalized as much as possible without altering the structure of the proof. This can be accomplished by removing from the proof tree those leaf nodes that contain facts from the TI_i[1]. This has two effects. First, some of the bindings to variables appearing in the proof are removed, because they are no longer required to match the particular facts given in TI_i. Second, the deletion of the leaf nodes creates several dangling subgoals DG_i in the tree. These subgoals can be collected into a conjunction of statements that, when combined with the domain theory, logically imply TC. Therefore, the formula

$$DG_1 \wedge DG_2 \wedge \ldots \wedge DG_k \supset TC$$

is a theorem in the domain theory.

There are two possible ways to carry out the third step. If the purpose of the learning process is to "operationalize" the target concept (i.e., to compile it for efficient recognition), then the conjunction $DG_1 \wedge DG_2 \wedge \ldots \wedge DG_k$ can serve as an efficient recognizer for a special case of TC. It is efficient because each condition tests only EL features. In such cases, the concept name C given in the training examples is ignored (or, equivalently, is assumed to be identical to TC).

The other case arises then the concept C is assumed to be distinct from (and more specific than) TC. In that case, the EBL method assumes that the conjunction of the DG_i is a necessary and sufficient definition for the concept C:

$$DG_1 \wedge DG_2 \wedge \ldots \wedge DG_k \equiv C.$$

The left-hand side of this rule constitutes a complete, efficient procedure for determining whether an example is an instance of C.

As an example of this second case, suppose we apply the EBL method to the knight-fork example from Figure 1. This example will demonstrate the basic ideas of the method as well as showing that EBL cannot easily learn the knight-fork concept.

[1] In PROLOG-EBG (Kedar-Cabelli and McCarty, 1987) this is accomplished by not binding the training instance facts to the leaf subgoals)

To apply EBL, we must first give a domain theory for chess. The domain theory is represented using a modified version of Prolog that supports embedded existential and universal quantification as well as the usual Horn clauses and control mechanisms (i.e., cut and negation-as-failure). This domain theory will be described in three parts. First, some simple elaborations to the representation for the chess board are given. These include the types and sides of the pieces. Second, the legal moves for the various pieces are defined. Third, the rules defining the target concept are given. (Readers may wish to skim these details upon first reading).

The environment language for chess boards introduced above simply gave unique names to the pieces. While this is adequate in principle, additional information about each piece is needed in order to define the legal moves and the target concept. 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 must 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,8).
legaldirection(Side,rook,e,8).
legaldirection(Side,rook,s,8).
legaldirection(Side,rook,w,8).
```

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:

```
legalmove(State,Newstate,Side):-
    move(State,Newstate,Side),
    not(incheck(Newstate,Side)).
```

Where `move` is defined as follows:

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

A move is described as an operator function `op` in the situation calculus (using techniques recommended in Genesereth & Nilsson, 1987). The function `op` takes four arguments: the source square, the destination square, the name of the piece moved, and the name of the piece taken (`empty` if no piece is taken). This rule checks to see that the indicated player, `Playerm`, is located on the source square; that `Playert` is located on the destination square; and that the indicated direction and number of squares is legal for the kind of piece being moved. In particular, the `movedirection` predicate 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.

Now we are in a position to define the target concept. Since we are focusing on tactical concepts during the chess middle game, the target concept, TC, is "favorable exchange." A favorable exchange is one in which a more valuable piece was captured in exchange for a piece of lower value (or possibly in exchange for the `empty` pseudo-piece, which has the lowest value of all). The favorable exchange goal is defined by two rules: a base case and a recursive case. Here is the base case:

```
goodgoal(exchange(Type1,Type2),State,Side1,Side2):-
    State=do(Move2,do(Move1,S)),
    Move1=op(From1,To1,Playermoved1,Playertaken1),
    Move2=op(From2,To2,Playermoved2,Playertaken2),
    type(Playertaken1,Type1),
    type(Playertaken2,Type2),
    morevaluable(Type1,Type2).
```

This rule says that if the current state was obtained by performing `Move1` followed by `Move2` and if `Move1` involved our side taking piece `Playertaken1` and `Move2` involved our opponent taking piece `Playertaken2` and if `Playertaken1` was more valuable than `Playertaken2` then this was a good exchange.

Here is the recursive case:

```
goodgoal(exchange(Type1,Type2),State,Side1,Side2):-
    exists(legalmove(State,Newstate,Side1),
           badgoal(exchange(Type2,Type1),Newstate,Side2,Side1)).
```

This says that an exchange is good for Side1 if there exists a move available to Side1 such that in the resulting state, a bad exchange has occurred for Side2. This implements the maximizing part of the min-max search for two-person games. The rules defining badgoal implement the minimizing part of the search. Again there are two rules, a base case and a recursive case:

```
badgoal(exchange(Type1,Type2),State,Side1,Side2):-
    State=do(Move2,do(Move1,S)),
    Move1=op(From1,To1,Playermoved1,Playertaken1),
    Move2=op(From2,To2,Playermoved2,Playertaken2),
    type(Playertaken1,Type1),
    type(Playertaken2,Type2),
    morevaluable(Type2,Type1).
badgoal(exchange(Type1,Type2),State,Side1,Side2):-
    forall([Newstate],
           legalmove(State,Newstate,Side1),
           goodgoal(exchange(Type2,Type1),Newstate,Side2,Side1)).
```

Notice that the recursive rule says that an exchange is unfavorable for Side1 if every move available to Side1 results in a state where a favorable exchange has occurred for Side2.

The above goodgoal and badgoal rules use 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.

In addition to these basic rules, frame axioms must be 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),Sq,P):-
    Sq\==F,Sq\==T,square(S,Sq,P).
```

This completes the description of the domain theory.

Now let us return to the question of how explanation-based learning will process the example of knight-fork shown in Figure 1. In all knight fork positions the side to move is in check and cannot avoid an unfavorable exchange. Hence, we use the conjunction of incheck and badgoal as the target concept for this training example.

Figure 3 shows part of the explanation produced by the first two steps of the EBL method applied to the example in Figure 1. The left subtree is the generalized proof that the king is in check from a piece on square J while the right subtree includes the proof of a move from square V that can take a piece on square Z. Notice that all of the specific constants from the training example (such the type of checking piece knight) have been replaced by variables.

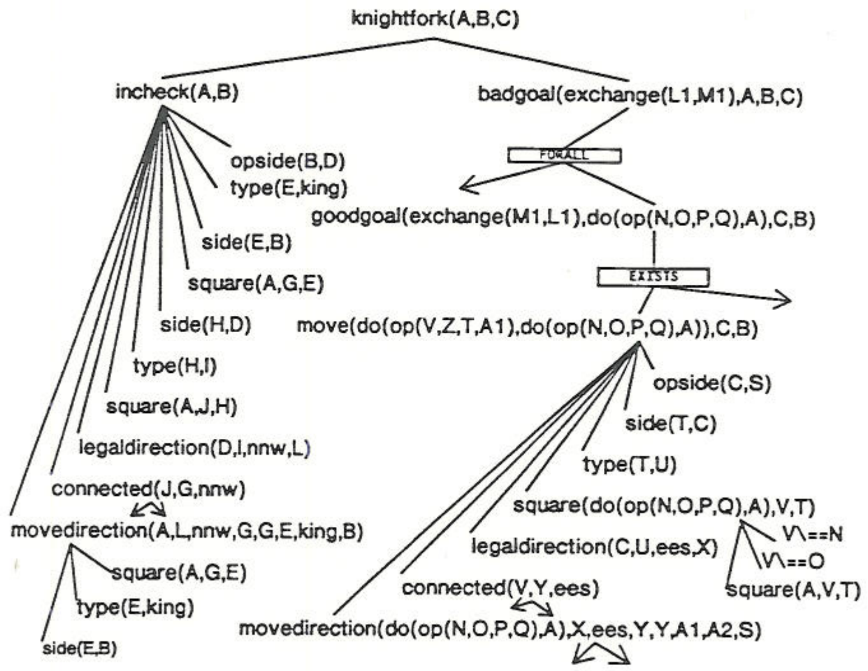

knightfork(A,B,C)
incheck(A,B)
badgoal(exchange(L1,M1),A,B,C)
opside(B,D)
type(E,king)
FORALL
goodgoal(exchange(M1,L1),do(op(N,O,P,Q),A),C,B)
side(E,B)
EXISTS
square(A,G,E)
side(H,D)
move(do(op(V,Z,T,A1),do(op(N,O,P,Q),A)),C,B)
type(H,I)
opside(C,S)
square(A,J,H)
side(T,C)
legaldirection(D,I,nnw,L)
type(T,U)
connected(J,G,nnw)
square(do(op(N,O,P,Q),A),V,T)
movedirection(A,L,nnw,G,G,E,king,B)
V\==N
legaldirection(C,U,ees,X)
V\==O
square(A,G,E)
connected(V,Y,ees)
square(A,V,T)
type(E,king)
movedirection(do(op(N,O,P,Q),A),X,ees,Y,Y,A1,A2,S)
side(E,B)

Figure 3: EBL generalized explanation generated from $TS2_1$

When we apply the third stage of the EBL process to this generalized proof the resulting concept definition will be incorrect. First, because constants from the domain theory are retained in the proof, the resulting rule will only cover cases where the checking and taking moves are in *exactly* the same directions as the example. This is because the legal knight directions are defined as domain theory rules with constants. Another problem is with the recursive `movedirection` rule. Here, because the example included only single moves (i.e., with no intervening squares), the resulting rule will only cover cases that involve single moves. This problem is more apparent if we had used the bishop-fork example given in Figure 2. Here, the resulting rule would only cover cases where both the checking and capturing moves pass through exactly one empty intervening square. This is known as the *generalization-to-n* problem.

There is a simple solution to both of these problems in this case. We simply terminate the explanation *construction* whenever the two troublesome predicates (`connected` and `movedirection`) are encountered.

Here is the definition of knight-fork that is obtained from applying the final EBL step to the modified explanation:

```
knight-fork(A,B,C):-
    opside(B,D),type(E,king),side(E,B),square(A,G,E),
    side(H,D),type(H,I),square(A,J,H),
    legaldirection(D,I,K,L),connected(J,F,K),
    movedirection(A,L,K,F,G,E,king,B),
    forall([M],
legalmove(A,M,B),
(M=do(op(N,O,P,Q),R),opside(C,S),
side(T,C),type(T,U),square(R,V,T),
V\==N,V\==O,
legaldirection(C,U,W,X),connected(V,Y,W),
movedirection(do(op(N,O,P,Q),R),X,W,Y,Z,A1,A2,S),
not(incheck(do(op(V,Z,T,A1),do(op(N,O,P,Q),R)),C)),
type(Q,L1),type(A1,M1),
morevaluable(M1,L1))).
```

Unfortunately, this rule is still incorrect—it is far too general. In fact, in one sense it is even more general than the "general fork" concept described in Section 2 because it does not necessarily define a "fork". The capturing piece need not be the same as the checking piece. It says that a knight fork is any board position (A), with B to play, in which:

- A piece H on square J threatens the king (on square G).
- For all moves (by side B, piece P taking piece Q) there is an opponent's move by piece T on square V that can capture a piece A1.
- The piece A1 is more valuable than Q.

