Machine Learning, 26, 227–252 (1997)

PAL: A Pattern–Based First–Order Inductive System

EDUARDO F. MORALES emorales@campus.mor.itesm.mx
ITESM – Campus Morelos, Apto. Postal C–99, Cuernavaca, Morelos, 62050, México

Editors: Stephen Muggleton and David Page

Abstract. It has been argued that much of human intelligence can be viewed as the process of matching stored patterns. In particular, it is believed that chess masters use a pattern–based knowledge to analyze a position, followed by a pattern–based controlled search to verify or correct the analysis. In this paper, a first–order system, called PAL, that can learn patterns in the form of Horn clauses from simple example descriptions and general purpose knowledge is described. The learning model is based on (i) a constrained least general generalization algorithm to structure the hypothesis space and guide the learning process, and (ii) a pattern–based representation knowledge to constrain the construction of hypothesis. It is shown how PAL can learn chess patterns which are beyond the learning capabilities of current inductive systems. The same pattern–based approach is used to learn qualitative models of simple dynamic systems and counterpoint rules for two–voice musical pieces. Limitations of PAL in particular, and first–order systems in general, are exposed in domains where a large number of background definitions may be required for induction. Conclusions and future research directions are given.

Keywords: first–order induction, ILP, chess, qualitative model, music.

1. Introduction

It is believed that chess masters use a pattern–based knowledge to analyze a position, followed by a pattern–based controlled search to verify or correct the analysis (Charness, 1977, de Groot, 1965). For example given the position of Figure 1, a chess player recognizes that:

- A white Rook *threatens* the black Queen.
- The black Bishop is *pinned*.
- The white Queen is *threatened* by a black Pawn.
- The white Knight *can fork* the black King, the black Rook and the black Knight.
- Moving the foremost white Pawn can *discover a threat*, create a *pin*, and possibly a *skewer*.
- ...

This analysis involves the recognition of concepts like, *threat*, *pin*, *discovered threat*, *fork*, *skewer*, ..., etc., which can then be used to choose a move (e.g., move the white Knight and check the black King). Previous work has shown how playing strategies can be constructed following a pattern–based approach (Berliner, 1977, Bramer, 1977, Bratko, 1982, Huberman, 1968, Pitrat, 1977, Wilkins, 1979). However, a substantial programming effort

needs to be devoted to the definition and implementation of the *right* patterns for the task. There have been also some attempts to learn chess concepts from examples described with a set of attributes (Quinlan, 1983, Shapiro, 1987). In this case too, most of the work is dedicated to the definition of adequate attributes to describe the examples and from which the target concept can be constructed. This work investigates whether chess patterns (such as those described above) which are powerful enough for play can be acquired by machine learning techniques from simple example descriptions.

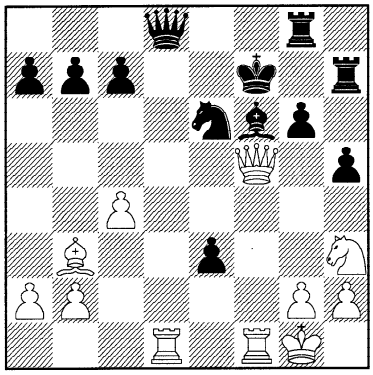

Figure 1. An example position

The limited expressiveness of attribute–based learning systems has lead to an increased interest in learning from first–order logical descriptions, despite the increased complexity of the learning problem. First–order learning systems have used background knowledge to induce concepts from examples. This is important as it allows a simpler (and often more natural) way to represent examples. Background knowledge can help as well to reduce the inductive steps taken when learning particular concepts. Furthermore, a basic core of background knowledge definitions can be used to learn several concepts.

Learning the chess concepts we are interested in is a non–trivial task and beyond the capability of existing systems. There are several technical issues that need to be addressed:

1. Chess requires a relatively large amount of background knowledge. This creates severe search problems when the background knowledge is used in learning.

2. Chess concepts are inherently non–determinate[1]. This is a problem for first–order learning mechanisms.

3. Chess concepts are learned incrementally. This leads to a requirement that the learned knowledge can be "recycled". The most effective way to do this is to have the background knowledge in the same form as the induced knowledge.

4. In many practical domains where the trainer does not understand the details of the learning mechanism it is important that the learning mechanism is robust with respect to the examples presented, and the order in which the examples are presented. This is particularly so with first–order systems which are particularly sensitive to such variations since their inductive steps are powerful and underdetermined.