Let us call this concept "check with forced exchange."

There are two ways in which this definition is overly general. First, important constants appearing in the training example, such as knight, are not retained in the concept definition. Second, identity constraints have been lost (such as the fact that the *same* piece should threaten the king as well as threaten and take the opponent piece).

This can be seen by considering how the correct definition of knight fork could be expressed by making a set of substitutions in the "check with forced exchange" rule above. The substitutions below overcome the first problem by constraining some of the variables in the general rule to be constants:

```
{I/knight,L/1,
 U/knight,X/1,
 M1/queen
 L1/empty,Q/empty,}
```

The I,L,U,X substitutions constrain the type of checking piece and capturing piece to be knight. The M1 substitutions constrain the type of piece taken by the knight to be queen. The final substitution constrains the type of piece taken by the king to be empty (i.e., the opponent exchanges the queen for nothing).

The substitutions below overcome the second problem by constraining some variables to be the same.

```
{G/N,E/P
 H/T,J/V}
```

The `G` and `E` substitution ensures that the only legal moves available to the losing side are to move the king out of check, while the `H` and `J` substitutions ensure that the checking and capturing piece are the same.

Since we can correct the rule by some simple substitutions the problem does not lie with the original domain theory—it is perfectly capable of representing the knight-fork concept. The problem lies in the overly aggressive generalization strategy of EBL. Too many of the constants and identity constraints appearing in the proof structure were replaced by distinct variables during the EBL generalization step.

Another way to understand this point is to consider what would happen if we attempted to teach this same EBL system the concept of bishop fork by presenting the example from Figure 2. The EBL system will output exactly the same concept description, because the same explanation structure is constructed and generalized in the same way.

From these examples, we can see that EBL will not discover the correct concept in these cases. Furthermore, it is very inflexible. Changing the training examples cannot lead to any change in the learned concept definition.

As with similarity-based learning, it is possible to engineer the representations so that EBL will learn the correct concept. The trick is to change the domain theory to employ more restricted rules. Notice that during the generalization process, any constants appearing in the *domain theory rules* are not removed or generalized (only constants appearing in the training example are removed). Therefore, if we incorporate important constants into the domain theory, we can prevent overgeneralization.

For example, one way to alter the domain theory so the piece types will be retained in the explanations (and hence the concept definitions) is to "promote" the type of piece moved (`Type`) into the head of the `move` rule on page 11 (as the last argument) and add six new rules to the domain theory, one for each piece type:

```
move(State,Nextstate,Side):-
    move(State,Nextstate,Side,pawn).
move(State,Nextstate,Side):-
    move(State,Nextstate,Side,queen).
move(State,Nextstate,Side):-
    move(State,Nextstate,Side,king).  ...
```

It is much more difficult to engineer the domain theory to ensure that identity constraints are retained. The only way in which the same variable can occur in different places in an EBL generated rule is when the domain theory rules used in the proof constrain the two variables to be the same. Consider the problem of engineering the domain theory to ensure that the checking piece (variable `H`) and the capturing piece (variable `T`) are the same. First the checking piece variable must be "promoted" into the head of the `incheck` rule. Second, the moving piece in the `move` rule must be "promoted" through `move`, `legalmove` and both the `badgoal` and `goodgoal` rules. Finally, the target concept, a conjunction of the altered `incheck` and `badgoal` rule heads, must be so defined to constrain the checking square variable (from `incheck`) and the capturing square variable (from `badgoal`) to have the same value. This last step in altering the target concept is the same process that was employed in PROLOG-EBG to generate the concept "suicide" from "kill" by setting both variables of "kill" to the same variable [Kedar-Cabelli and McCarty, 1987].

Let us now evaluate EBL according to our criteria. We consider only the case in which the concept C is assumed to be a specialization of TC. In the other case, the learning system already knows the desired concept $C \equiv TC$, so it is correct by definition; it is efficient both during learning and during performance; it requires little vocabulary engineering; but it is completely inflexible, since it is only capable of learning a single concept: TC. Here is the evaluation of EBL in the more interesting case:

Correctness: EBL will only learn the correct concept C if it corresponds to a maximally general explanation structure. Without vocabulary engineering, this is not satisfied for the concepts we are interested in. With engineering, it will only be satisfied for some concepts and not others. For example, all the concepts we can learn will either: ignore the piece types (when using the original domain theory) or incorporate the piece types (when using the modified theory). There is no middle ground.

Performance Efficiency: A strong point of EBL is that it provides a way of translating a functional concept definition into an efficient recognition procedure.

Flexibility: With or without vocabulary engineering, EBL systems are very inflexible. If two training examples for two different concepts, C_1 and C_2, share the same explanation structure, then EBL will assume that C_1 and C_2 are equivalent.

Learning Efficiency: The principal computational cost of EBL is the first step of constructing an explanation. In some domain theories this can be intractible (see Mitchell, et al. 1986). The cost of the generalization step is very low. EBL methods learn from a single example, so they are very efficient by that measure.

Ease of Engineering: Vocabulary engineering of the domain theory can be very difficult or even impossible, especially if more than one concept is to be learned using the same theory.

3.3 Summary

Table 1 summarizes the evaluation of SBL and EBL methods for learning functional concepts from examples. We can see that without vocabulary engineering, neither method achieves high correctness. On the other hand, both methods produce efficient classification procedures. The main tradeoff concerns flexibility and learning efficiency. Unengineered SBL is very flexible, but requires large numbers of training examples. EBL is very inflexible, but requires only one training example. The goal of our research has been to develop a method that combines the strong points of both of these methods and meets all five criteria.

4 Induction Over Explanations

4.1 Statement of the Method

The method that we have developed is called Induction Over Explanations (IOE). Here is a description of the inputs and outputs of the method:

Criterion	SBL		EBL
	Structural CDL	Engineered CDL	
Correctness	low	high	low
Performance efficiency	high	high	high
Flexibility	high	low	low
Learning efficiency	low	low	high
Ease of engineering	high	low	low

Table 1: Evaluation of Methods for Learning Functional Concepts

Given:

- **Training examples:** A set of n *positive* training examples of the form $\langle TI_i, C, + \rangle$ where each TI_i is a description written in an EL.
- **Domain Theory:** A set of axioms that define a target concept TC and connect it to the EL. The axioms must permit the method to prove that each training example is an instance of TC. Furthermore, the intended concept C must be a specialization of TC, that is, $C \supset TC$.

Determine:

- **The Intended Concept:** An efficient procedure for recognizing instances of the intended concept C.

The inputs and outputs are identical to those for the EBL method except that IOE can accept several training examples rather than only one. As with EBL, the intended concept C must be a specialization of the target concept TC.

The IOE method proceeds by the following three steps:

1. **Explain:** For each training example, TI_i, construct a proof (i.e., explanation) showing that it is an instance of the target concept TC.
2. **Generalize:** Apply syntactic generalization methods to these proofs to construct a single generalized proof that represents the maximally specific common generalization of the input proofs. The generalized proof is formed by a combination of a simple constants to variable bias that is employed over the syntactic structure of the explanations and the pruning of dissimilar explanation sub-trees among the instances. Like EBL, the generalized proof is still a valid proof of the target concept.
3. **Compile:** The leaves of the generalized proof tree are extracted and simplified to produce an efficient recognition procedure for the desired concept C.

Of these three steps, only the second is a significant departure from EBL. Instead of the aggressive EBL policy of generalizing the proof as much as possible without altering its structure, IOE employs the more conservative policy of generalizing the proof only as much as is needed to develop a common proof.

We claim that this method satisfies all five evaluation criteria. In the rest of this section, we briefly sketch the arguments for each criterion. Subsequent sections provide an empirical demonstration of these claims.

Correctness: The discussion of the EBL method showed that the domain theory language provided a concise way of describing functional concepts. Unfortunately, the EBL generalization step is unable to select those TL descriptions intended by the teacher. IOE, because it is more conservative in its generalization step, is able to produce the desired TL descriptions.

Performance Efficiency: Because the same EBL compilation methods are applied in step 3 of IOE, the resulting recognition predicate is efficient. In fact, because IOE can learn more specific rules than EBL, more efficient recognition predicates can be identified.

Flexibility: By employing an SBL-style generalization technique in step 2, IOE inherits the flexibility of SBL methods. Different sets of training examples will cause IOE to replace different sets of constants by variables in the generalized proof. This allows it to learn a variety of different concepts.

Learning Efficiency: As with EBL, the first step of constructing the explanations is the most expensive. The remaining steps are very efficient.

Ease of Engineering: The sections below demonstrate that IOE attains its correctness and flexibility without requiring any vocabulary engineering. The domain theory described in Section 3.2 is employed without modification. Training examples are described in the simple structural language given in Section 3.1. Hence, IOE is easy to engineer when compared to SBL and EBL.

4.2 *Wyl2*: An Implementation of IOE

In this section we describe an implemented system called *Wyl2* (after James Wyllie, checker champion of the world from 1847 to 1878) that applies the IOE method. *Wyl2* is implemented in Prolog as a meta interpreter that takes as input a set of Prolog facts and rules, and outputs a new Prolog rule. It is intended to be domain independent in the same sense that two previous EBL systems (Kedar-Cabelli and McCarty 1987, Hirsh 1987) are domain independent: the inputs and outputs of the system are written in a logical programming language and not in a vocabulary-specific representational scheme.

To present *Wyl2*, we will show it learning the concept of knight-fork using two examples: the original position shown in Figure 1 and an additional position shown in Figure 4.

A learning problem is specified to *Wyl2* by asserting into Prolog the domain theory and the training examples (each described by 64 `square` facts). Then, a small set of assertions such as

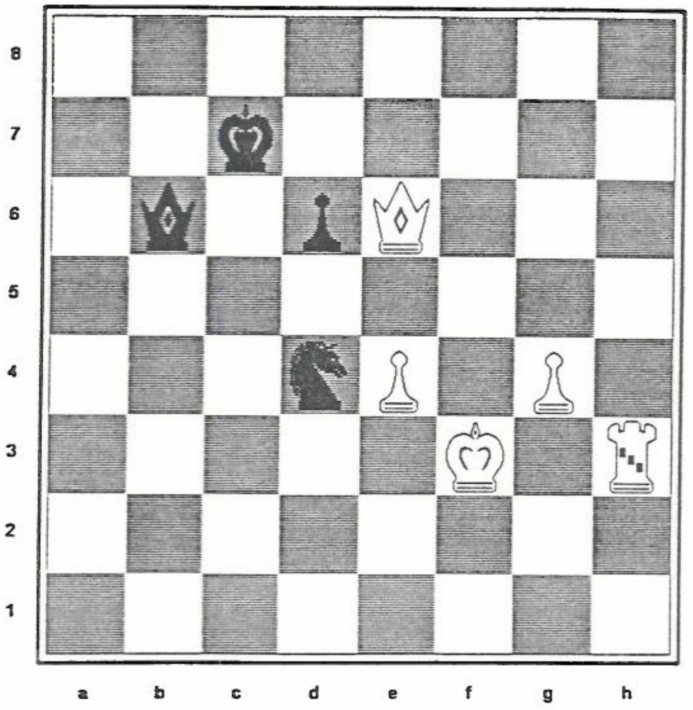


Figure 4: Knight-Fork, white to play, $TS1_1$ (`state2`)

```
knightfork(state1,black,white).
knightfork(state2,white,black).
```

are given, using a new predicate `knightfork`. Finally, an assertion is made that these knight forks are instances of the target concept. For the knight fork example, this assertion looks like

```
targetconcept(knightfork(State,Winner,Loser),
              (incheck(State,Side),
               badgoal(exchange(queen,empty),State,Winner,Loser))).
```

This says that knight fork is an instance of the target concept "positions in which the player is in check and will lose his queen in exchange for nothing."

Wyl2 operates by performing the three steps of the IOE method.

4.3 Constructing the Explanations

The input to the explanation generator is the domain theory, training instance and the target concept definition `targetconcept(`C, TC`)`. The output is `targetconcept(`Ct, TCp`)`, where Ct is C instantiated from the training example and TCp is a fully ground explanation tree with an instantiation of TC at the root.

Constructing TCp is straightforward. There are two points to note. First, because the language includes universal quantification (in the form of `forall(`$V, P1, P2$`)`), the explanation includes all possible solutions to $P1$, the corresponding bindings of V and proof trees

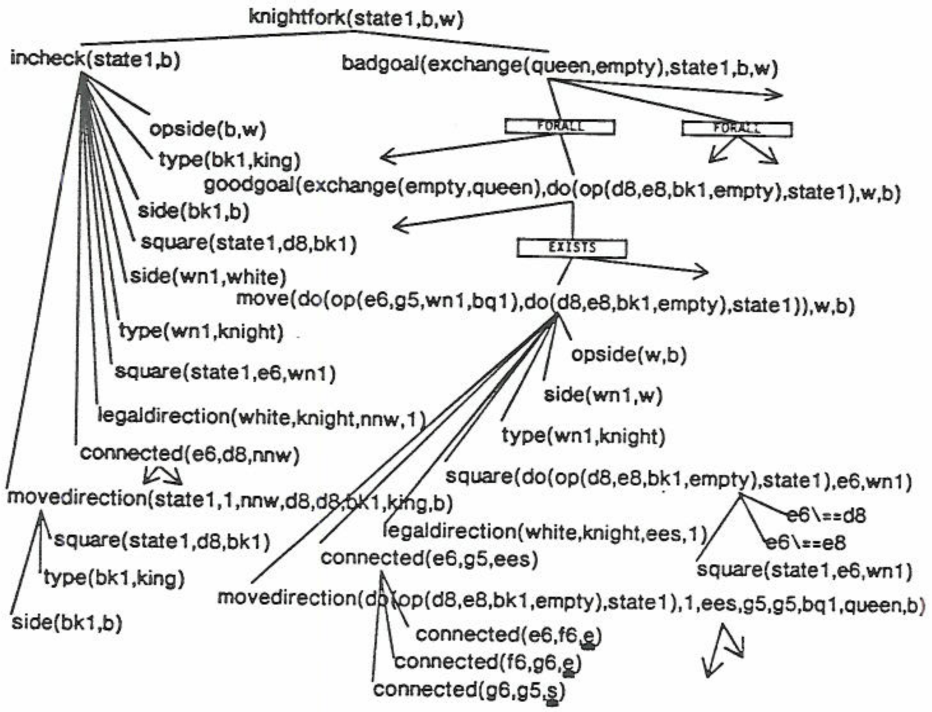

knightfork(state1,b,w)
incheck(state1,b)
badgoal(exchange(queen,empty),state1,b,w)
opside(b,w)
FORALL
FORALL
type(bk1,king)
goodgoal(exchange(empty,queen),do(op(d8,e8,bk1,empty),state1),w,b)
side(bk1,b)
square(state1,d8,bk1)
EXISTS
side(wn1,white)
move(do(op(e6,g5,wn1,bq1),do(d8,e8,bk1,empty),state1)),w,b)
type(wn1,knight)
opside(w,b)
square(state1,e6,wn1)
side(wn1,w)
legaldirection(white,knight,nnw,1)
type(wn1,knight)
connected(e6,d8,nnw)
square(do(op(d8,e8,bk1,empty),state1),e6,wn1)
movedirection(state1,1,nnw,d8,d8,bk1,king,b)
e6\==d8
square(state1,d8,bk1)
legaldirection(white,knight,ees,1)
e6\==e8
connected(e6,g5,ees)
square(state1,e6,wn1)
type(bk1,king)
movedirection(do(op(d8,e8,bk1,empty),state1),1,ees,g5,g5,bq1,queen,b)
side(bk1,b)
connected(e6,f6,e)
connected(f6,g6,e)
connected(g6,g5,s)

Figure 5: Explanation generated from $TS2_1$, black to play

for *P2*. Second, during explanation construction, *Wyl2* keeps track of which constants arose from the training examples and which constants arose from the domain theory rules. This information is needed during the generalization step. Using a technique similar to Prolog-EBG, *Wyl2* constructs the proof tree and returns it.

Figure 5 shows the explanation tree produced by *Wyl2* when analyzing the board position illustrated in Figure 1. Constants arising from the domain theory rules are underlined. The top node is the target concept (the conjunction of the `incheck` and `badgoal` predicates). The left subtree proves that the black king on square `d8` is currently in check because there is a knight in square `e6` that can move in the direction `nnw` to take the king. The right subtree proves that the bad exchange is unavoidable. Included in the Figure 5 is the branch in which the black king moves to `e8` and the knight then takes the queen on square g5 by moving in direction `ees` . Notice the minimax structure of this right subtree. The situation is a `badgoal` if for all possible black moves, the resulting situation is a `goodgoal` for the opponent. This is true if there exists a white move such that the resulting situation is a `badgoal` for black. Figure 6 shows a similar explanation for the second training example shown in Figure 4.

4.4 The Generalizer

The input to the generalizer is a list of explanations in the form `targetconcept`(Ct_i, TCp_i). The output is of the form of a generalized explanation, `targetconcept`(Cg, TCg) where $Cg = Gen([Ct_1, Ct_2, \ldots, Ct_n])$ and $TCg = Gen([TCp_1, TCp_2, \ldots, TCp_n])$. $Gen(X)$ takes as

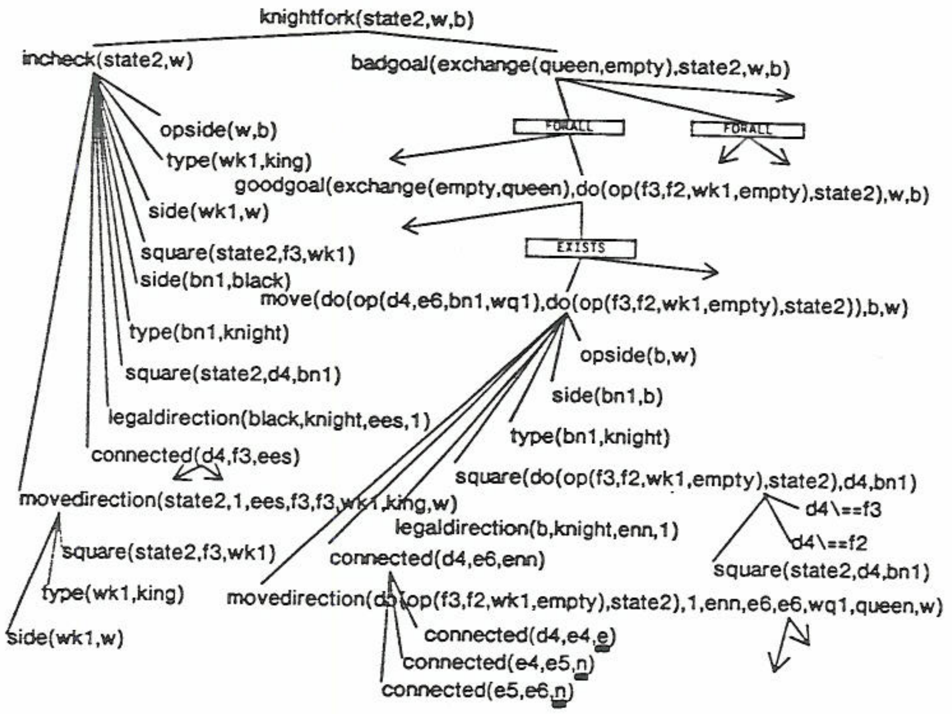

knightfork(state2,w,b)
incheck(state2,w)
badgoal(exchange(queen,empty),state2,w,b)
opside(w,b)
FORALL
FORALL
type(wk1,king)
goodgoal(exchange(empty,queen),do(op(f3,f2,wk1,empty),state2),w,b)
side(wk1,w)
square(state2,f3,wk1)
EXISTS
side(bn1,black)
move(do(op(d4,e6,bn1,wq1),do(op(f3,f2,wk1,empty),state2)),b,w)
type(bn1,knight)
opside(b,w)
square(state2,d4,bn1)
side(bn1,b)
legaldirection(black,knight,ees,1)
type(bn1,knight)
connected(d4,f3,ees)
square(do(op(f3,f2,wk1,empty),state2),d4,bn1)
movedirection(state2,1,ees,f3,f3,wk1,king,w)
d4\==f3
legaldirection(b,knight,enn,1)
d4\==f2
square(state2,f3,wk1)
connected(d4,e6,enn)
square(state2,d4,bn1)
type(wk1,king)
movedirection(do(op(f3,f2,wk1,empty),state2),1,enn,e6,e6,wq1,queen,w)
side(wk1,w)
connected(d4,e4,e)
connected(e4,e5,n)
connected(e5,e6,n)

Figure 6: Explanation generated from $TS1_1$, white to play

input a list of ground terms and outputs a single term that is the maximally specific common generalization of the inputs that is still a valid proof.