These issues are addressed by PAL. The central theme is that the notion of a pattern, defined as a set of relations between components of a state (such as a chess position), allows the efficiency issues to be addressed without compromising the quality of the induced knowledge. In addition, PAL's learning task is not–standard, in the sense that the name and the number of arguments of the target concept are not explicitly stated in the examples.

Although this research was originally centered around chess, pattern–based reasoning is not exclusive to chess. It has been argued that a large part of human intelligence can be viewed as the process of matching stored patterns (Campbell, 1966, Lorenz, 1973). It is shown that the same pattern–based approach can be used in qualitative modeling and in music, where other learning systems have similar difficulties to those experienced in chess.

Section 2 provides some definitions from logic. The concepts and notations will be used in the sections to follow. Section 3 briefly describes PAL and its generalization method. Section 4 shows how PAL is used to learn several chess concepts, such as those illustrated in Figure 1. In section 5, PAL is used to learn a qualitative model of a simple dynamic system. In section 6, counterpoint rules for two–voice musical pieces are learned by PAL. Section 7 discusses PAL's main limitations and its relation to previous work. Finally, conclusions and future research directions are given in section 8.

2. Preliminaries

A *variable* is represented by a string of letters and digits starting with an upper case letter. A *function symbol* is a lower case letter followed by a string of letters and digits. A *predicate symbol* is a lower case letter followed by a string of letters and digits. A *term* is a constant, variable or the application of a function symbol to the appropriate number of terms. An *atom* or *atomic formula* is the application of a predicate symbol to the appropriate number of terms. A *literal* is an atom or the negation of an atom. Two literals are *compatible* if they have the same symbol, name and number of arguments. The negation symbol is $\neg$. A *clause* is a disjunction of a finite set of literals, which can be represented as $\{A_1, A_2, \ldots, A_n, \neg B_1, \ldots, \neg B_m\}$. The following notation is equivalent:

$$A_1, A_2, \ldots, A_n \leftarrow B_1, B_2, \ldots, B_m.$$

A *Horn clause* is a clause with at most one positive literal (e.g., $H \leftarrow B_1, \ldots, B_m$). The positive literal (H) is called the *head*, the negative literals (all B_is) the *body*. A clause with empty body is a *unit clause*. A set of Horn clauses is a *logic program*. F_1 *syntactically entails* F_2 (or $F_1 \vdash F_2$) iff F_2 can be derived from F_1 using the deductive inference rules. A *substitution* $\Theta = \{V_1/t_1, V_2/t_2, \ldots, V_n/t_n\}$ consists of a finite sequence of distinct variables paired with terms. An *instance* of a clause C with substitution Θ, represented by $C\Theta$, is obtained by simultaneously replacing each occurrence of a component variable of

Θ in C by its corresponding term. A *model* of a logic program is an interpretation for which the clauses express true statements. We say that F_1 *semantically entails* F_2 (or $F_1 \models F_2$, also F_1 logically implies or entails F_2, or F_2 is a logical consequence of F_1), iff every model of F_1 is a model of F_2.

3. Generalization method and PAL

Inductive logic programming (ILP) is a fast growing research area which combines logic programming with machine learning (Muggleton, 1992). A general setting for ILP is, given a background knowledge $\mathcal{K}$ (in the form of Horn clauses) and sets of positive ($\mathcal{E}^+$) and negative ($\mathcal{E}^-$) examples, find a hypothesis $\mathcal{H}$ (another set of Horn clauses) for which $\mathcal{K} \wedge \mathcal{H} \vdash \mathcal{E}^+$ and $\mathcal{K} \wedge \mathcal{H} \nvdash \mathcal{E}^-$. That is, find a hypothesis which can explain the data in the sense that all the positive ($\mathcal{E}^+$) but none of the negative ($\mathcal{E}^-$) examples can be deduced from the hypothesis and the background knowledge. This inductive process can be seen as a search for logic programs over the hypothesis space and several constraints have been imposed to limit this space and guide the search. For learning to take place efficiently, it is often crucial to structure the hypothesis space. This can be done with a model of generalization. Searching for hypothesis can then be seen as searching for more general clauses given a known specialized clause.