Figure 7 shows the result of applying *Gen* to the two explanations show in Figure 5 and Figure 6 (variables are prefixed by capital letters following the prolog convention). Notice that many constants have been changed to variables. For example, because one example showed white-to-move and the other showed black-to-move, the specific sides involved have been replaced by variables `B` and `C`. Similarly, the specific squares involved have been replaced by the variables `G` for the knight square, `E` for the king square, and `K` for the queen square. The specific directions of each move have also been generalized.

Another thing to note is that the same variables appear in more than one place. The variable `F` represents both the piece that checks the king and the piece that takes the queen while `G` represents both the originating square of the check threat and the originating square of the queen capture.

Also note that some constants have not been changed to variables. In particular, the constraints that `F` must be a `knight`, `D` a `king`, and `L` a `queen` are all retained in the generalized explanation.

The generalizing function *Gen* applies the following two generalization policies (biases):

Maximally-specific conjunctive generalization: The generalized explanation must be expressible as a logical conjunction with nested quantifiers as needed. The effect of this bias is two-fold. First, it rules out the option of retaining the two separate explanations and simply joining them with "or". Instead, a single, general explanation must be found that covers both of the specific explanation trees. Second, it requires that all

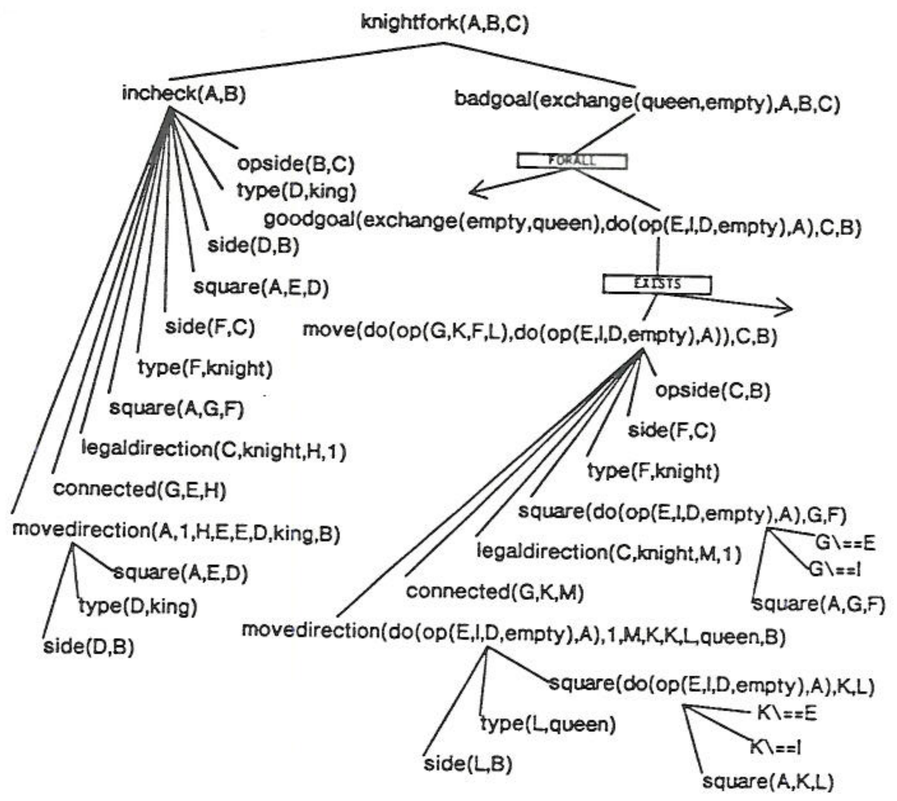

knightfork(A,B,C)
incheck(A,B)
badgoal(exchange(queen,empty),A,B,C)
opside(B,C)
FORALL
type(D,king)
goodgoal(exchange(empty,queen),do(op(E,I,D,empty),A),C,B)
side(D,B)
square(A,E,D)
EXISTS
side(F,C)
move(do(op(G,K,F,L),do(op(E,I,D,empty),A)),C,B)
type(F,knight)
opside(C,B)
square(A,G,F)
side(F,C)
legaldirection(C,knight,H,1)
type(F,knight)
connected(G,E,H)
square(do(op(E,I,D,empty),A),G,F)
movedirection(A,1,H,E,E,D,king,B)
G\==E
legaldirection(C,knight,M,1)
G\==I
square(A,E,D)
connected(G,K,M)
square(A,G,F)
type(D,king)
movedirection(do(op(E,I,D,empty),A),1,M,K,K,L,queen,B)
side(D,B)
square(do(op(E,I,D,empty),A),K,L)
K\==E
type(L,queen)
K\==I
side(L,B)
square(A,K,L)

Figure 7: Generalized explanation generated from $TS2_1$ and $TS1_1$

branches under a universal quantifier in the explanations must be merged to form a single, universally-quantified conjunction.

To implement this bias, *Wyl2* applies *Gen* to the list of explanations. *Gen* recursively descends the explanation trees starting at the root, merging the explanations to form single general explanation. If *Gen* encounters universal quantification, the multiple branches of each explanation are first merged to form single, universally-quantified conjunctions. *Gen* is then applied to the resulting list of universally-quantified conjunctions giving a general conjunction that describes all of the branches found in the examples.

Gen merges a list of explanations by applying the following rule:

$$Gen([a_1, a_2, \ldots, a_n]) = A.$$

There are four simple cases that determine A from $a_1, a_2, \ldots a_n$. First, if $a_1 = a_2 = \ldots a_n$ then $A = a_1$. All examples share the same constant and thus the definition should include that constant. Second, if all a_i are constants from the training instances and are not all identical, then A is set to a variable. In this case the examples are different so the definition includes a variable. Third, if a_i include constants from the domain theory rules that are not all identical, then special processing is needed that is described below. The final case is when the a_i are composite terms, in this case A is the result of applying *Gen* recursively to each of the sub terms.

This bias identifies constants to be retained in the final definition. In Figure 7 the gen-

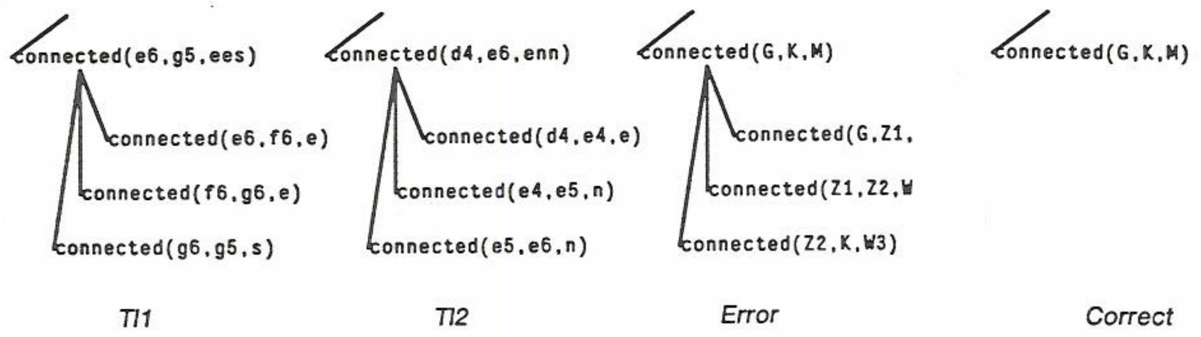


Figure 8: Example of pruning dissimilar explanation trees

eralized explanation includes the constraint that the checking piece must be a `knight` (left subtree) because this was a constant that both examples shared (i.e., *Gen* case 1 applies). This bias also identifies irrelevant constants such as the threatening square or the particular direction of movement (i.e., *Gen* case 2 applies).

No coincidences: This bias retains the identity constraints in the generalized explanation that occur in the explanations. This bias states that if the same constant appears at different points in the same explanation these points are *necessarily* equal. The chess concept knight-fork demonstrates this bias. In the explanation given in Figure 5 the originating square of the check (`e6`) and the originating square of the capture are the same. This bias says the two occurrences of `e6` are equal. In the explanation given in Figure 6 the originating square of the check (`d4`) and the originating square of the capture are the same. Again the bias says these are equal. When these explanations are merged together during generalization, `e6` and `d4` will match in two places, as the originating square of the check and the originating square of the capture. This bias says the two lists [`e6`,`d4`] are equal and hence are assigned the *same* variable. This is implemented by making the second base case of *Gen* above, from constants to variables $Gen([a_1, a_2, \dots a_n]) = A$, a one to one function. If ever the same list of constants appears as an argument to *Gen*, it is assigned the *same* variable as before.

We have described the basic operation of the generalizer. We now turn to the special case when $[a_1, a_2, \dots, a_n]$ includes constants from the domain theory rules that are not all identical. Here, the simple constants to variables bias cannot be applied because it could lead to an explanation that is not a valid proof of the target concept. To see why, consider the result of applying the simple generalization rule between the three `connected` facts that make up the proofs of `connected(e6,g5,ees)` in Figure 5 and `connected(d4,e6,enn)` in Figure 6 reproduced in Figure 8. Simple variablization (shown in Figure 8) leaves the constraint that the two squares `G` and `K` in `connected(G,K,M)`, must be connected by *any* two intermediate squares. This violates the target concept because it violates the domain theory rules (i.e., legal knight moves).

To avoid this problem, the correct generalization is to prune the explanations until a common parent is found that can be generalized by applying the simple biases. In this example the common parent is the top `connected` clause illustrated in Figure 7 and Figure 8.

This problem with dissimilar explanation trees can come up when the explanations to be generalized involve different predicate names. We treat this case in exactly the same way as we treat dissimilar domain theory constants because they are equivalent. A domain theory with constants can be rewritten into an equivalent domain theory without constants, by *currying* the constants into the predicate names. Both of these domain theories, when used for learning, will exhibit problems with dissimilar explanation trees in exactly the same situations. With the first domain theory the problems will arise from different constants. With the second domain theory the problem will arise from different predicate names. Hence, by treating the two domain theories the same, the generalization policy is insensitive to either formalism.

By pruning dissimilar explanation trees, the IOE method overcomes one of the problems with the EBL generalization policy: incorrectly retaining some domain theory constants. Recall that when learning knight-fork from $TS2_1$ in Figure 1, an EBL generalizer retains the exact direction of movement for the knight. The responsibility for solving this problem in EBL lies with the system designer who must ensure that the explanation is truncated at the top `connected` predicate. In contrast, with IOE, the responsibility lies with the teacher to choose examples that involve different knight move directions.

Another problem with EBL that can be avoided in IOE through this policy of pruning dissimilar explanation trees is the generalization-to-n problem. Recall that when an EBL generalizer was applied to learn from the chess examples, the rules formed made incorrect restrictions on the *length* of the moves involved. When learning from Figure 1 the rule incorrectly restricted both the capturing and checking moves to pass through no intervening squares (i.e. to be of length 1) and when learning from Figure 2 both moves where restricted to move through exactly one intervening empty square (i.e., to be of length 2). If *Wyl2* were given both of these examples, the desired concept would be the one where the length of moves is unrestricted (but within the domain theory). This is achieved by slightly changing the pruning policy when the dissimilar explanations involve recursive invocations of the same rule. Rather than pruning at the "lowest" suitable common parent, we prune at the first invocation of the recursive rule. The generalized explanation formed by learning from Figure 1 and Figure 2 will include only the generalized `movedirection` literal.

4.5 Compiling the Generalized Explanation

The compiler takes the generalized explanation `targetconcept(`Cg`,`TCg`)` and produces a Prolog rule Cg`:-`Bc where Bc is both logically equivalent to TCg and efficient to execute. The compiler forms Bc by extracting the leaves of TCg while maintaining the universal quantification present in the explanation.

The final rule generated by *Wyl2* from the generalized explanation in Figure 7 is given in Figure 9. This rule expresses the following definition for knight-fork: A state `A` with side `B` to play is an example of knight-fork if the following holds:

- A `king` of side `B` exists on square `E` in state `A` (line 2). A `knight` of side `C` exists on

```
knightfork(A,B,C):-
1 opside(B,C),
2 type(D,king), side(D,B), square(A,E,D),
3 side(F,C), type(F,knight),square(A,G,F),
4 legaldirection(C,knight,H,1), connected(G,E,H),
5 opside(C,B),G\==E,
6 forall([Z],
7        legalmove(A,Z,B),
8        (Z=do(op(E,I,D,empty),A),
9         G\==I,
10        legaldirection(C,knight,M,1), connected(G,K,M),
11        K\==E, K\==I,
12        square(A,K,L), type(L,queen), side(L,B),
13        not(incheck(do(op(G,K,F,L),do(op(E,I,D,empty),A)),C)))).
```

Figure 9: Final Prolog rule generated from $TS1_1$ and $TS2_1$ (numbers are for reference)

square `G` in state `A` (line 3), where `C` is the opposite side of `B` (line 1). Square `G` is connected to a square `E` in direction `H`, where `H` is a legal direction for the knight (line 4). (i.e., the king is in check from a knight).

- All legal moves possible for side `B` from state `A` involve moving the king from square `E` to an empty square `I` (lines 6-8).
- For all legal moves, the `knight` on square `G` is connected to square `K` in direction `M`, where direction `M`, is legal for a `knight` (line 10). Square `K` is occupied by the queen of side `B` (line 12). Following the king move and knight capture move the capturing side is not in check (line 13). (i.e., the knight can legally take the queen).

It is easy to see how the compiler works when one compares the rule in Figure 9 to the generalized explanation in Figure 7. The first four lines of the conjunction in the rule come from simply collecting the leaves of the left subtree and removing repeated literals. The code in lines 9 to 13 was built similarly by collecting the leaves of the right subtree. In general, the compiler generates a conjunction by collecting the leaves while traversing the generalized explanation in the same order in which it was built.

To compile the exists construct in the explanation is straightforward. The result of compiling `exists`$(P3, P4)$ is simply the conjunction formed by appending the result of compiling $P3$ to the result of compiling $P4$.

Compiling the `forall` construct in the explanation is more complex. The complexity stems from the important restriction that the resulting rule must still be a valid proof of the original target concept. In our example, the target concept specifies a knight-fork position is one were the side to move cannot avoid the bad exchange of a queen for nothing. This constraint was guaranteed during explanation construction by the domain theory rule for the `badgoal` goal. This rule states that a position is a `badgoal` if *forall* moves the resulting position is a goodgoal for the opponent. Hence, to ensure the rule generated by the compiler is still a valid proof of the target concept, the rule must also include universal quantification over all legal moves. In place of the general goodgoal goal from the domain theory, we include in the rule a specialization: the one identified by the generalized explanation of goodgoal, were the queen is captured by the knight. The goodgoal is further specialized by restricting all the legal moves to be of the same form as those moves that occurred in the examples. This is illustrated in the knight-fork rule, Figure 9 line 8. The rule restricts all legal moves to move the king into an empty square because this was the common form of the moves found in the examples.

In general, if the training instance explanations for a concept include a forall from the domain theory rules of the form `forall(`$V, P1, P2$`)`, the generalized explanation will include a generalization of solutions to $P1$, referred to as $P1g$ and a generalized proof tree of $P2$, referred to as $P2p$. The compiler forms a new term, `forall(`$V', P1', P2'$`)`, where V', $P1'$ and $P2'$ are defined as follows: V' is a list of new universal variables, $P1'$ is the result of substituting a new universal variable from V' in $P1g$ for each position where $P1$ has a universal variable, $P2'$ is formed by appending the previous substitutions stated as equalities to the result of compiling $P2p$. In the knight-fork example, V' is the new variable list `[Z]`, while $P1g$ is `legalmove(A,do(op(E,I,D,empty),A),B)`. The compiler substitutes `Z` for `do(op(E,I,D,empty),A)` in $P1g$ to form $P1'$ and appends `Z=do(op(E,I,D,empty),A)` to the result of compiling the left subtree of the generalized explanation.

The compiler includes a set of general transformations that implement a simple partial evaluator. One transformation rule removes repeated literals from a conjunction. One rule maps a clause P to either P, if it cannot be evaluated, or `true` or `false` if it can be evaluated. Other transformations remove `true` from conjunctions and replace conjunctions that contain `false` with `false`.

The compiler also includes a simple scope analyzer that tries to move each literal to its highest scope. This rule is responsible for the literals in line 5 in Figure 9. The literal `opside(C,B)` was part of the capture move proof but because it is entirely bound at a higher scope it is moved out of the scope of the universal quantification. The capture move rule also included a conjunction of the three literal given on line 3. These were moved up and removed by the partial evaluator because they already exist in the higher conjunction.

The rule could be further simplified if a more powerful partial evaluation where available. For example, the inequalities could be removed if the system could apply knowledge of functional dependencies and prove that `E`, (the king's square), `G` (the knight's square) and `K` (the queen's square) can never be equal. The test for the `queen` at square `K` (line 12) could be moved out of the scope of the universal, if the system could prove that the existentials `J`, `K` and `L` are all independent of `Z`. The final test for non-check (line 13) is time consuming and could be simply ignored. This would trade accuracy for efficiency.

5 Evaluation of *Wyl2* according to our Criteria

In this section we empirically evaluate *Wyl2* according to the criteria developed earlier in the paper through a simple test.

In this test the domain theory is fixed while the training instances are varied in a systematic way. We demonstrate *Wyl2* learning a variety of concepts in chess that are all specializations of the concept introduced earlier: "check with forced capture." Here are some typical examples:

1. Knight fork of king and queen that captures the queen.
2. General fork of king and queen that captures the queen.
3. General fork of king and another piece that captures the piece.
4. Rook fork of king and another piece that captures the piece.
5. Bishop fork of king and rook that captures the rook.
6. General skewer of king that captures some piece.
7. Rook skewer of king that captures the queen.

We have already seen how the fork concept 1 is a specialization of "check with forced capture." In general, the fork concepts specialize the general concept by constraining the checking piece and the capturing piece to be the same. A skewer concept is a further specialization of the fork concept. In a skewer, a piece threatens the king, forcing it to move out of check and reveal a piece that is captured. A skewer, therefore, further constrains a fork definition so that the direction of check threat and the direction of capture are the same.

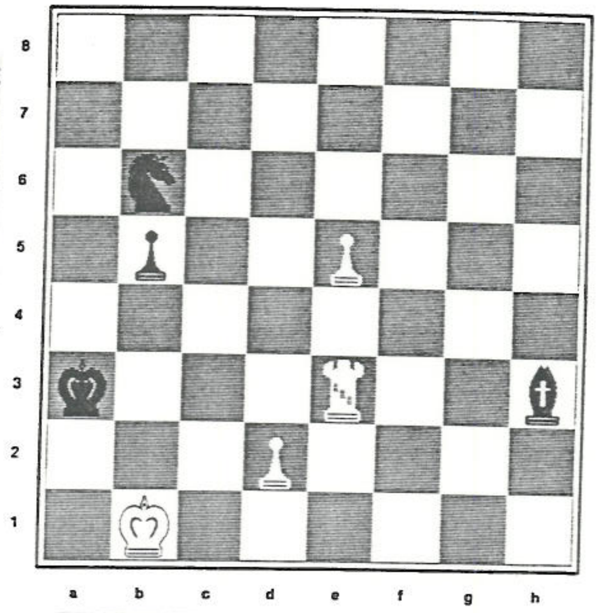
To learn these concepts *Wyl2* is given a set of 7 training instances—all examples of some form of the concept "check with forced capture." The training instances are divided into two sets, three black-to-play positions (called set $TS2$) and four white-to-play positions (called $TS1$). *Wyl2* learns a concept by applying the IOE method from every possible pair-wise combination, $\{TS1_i, TS2_j\}$ where $TS1_i \in TS1$ and $TS2_j \in TS2$. For comparison, we also employ a simple EBL system to learn a concept definition from the 7 examples. The resulting concept definitions learned by *Wyl2* and the EBL system are then compared and partially ordered according to generality and used to classify a test set of further instances.

This simple test allows an evaluation of the learning goals introduced earlier. Learning efficiency is evaluated by both monitoring the resources used in learning and observing the range of concepts that can be learned from just two examples. Performance efficiency is evaluated by monitoring the resources used when employing the concept definitions in classification tasks. Both correctness and flexibility is evaluated by studying the concept space searched that is revealed by these tests.

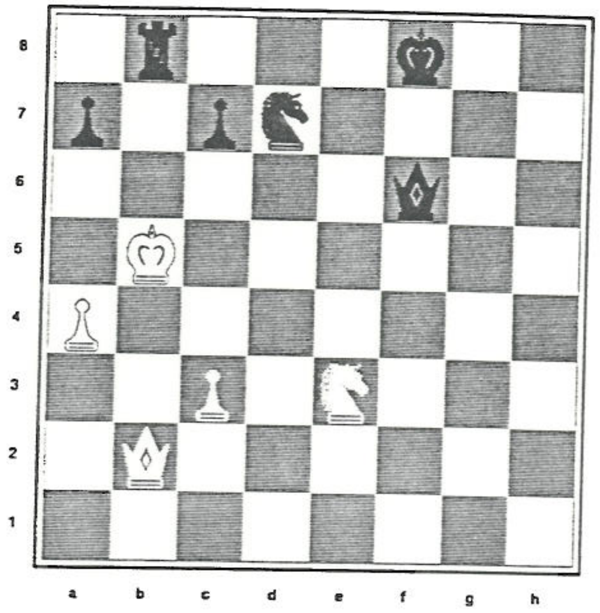
The remaining criteria—ease of engineering—cannot be evaluated thoroughly without exploring the effect systematically changing the domain theory has on the other criterion. This is an area of active and future research.