Plotkin (Plotkin, 1969, Plotkin, 1971a, Plotkin, 1971b) was the first to study in a rigorous manner the notion of generalization based on Θ-subsumption. Clause C Θ-subsumes clause D iff there exists a substitution σ such that $C\sigma \subseteq D$. Clause C_1 is more general than clause C_2 if C_1 Θ-subsumes C_2. Plotkin investigated the existence and properties of least general generalizations or *lgg* between clauses and the *lgg* of clauses relative to some background knowledge or *rlgg*. That is, generalizations which are less general, in terms of Θ-subsumption, than any other generalization. The *lgg* algorithm replaces all the different terms that have the same place within compatible literals by new variables (see (Plotkin, 1969) for more details). For example, if we have two compatible literals:

L_1 = threat(white,rook,square(1,3),black,bishop,square(4,3))
L_2 = threat(white,queen,square(1,4),black,bishop,square(7,4))
then: *lgg*(L_1, L_2) =
threat(white,Piece1,square(1,Y),black,bishop,square(X,Y))

This generalization process is repeated between all the pairs of compatible literals within clauses. That is, the *lgg* of two clauses C_1 and C_2 is defined as: $\{l : l_1 \in C_1$ and $l_2 \in C_2$ and $l = lgg(l_1, l_2)\}$.

More recently, Buntine (Buntine, 1988) defined a model–theoretic characterization of Θ-subsumption, called *generalized subsumption* for Horn clauses (see (Buntine, 1988) for more details). Buntine also suggested a method for constructing *rlgg*s using Plotkin's *lgg* algorithm between clauses. The general idea of the *rlgg* algorithm is to augment the body of the example clauses with facts derived from the background knowledge definitions ($\mathcal{K}$) and the current body of the example clauses, and then generalized these "saturated" clauses using Plotkin's *lgg* algorithm.

- **given:**
  - a logic program ($\mathcal{K}$)
  - a set of example clauses (*SC*)
- Take an example clause (C_1) from *SC*. Let $\theta_{1,1}$ be a substitution grounding the variables in the head of C_1 to new constants and $\theta_{1,2}$ grounding the remaining variables to new constants
- Construct a new clause (*NC*) defined as:
  $NC \equiv C_1\theta_{1,1} \cup \{\neg A_{1,1}, \neg A_{1,2}, \ldots\}$ where
  $\mathcal{K} \wedge C_{1body}\theta_{1,1}\theta_{1,2} \models A_{1,i}$, and $A_{1,i}$ is a ground atom
- Set *SC* = *SC* $-\{C_1\}$
- **while** *SC* $\neq \{\emptyset\}$
  - Take a new example clause (C_j) from *SC*. Let $\theta_{j,1}$ be a substitution grounding the variables in the head of C_j to new constants, and $\theta_{j,2}$ grounding the remaining variables to new constants
  - Construct a new clause (C'_j) defined as:
    $C'_j \equiv C_j\theta_{j,1} \cup \{\neg A_{j,1}, \neg A_{j,2}, \ldots\}$ where
    $\mathcal{K} \wedge C_{jbody}\theta_{j,1}\theta_{j,2} \models A_{j,k}$ and $A_{j,k}$ is a ground atom
  - Set *NC* = *lgg*(C'_j,*NC*)
  - Set *SC* = *SC* $-\{C_j\}$
- **output** *NC*

Table 1. A plausible *rlgg* algorithm for a set of example clauses

Following (Buntine, 1988), the basis for a learning algorithm using *rlgg* is as follows (see Table 1): Let $\mathcal{K}$ be a logic program or theory, $\theta_{i,1}$ be a substitution grounding the variables in the head of clause C_i to new constants and $\theta_{i,2}$ grounding the remaining variables to new constants. If *rlgg*$(C_1, C_2, \ldots, C_n)$ exists, it is equivalent w.r.t. $\mathcal{K}$ to the $lgg(C'_1, C'_2, \ldots, C'_n)$, where for $1 < i < n$,

$$C'_i \equiv C_i\theta_{i,1} \cup \{\neg A_{i,1}, \neg A_{i,2}, \ldots\} \tag{1}$$

where $\mathcal{K} \wedge C_{ibody}\theta_{i,1}\theta_{i,2} \models A_{i,k}$, $A_{i,k}$ is a ground atom, and the $A_{i,k}$s are all the possible ground atoms deduced from the theory. Equation 1 can be rewritten as:

$$C'_i \equiv C_{ihead}\theta_{i,1} \leftarrow C_{ibody}\theta_{i,1}, A_{i,1}, A_{i,2}, \ldots \tag{2}$$