5.1 Test of *Wyl2* in Chess

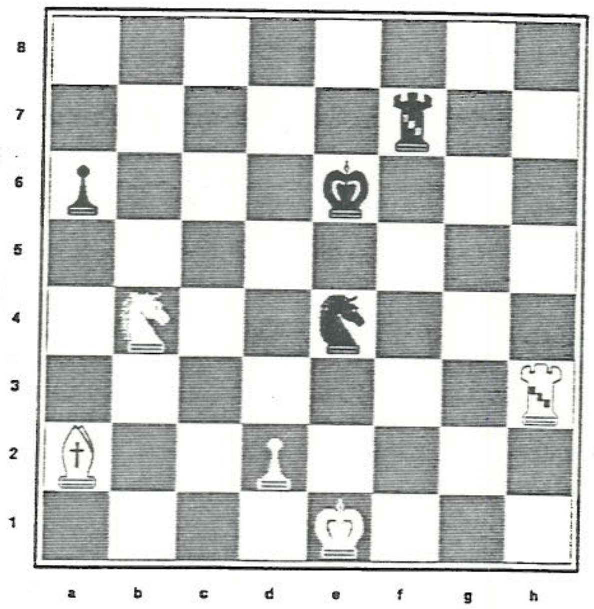
The domain theory used in this test is the one for chess introduced earlier. The sets of training instances include 4 new chess positions in addition to $TS2_1$ in Figure 1, $TS1_1$ in Figure 4 and $TS1_2$ in Figure 2. The chess positions of each new training instance are illustrated in Figure 10. Notice that in each position the side to move is in check and cannot avoid the loss of a piece in exactly two moves. In $TS2_2$, with black to play, the king is checked by the rook on `e3` who captures the bishop on `h3`. This is an example of a rook fork of the king that captures a bishop. $TS2_3$ illustrates a skewer with black to play; the bishop



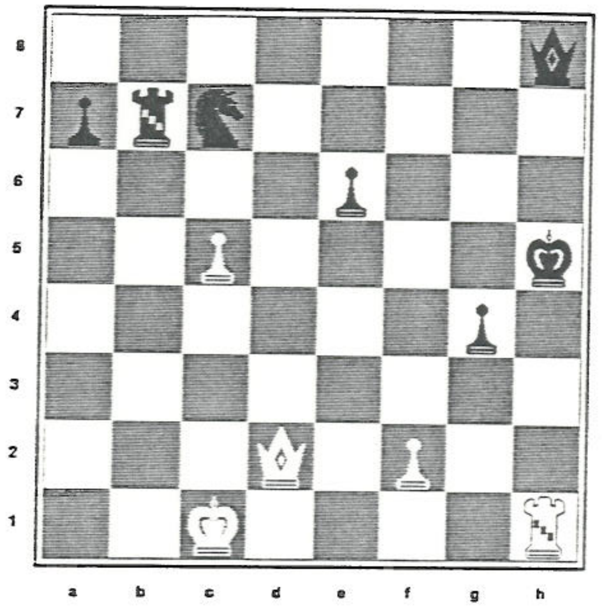
$TS2_2$, black-to-play



$TS1_3$, white-to-play



$TS2_3$, black-to-play



$TS2_4$, black-to-play

Figure 10: Additional training instances for the empirical test

Mnemonic	$TS2_1$	$TS1_1$	$C_{1,1}$	Interpretation of features
state	state1	state2	A	Current state.
side1	black	white	B	Current side to move.
side2	white	black	C	Opposite side side to move.
pk	bk1	wk1	D	Player of type king of side B.
sqk	d8	f3	E	Square occupied by the king D.
pthreat	wn1	bn1	F	Player of side C threatening the king.
tthreat	knight	knight	knight	Type of threatening player (F).
sqthreat	e6	d4	G	Square occupied by threatening player (F).
nthreat	1	1	1	Maximum length of threatening player's move.
dthreat	nnw	ees	H	Direction of checking threat.
sqin	{c8,d7,e8}	{e3,...,e2}	I	Square moved into out of check by king.
tin	empty	empty	empty	Contents of I.
din	{w,s,e}	{w,...,sw}	J	Direction of the king move out of check.
ptake	wn1	bn1	F	Player of side C capturing L.
ttake	knight	knight	knight	Type of capturing player (F).
sqtake	e6	d4	G	Square occupied by capturing player (F).
dtaken	ees	enn	M	Direction capture.
ntaken	1	1	1	Maximum length of capturing player's move.
sqtaken	g5	e6	K	Square occupied by L.
ptaken	bq1	wq1	L	Player of side 2 captured.
ttaken	queen	queen	queen	Type of L.

Table 2: Table showing all the features identified from $TS1_1$ and $TS2_1$

on a2 checks the king on e6 who must move out of check, allowing the rook on f7 to be captured.

Since all of the training instances are examples of "check with forced capture," they all have the same form of target concept—a conjunction of the incheck and badgoal literals from the domain theory. The explanations generated from each training instance also have a very similar structure. So similar, in fact, that we can show that each of the explanations can be defined as *a set of substitutions* to the modified general EBL explanation shown in Section 3.2. This can be seen by comparing the explanations produced from $TS1_1$ in Figure 5 and $TS2_1$ in Figure 6 to Figure 3. Under these conditions, since the concept definitions are identified by generalizing two explanations, the resulting concept is defined by the relationship between the substitutions. Indeed, the set of substitutions defining knight-fork given in Section 3.2 are derived directly from the comparison of the explanations substitutions for $TS1_1$ and $TS2_1$.

This comparison of substitutions is illustrated in Table 2. In Table 2 the $TS1_1$ column contains all the constant substitutions defining the explanation in Figure 6 and $TS2_1$ column contains all the constants substitutions defining the explanation in Figure 5. The fourth column gives the output of $Gen(TS2_1, TS2_1)$ for each constant pair and is termed $C_{1,1}$. The general EBL explanation and concept definition (referred to as C_{EBL}) would be represented in this table by 21 unique variables, one for each row. Hence, the knight-fork concept definition $C_{1,1}$, is a set of substitutions of C_{EBL}, determined by the relationship between the values in the $TS1_1$ and $TS2_1$ columns.

Another thing to note is how each substitution can be interpreted as a feature whose value is set by the training examples. Thus we view a concept definition as a general definition C_{EBL} and a set of substitutions, one for each variable in C_{EBL}, that are interpreted as a vector of features.

This view of concept definitions clarifies the space being searched by *Wyl2*. Given training examples that all have a common explanation structure (as with this test set) with n features, *Wyl2* can learn *any legal* instantiation of the n features. This is illustrated in Table 2 where the concepts learnable are any legal substitution of the 21 features listed. The substitutions can be specializations of C_{EBL} in the two ways introduced earlier: constant substitutions or identity constraints. When the feature value is a variable, the actual values it can take are restricted. This is because the instantiation of these features must also be a *legal* instantiation of the corresponding C_{EBL}. For example, in the definition given in Table 2 the variable `B` and `C` are not free to take any value from their legal range. Once `B` is bound to `black`, `C` can only be bound to `white`. Thus, a concept definition is not a simple linear sequence of independent features like in Quinlan's [1983] system. Rather, the range of values that each feature can take is determined *dynamically* by the domain theory constraints expressed in the generalized explanation structure.

Such a characteristic is what allows *Wyl2* to learn many concept definitions. Consider a set of training examples TS where each training example generates an explanation that has the same structure. *Wyl2* can learn a concept from any subset of TS, and hence has the potential to learn p distinct concept definitions, where p is the size of the power set of TS.

The chess example test introduced in this section selects a small subset of this power set to illustrate the principle.

5.2 Results of *Wyl2* in Chess

The twelve concept definitions learned from the two training sets are illustrated in Table 3. Each instance from the set $TS1$, $TS1_i$ is listed on the horizontal axis while each instance from set $TS2$, $TS2_j$, is listed on the vertical axis. Each location in the table gives the concept definition $C_{i,j}$ identified by *Wyl2* from instances $TS2_i$ and $TS1_j$. If the complete definition were given it would be of the form $C_{i,j}(a_1, a_2, \ldots, a_{21})$ where a_k, $1 \leq k \leq 21$, is the corresponding feature value. However, only a small subset of the features are included in the figure since the others are either not relevant to the test or common among all the concept definitions. The features included are

$$C_{i,j}(tthreat, pthreat, ptake, ttaken, dthreat, dtaken),$$

the type of threatening player, the threatening player, the capturing player, the type of captured player, the direction of checking threat, and the direction of capture. Feature values are either constants or variables. When a feature has the variable value `*` it can take any value independent of all other features. When a feature has the variable value `$` or `#`, it can take any value, but is not independent. If the variable is repeated in a concept definition, all locations must take the same value.

Note that in Table 3 that some of the concepts are repeated. The seven distinct concept definitions identified are illustrated in a generality hierarchy of Figure 11. The concepts found are the ones given earlier, and consist of forks and skewers. The important feature of a fork concept is the constraint that the threatening and capturing players be the same.

$TS2_1$ $C(\texttt{k,wn,wn,q,nnw,ees})$	$C_{1,1}(\texttt{k,\$,\$,q,*,*})$	$C_{2,1}(\texttt{*,\$,\$,*,*,*})$	$C_{3,1}(\texttt{*,\$,\$,q,*,*})$
$TS2_2$ $C(\texttt{r,wr,wr,b,e,w})$	$C_{1,2}(\texttt{*,\$,\$,*,*,*})$	$C_{2,2}(\texttt{*,\$,\$,*,*,*})$	$C_{3,2}(\texttt{r,\$,\$,*,*,*})$
$TS2_3$ $C(\texttt{b,wb,wb,r,ne,ne})$	$C_{1,3}(\texttt{*,\$,\$,*,*,*})$	$C_{2,3}(\texttt{b,\$,\$,r,*,*})$	$C_{3,3}(\texttt{*,\$,\$,*,\#,\#})$
$TS2_4$ $C(\texttt{r,wr,wr,q,n,n})$	$C_{1,4}(\texttt{*,\$,\$,q,*,*})$	$C_{2,4}(\texttt{*,\$,\$,*,*,*})$	$C_{3,4}(\texttt{r,\$,\$,q,\#,\#})$
	$TS1_1$ $C(\texttt{k,bn,bn,q,ees,enn})$	$TS1_2$ $C(\texttt{b,bb,bb,r,se,sw})$	$TS1_3$ $C(\texttt{r,br,br,q,s,s})$

Table 3: Concepts Learned in Chess Test

This is represented as $C(\texttt{-,\$,\$,-,-,-})$ in the feature vector. A skewer concept additionally constrains the directions of threat and capture to be the same. This is represented as $C(\texttt{-,-,-,-,\#,\#})$ in the feature vector. Below we give the english interpretation of the concepts learned:

$C_{ebl}(\texttt{*,*,*,*,*,*})$	Check with forced capture.
$C1(\texttt{k,\$,\$,q,*,*})$	Knight fork of king and queen that captures the queen.
$C2(\texttt{*,\$,\$,q,*,*})$	General fork of king and queen that captures the queen.
$C3(\texttt{*,\$,\$,*,*,*})$	General fork of king and another piece that captures the piece.
$C4(\texttt{r,\$,\$,*,*,*})$	Rook fork of king and another piece that captures the piece.
$C5(\texttt{b,\$,\$,r,*,*})$	Bishop fork of king and rook that captures the rook.
$C6(\texttt{*,\$,\$,*,\#,\#})$	General skewer of king that captures some piece.
$C7(\texttt{r,\$,\$,q,\#,\#})$	Rook skewer of king that captures the queen.

5.3 Evaluation of Results

In this section we evaluate *Wyl2* by considering how well the first four goals of a learning system introduced earlier are satisfied.

5.3.1 Learning Efficiency Goal

There are two aspects to learning efficiency. The first is the computational resources required to learn from a set of examples. The second is the number of training instances required to enable the system to learn effectively.

The resources required to learn are dominated by the time taken to construct the explanation. During this test the time taken to construct explanations and the time taken to

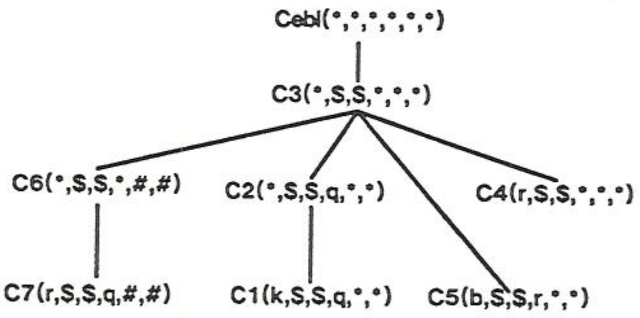


Figure 11: Concepts in Generality Hierarchy

generalize and compile the concept definitions was recorded[2]. The information is reproduced below:

Step in method	Average time (seconds)
Construct explanations	127
Generalize explanations	3
Compile explanations	4

The test illustrates that many interesting concepts can be learned by *Wyl2* from only two examples. Two examples will be sufficient if two can be found by the teacher that a) share a common explanation structure that define a vector of relevant features, b) exhibit the intended similarities and differences between these features. Consider the two lesson sets $\{TS1_1, TS2_3\}$ and $\{TS1_2, TS2_2\}$ that both generated the $C2($`*,$,$,*,*,*`$)$ illustrated below:

```
TS1₁  C(k,bn,bn,q,ees,enn)          TS1₂  C(b,bb,bb,r,se,sw)
TS2₃  C(b,wb,wb,r,ne ,ne )          TS2₂  C(r,wr,wr,b,e ,w )
      --------------------                ------------------
      C(*,$ ,$ ,*,*  ,*)                  C(*,$ ,$ ,*,* ,*)
```

Both sets of training instances share the same identity constraint among the *pthreat* and *ptake*, and different values for *ttaken* and *tthreat*. These two lesson sets are *equivalent* because they cause *Wyl2* to learn the same concept definition. In other words, they both exhibit the same properties above: they have the same common explanation structure and the same syntactic differences and similarities between their feature vectors.

The IOE method is nearly as learning efficient as the EBL method. Explanation construction and generalization are compatible while IOE usually requires at least one more training example.

[2]The program runs in Quintus Prolog on a Tektronix 4317.

5.3.2 Performance Efficiency

The goal of this test is twofold. First, we compare the computation required by the different concept definitions compared with the EBL definition and identify characteristics of concept definitions that affect efficiency. Second, we evaluate how the complexity of a board positions affects the computation required to recognize a concept. The performance efficiency is assessed by counting the number of logical inference steps (LIs) needed by concept definitions (i.e., prolog rules) to classify positive instances.

The results are show in Figure 12. The vertical axis of the graphs represents the count of logical inferences needed by the concept definitions to recognize a board position as an example of the concept. The horizontal axis represents a board position of increasing complexity. Graphs (a) and (b) show the results of using concepts C_{EBL}, $C3$,$C6$ and $C7$ to recognize a positive instance. The concepts were chosen because they exhibit a total ordering of generality from the most specific, $C7$ to the most general, C_{EBL} (see Figure 11).

The Base position is a simple rook-skewer example, comprising of only a rook of *side1*, and a king and queen both of *side2* (with *side2* side to play). The +knight position is the Base position with an added knight placed on the board in such a way as not to affect the concept definition. The +bishop is the +knight position with a bishop added similarly. The +rook and +queen represent further increases in complexity without effecting the concept definition. The two graphs (a) and (b) correspond to adding pieces of different sides. In graph (a) we add pieces of the attacking side (*side1*) while in graph (b) we add pieces of the defending side (*side2*).

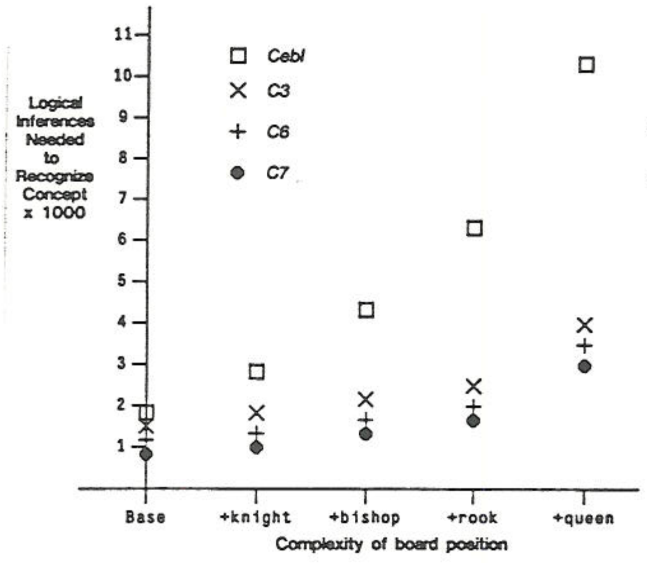
The graphs (c) and (d) are similarly set up for the knight-fork concepts C_{EBL}, $C3$,$C2$ and $C1$.

We draw the following conclusions from this data:

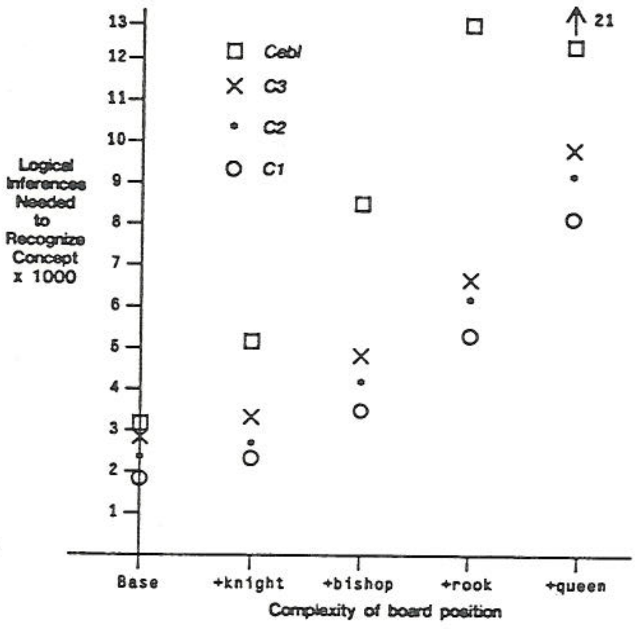
- **Claim:** Concepts of more specificity are more efficient.
  This has been observed by Segre, [1987]. It is clear in *Wyl2* that a less constrained definition will perform more generate and test in order to find a consistent set of bindings to the rule. For example, knight-fork ($C1$) is always more efficient than general-fork ($C3$) because $C1$ does not have to generate and test to find the correct checking piece. $C1$ directly binds the `knight`.

- **Claim:** There is little difference in efficiency among concepts when the complexity of a board position is due to pieces of the defending side.
  This is observed in both graph (b) and (d). The reason for this lies with the inefficient way that `forall` is compiled in the rules. Each definition generates *all* legal moves for the defending player. As the number of defending pieces increases the cost of this computation increases. This increase is due to the inefficient way that a legal move is expressed in the domain theory. When the playing side is already in check, this formalism wastes much work.

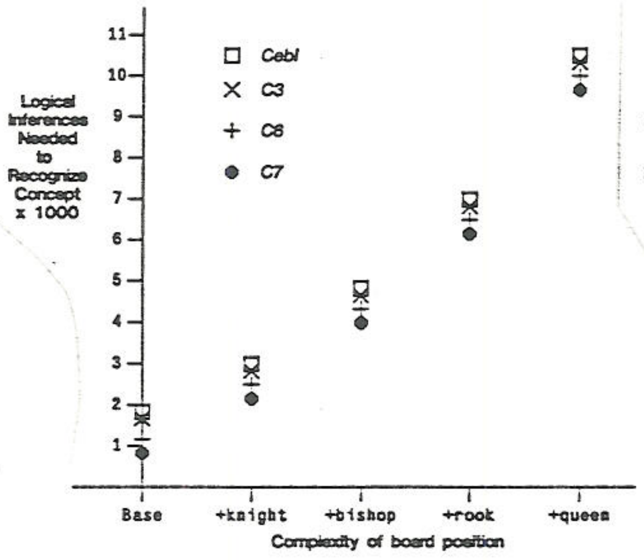
- **Claim:** Constant constraints make little difference in efficiency.
  In all the graphs of the concept definitions learned by the IOE method all lie within 2000 LIs. This demonstrates that the constant constraint on the type of capturing piece or checking piece has little effect in efficiency.