Let the resulting least general generalization of N clauses be denoted as:

$$GC(n) = lgg(C'_1, C'_2, \ldots, C'_n).$$

Since $lgg(C'_1, C'_2, \ldots, C'_n) \equiv lgg(C'_1, lgg(C'_2, \ldots, lgg(C'_{n-1}, C'_n)\ldots))$ (Plotkin, 1971b), then,

$$GC(n) = lgg(C'_n, GC(n-1)).$$

This can be used for constructing the *rlgg* of a set of clauses. If a set of examples is described with a set of clauses, a learning algorithm can accept a new example, construct a clause with it and atoms derived from the background knowledge (logic program), and gradually generalize the clause by taking *lgg*s of this clause and subsequent clauses constructed from new examples until meeting a termination criterion. PAL's learning algorithm is based on this framework. A direct implementation of it is impractical for all but the simplest cases, as it essentially involves the deduction of all ground atoms logically implied by the theory (see (Niblett, 1988) for a more thorough discussion on generalization). However, *rlgg* exists for theories without variables (as in Golem (Muggleton & Feng, 1990)), theories without function symbols (as in Clint (De Raedt & Bruynooghe, 1988)), and when only a finite number of facts are deducible from the theory, either by limiting the depth of the resolution steps taken to derive facts and/or by constraining the background knowledge definitions, as in PAL. Even with a finite set of facts, the *lgg* of two clauses can generate a very large number of literals. The main problem is that *lgg* produces very small generalization steps and some heuristics are required to converge faster into a solution to achieve practical results. PAL (i) uses a pattern–based background knowledge representation to derive a finite set of facts and (ii) applies a novel constraint which identifies the role of the components in different example descriptions to reduce the complexity of the *lgg* algorithm (these will be explained below).

3.1. PAL

Examples in PAL are given as sets of ground atoms (e.g., descriptions of chess positions stating the position of each piece in the board), and unlike other systems, the name and the exact arguments involved in the target concept are not specified in advance. A chess position can be completely described by a set of four–place atoms (*contents/4*) stating the side, name, and place of each piece in the board. For instance, *contents*(*white, rook, square*(*2,3*), *pos1*) states that a white Rook is at the second file and third rank in position 1. Other pieces in a board position can be described in the same way. In general, other descriptions can be used as well (see (Morales, 1992)) and we will see other example descriptions for different domains. Each example description is added to the background knowledge from which a finite set of facts is derived.

In the context of chess a pattern refers to a relation between pieces and places in the board. More generally patterns arise in any domain which can be represented by states which have an internal structure, with well defined components and relations between the components that define the particular pattern. PAL induces pattern definitions with the following format:

$$Head \leftarrow D_1, D_2, \ldots, D_i, F_1, F_2, \ldots$$

where,

- *Head* is the head of the pattern definition. Instantiations of the head are regarded as the patterns recognized by the system.
- The D_is are "input" predicates used to describe positions (i.e., *contents/4*) and represent the components which are involved in the pattern.

- The F_is are instances of definitions which are either provided as background knowledge or learned by PAL, and represent the conditions (e.g., relations between pieces and places) to be satisfied by the pattern.

In the context of chess, PAL can learn patterns which are associated with a particular move. Only one move ahead is considered during the learning process; however, once a pattern is learned, it can be used to learn other patterns as well. *Make_move* is a predicate that changes the current state of the board description. The *make_move* predicate defines 1-ply moves,[2] instantiations of this predicate represent the different possible 1-ply movements. To learn such patterns, the actual movement of a piece is performed (changing the description of the board) and new patterns (ground atoms) are generated (deduced) after each move.

For such patterns PAL induces concepts with the following format:

$$
\begin{aligned}
Head \leftarrow\ & \\
& D_1, D_2, \ldots, D_k, \\
& F_1, F_2, \ldots, F_m, \\
& MV_1, F_{1,1}, F_{1,2}, \ldots, F_{1,n}, \\
& MV_2, F_{2,1}, F_{2,2}, \ldots, F_{2,p}, \\
& \vdots \\
& MV_r, F_{r,1}, F_{r,2}, \ldots, F_{r,s}.
\end{aligned} \tag{3}
$$

where,

- MV_i is an instance of the *make_move* predicate representing a legal move of one piece with the opponent's side not in check, and the $F_{i,j}$'s are instances of pattern definitions that change as a consequence of the move. Each *make_move* predicate can be seen as a "what if" move, where MV_i is the actual movement and the $F_{i,j}$s are the consequences that occur if that particular move is followed.