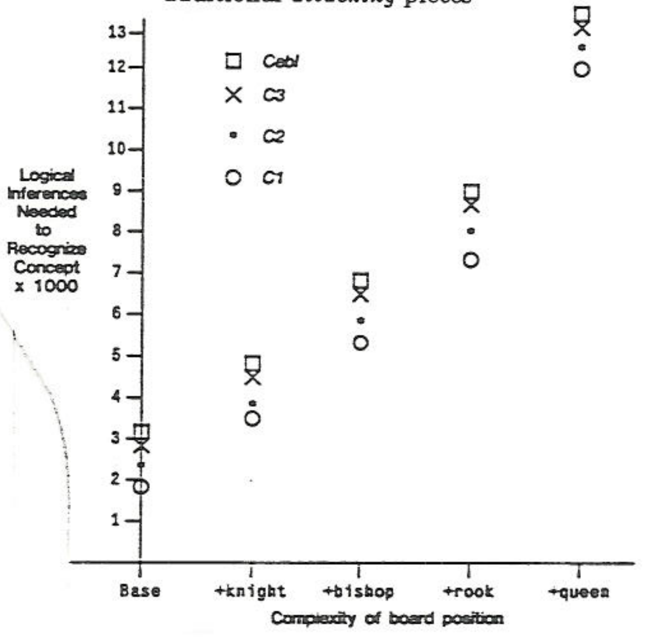
(a) Rook-skewer, Increasing board position complexity through additional *attacking* pieces



(c) Knight-fork, Increasing board position complexity through additional *attacking* pieces



(b) Rook-skewer, Increasing board position complexity through additional *defending* pieces



(d) Knight-fork, Increasing board position complexity through additional *defending* pieces

Figure 12: Results for evaluating performance efficiency

- **Claim:** Identity constraints can cause a great increase in efficiency when the complexity of a board position is due to pieces of the attacking side.
  From graph (a) and (b) it is clear that the identity constraint restricting the checking and capturing pieces be the same has a great impact on efficiency. The EBL rule (which cannot include such a constraint) is 2 to 3 times slower. This is due to the additional work of rebinding the capturing piece for each legal move.

The concept definitions generated by the IOE method are at least as efficient during performance as those generated by the EBL method. This is because IOE generates specializations of the rules generated by EBL. In fact, because of this specialization, much more efficient rules can be learned. The identity constraint in the fork concepts results in a 3 times speed up over EBL.

5.3.3 Flexibility and Correctness

This test clearly demonstrates the flexibility of IOE over EBL. The EBL system learns the same concept definition ("check with forced capture") from any of the seven training examples. The IOE method, on the other hand, learns many subtle specializations of this general concept. Where EBL learns a general definition C_{EBL}, IOE is capable of learning all legal specializations of C_{EBL}.

This additional flexibility improves the correctness of *Wyl2* and the IOE method over EBL for two reasons.

First, the IOE method produces a larger concept space than EBL, given the same domain theory. This increases the chance that the correct concept is representable. For example, in EBL it is very difficult to design a domain theory that could learn both bishop-skewer and general-fork.

Second, the IOE method can be more easily influenced by the environment or a teacher and is therefore more likely to identify the correct concept. It is hard for the teacher of an EBL system to influence the concept chosen because EBL chooses a concept definition from only one example. With IOE, because multiple examples are accepted, the teacher can choose examples that best express the intended concept.

Such an influence over the learning system will be beneficial if, given a training set, a teacher can predict with some degree of accuracy, the actual concept definition chosen. In the following discussion we argue that the IOE method and in particular *Wyl2* satisfies the correctness goal because it has this property. That is, *it is easy to teach.*

In order to show this let us follow the reasoning a teacher could use to choose the training set $\{TS1_1, TS2_1\}$ to teach the concept "Knight fork of king and queen that captures the queen." The teacher must know something of how *Wyl2* works and what it knows. Let us suppose, therefore, the teacher knows that *Wyl2* possesses a declarative encoding of the rules of the game of chess and that it follows the IOE method. With this knowledge, we claim, the teacher can identify the features *Wyl2* will first generate in explanation construction and compare in generalization to form the concept definition. Once the teacher knows the features used by *Wyl2* it is a simple process to find two (or perhaps more) training instances that exhibit the intended similarities and differences among these features.

Consider the two training examples given in Table 2 where the 21 features where identified. Each feature was directly involved in the proof of check or the unavoidable piece capture. For example, in Figure 1, the type of the playing piece on square e6 was identified as a relevant feature while the type of the playing piece on square b8 was ignored. This is because the piece on e6 was involved in the capture and threat to the king (i.e., involved in the proof of the target concept) while the piece on square b8 played no part in the outcome. Hence, the set of features generated by *Wyl2* are those that are needed in the proof of the target concept among the instances.

The teacher will know, therefore, what features are generated by *Wyl2* because he knows what aspects of the training examples will be used in proving the target concept. It is now a simple task to find examples that will set the feature values as intended. For example, in teaching knight-fork, two examples where chosen that shared the feature values of the type of checking and capturing piece and the type of piece taken. The two examples differed in the other feature values that the teacher did not intend to be retained as constants in the definition.

To conclude, because the IOE method generates a much larger concept space than EBL and is easier to teach, it offers both high correctness and flexibility.

6 Conclusions

In this paper we have presented a new learning method: Induction Over Explanations and compared it to both SBL and EBL. Through examples and an empirical study we have demonstrated that IOE is superior to these alternative methods.

In this section we conclude by considering two reasons for the success of the IOE method. Both reasons attribute this success to a better distribution of responsiblites for achieving two of the important learning criterion introduced earlier: performance efficiency and correctness. We show how IOE shifts these responsibilities from the learning system designer to the teacher of the learning system and to the learning system itself.

IOE represents a better distribution of the responsibility for achieving performance efficiency over SBL. In SBL, this responsibility lies with the system designer since the method does not include the compilation step found in the EBL and IOE methods. The concept representation supplied to the learning system by the designer must be both effective for learning and efficient for performance. This is an unreasonable burden on the system designer for two reasons: First, it is often the case that the most suitable representation for learning is incompatible with performance. Quinlan's work with chess end games demonstrated this problem (as described in Flann and Dietterich, 1986). Second, by not incorporating a compilation stage, the system designer is forced to encode any knowledge the learning system is to use in an already complied form. Compiled knowledge is traditionally much more complex and hard to formalize than a purely declarative encoding.

Methods that incorporate a compilation stage (such as IOE) are more in keeping with the trend in software engineering and logic programming: shifting the burden of performance efficiency from the system designer to the system itself.

IOE represents a better distribution of the responsibility for achieving correctness over EBL. We have shown previously that in EBL the responsibility for correctness lies with the system designer, while in IOE, the responsibility lies with the teacher. To see this clearly we identify four factors that effect a concept definition and consider for each method the agent

Factor	EBL	IOE
Constants	**System designer:** must engineer the domain theory with appropriate rule constants	**Teacher:** must select training instances with the appropriate similarities and differences
Identity	**System designer:** must engineer the domain theory with appropriate identity constraints	**Teacher:** must select training instances with the appropriate repeated patterns
Generalize-to-n	**System designer:** must choose the policy at design time	**Teacher:** must select training instances with the appropriate similarities and differences
Truncation	**System designer:** must choose the policy at design time	**Teacher:** must select training instances with the appropriate similarities and differences

Table 4: Comparison of IOE and EBL: Responsibility for correctness

(the teacher or designer) that must take responsibility for setting these factors. The four factors have all been introduced previously and are listed below:

Constants Which constants are to be included in the concept definition?

Identity Which identity constraints are to be included in the concept definition?

Generalize-to-n Which recursive invocations of rules are to be included in the concept definition?

Truncation Which predicates are to be prematurely truncated in the explanation construction and hence in the concept definition?

In Table 4 we summarize the responsibilities with respect to the two methods, IOE and EBL. With EBL, these factors must all be decided at *design time*, thereby placing a burden on the system designer and losing flexibility. In contrast, with IOE, all the decisions are made by the teacher at *instruction time*, removing this burden from the system designer and increasing flexibility.

7 Acknowledgements

Discussions with Rich Keller helped clarify the issue of correctness in EBL systems. Correspondence with Haym Hirsh clarified the issue of back-propagation in EBL systems. Caroline Koff, Giuseppe Cerbone, Jim Holloway, and Ritchey Ruff provided helpful comments on earlier drafts of this paper.

This research was partially supported by a Tektronix Graduate Fellowship (to Flann) and by the National Science Foundation under grant numbers IST-8519926 and DMC-8514949.

8 References

DeJong, G., and Mooney, R. (1986). Explanation-Based Learning: An Alternative View, in *Machine Learning* Vol 1, No. 2, 145-176, 1986.

Dietterich, T. G., London, B., Clarkson, K., and Dromey, G., Learning and Inductive Inference, in *The Handbook of Artificial Intelligence,* Vol, 3, P. R. Cohen and E. A. Feigenbaum (Eds.) Kaumann, 1982.

Flann, N. S. and Dietterich T. G. (1986). Selecting Appropriate Representations for Learning from Examples, in *Proceedings of the Fifth National Conference on Artificial Intelligence,* 1986.

Genesereth, M. R., and Nilsson, N. J. (1987). *The Logical Foundations of Artificial Intelligence,* Morgan Kaufmann Pub. Los Altos.

Hirsh, H. (1987). Explanation-Based Generalization in a Logic-Programming Environment, in *Proceeding of the 10th International Joint Conference on Artificial Intelligence,* 1987.

Kedar-Cabelli, S. T. and McCarthy, L. T. (1987). Explanation-Based Generalization as Resolution Theorem Proving, *Proceedings of the Fourth International Workshop on Machine Learning,* 1987.

Michalski, R. S., (1980). Knowledge Acquisition Through Conceptual Clustering: A Theoretical Framework and an Algorithm for Partitioning Data into Conjunctive Concepts, in *Policy Analysis and Information Systems,* Vol. 4, No. 3, pp. 219-44. 1980.

Minton, S. (1984). Constraint-Based Generalization: Learning Game-Playing Plans from Single Examples, in *Proceedings of the Third National Conference on Artificial Intelligence,* 1984.

Mooney, R. J., and Bennett, S. W. (1986). A Domain Independent Explanation-Based Generalizer, in *Proceedings of the Fifth National Conference on Artificial Intelligence,* 1986.

Mitchell, T.M., (1977). Version Spaces: A Candidate Elimination Approach to Rule Learning, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* Cambridge, Mass., 305-310, 1977.

Mitchell, T., Keller, R., and Kedar-Cabelli, S. (1986). Explanation-Based Generalization: A Unifying View, in *Machine Learning,* Vol. 1, No. 1, pp. 47-80, 1986.

Quinlan, J. R., (1983). Semi-Autonomous Acquisition of Pattern-Based Knowledge, in *Machine Intelligence,* Vol. 10, D. Michie, J. E. Hayes, and Y. H. Pao (Eds.). 1982.

Quinlan, J. R., (1983). Learning Efficient Classification Procedures and their Application to Chess End Games in *Machine Learning: An Artificial Intelligence Approach, Vol I.* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), Tioga Press, Palo Alto, 1983.

Soloway, E. M., (1978). Learning = Interpretation + Generalization: A Case Study in Knowledge-Directed Learning, Ph. D. diss. Department of Computer and Information Science, University of Massachusetts, 1978.

Segre, A., (1987). On the Operationality/Generality Trade-Off in Explanation Based Learning, in the *Proceedings of the Tenth International Joint Conference on Artificial Intelligence,* Milan., 1977.

Tadepalli, T., (1987). Learning from Intractable Theories, unpublished thesis proposal, Rutgers University, 1987.