Only movements which introduce a new predicate name or remove an existing predicate name after the move are considered. The complement[3] of the new predicate is added before the move. No equivalent definitions to the *make_move* predicate have been considered in other domains, however, in a qualitative model's domain (see section 5) it would correspond to a change on a qualitative state.

PAL starts with some pattern definitions in its background knowledge and use them to learn new patterns. For instance, the definition of *being in check* is given to PAL as follows:

```
in_check(Side,KPlace,OPiece,OPlace,Pos) ←
        contents(Side,king,KPlace,Pos),
        contents(OSide,OPiece,OPlace,Pos),
        other_side(Side,OSide),
        piece_move(Oside,OPiece,OPlace,KPlace,Pos).
```

where *contents/4* are "input" predicates (D_is) while *other_side/2* and *piece_-move/5* are background knowledge definitions (F_js). This definition gets instantiated only with example descriptions with a King at *KPlace* and an opponent's piece *OPiece* at *OPlace* which could be moved to *KPlace*.

Given an example description, PAL "collects" instantiations of its pattern–based background knowledge definitions to construct an initial hypothesis clause. The head of the clause is initially constructed with the arguments used to describe the first example description. The initial head, in conjunction with the facts derived from the background knowledge and the example description, constitutes an initial concept clause. This clause is generalized by taking the *lgg* of it and clauses constructed from other example descriptions.

Even with a finite theory for chess, the large number of plausible facts derivable from it, makes the finiteness irrelevant in practice (e.g., consider all the possible legal moves of pieces in chess). In PAL a fact F is *relevant* to example description D if at least one of the ground atoms of D occurs in the derivation of F. Since PAL constructs its clauses using pattern–based definitions, only a finite set of relevant facts are considered[4].

For instance, if we only have a white Bishop and a black Pawn in a chess position described as follows:

```
contents(white,bishop,square(2,3),pos1).
contents(black,pawn,square(3,4),pos1).
```

and PAL's only pattern–definition is *legal_move/5*, then PAL produces the following clause involving only all the legal moves of the pieces in the chess position:

```
tmp(white,bishop,square(2,3),black,pawn,square(3,4),pos1) ←
    contents(white,bishop,square(2,3),pos1),
    contents(black,pawn,square(3,4),pos1),
    legal_move(white,bishop,square(2,3),square(1,4),pos1),
    legal_move(white,bishop,square(2,3),square(3,2),pos1),
    legal_move(white,bishop,square(2,3),square(4,1),pos1),
    legal_move(white,bishop,square(2,3),square(3,4),pos1),
    legal_move(white,bishop,square(2,3),square(1,2),pos1),
    legal_move(black,pawn,square(3,4),square(2,3),pos1),
    legal_move(black,pawn,square(3,4),square(3,3),pos1).
```

3.2. Novel constraints

A common constraint in ILP systems, is to limit the size of the resulting generalized clause by requiring all the variables arguments to appear at least twice in the clause (e.g., (De Raedt & Bruynooghe, 1988, Muggleton & Feng, 1990, Rouveurol, 1991)). In addition, PAL uses a novel constraint based on labeling the different components which are used to describe examples to guide and constrained the *lgg* algorithm. Every constant occurring in the atoms used to describe examples is labeled with a unique constant symbol. For instance, in the example position given above, the *Bishop* and the *Pawn* are represented

as follows (for presentation purpose we have adopted the following notation: bl = black, wh = white, and square(X,Y) = (X,Y)):

contents(wh,bishop,(2,3),pos1) $\longrightarrow$ contents(wh$_\alpha$,bishop$_\beta$,(2$_\gamma$,3$_\delta$),pos1)
contents(bl,pawn,(3,4),pos1) $\longrightarrow$ contents(bl$_\nu$,pawn$_\mu$,(3$_\lambda$,4$_\psi$),pos1)

The labels are kept during the derivation process, so the system can distinguish which component(s) is(are) "responsible" for which facts derived from the background knowledge by following the labels. So the instances of *legal_move/5* which involve the *bishop* use the same labels associated with the description of the *bishop*, i.e.,

legal_move(wh,bishop,(2,3),(4,1),pos1) $\longrightarrow$
legal_move(wh$_\alpha$,bishop$_\beta$,(2$_\gamma$,3$_\delta$),(4,1),pos1)
legal_move(wh,bishop,(2,3),(3,4),pos1) $\longrightarrow$
legal_move(wh$_\alpha$,bishop$_\beta$,(2$_\gamma$,3$_\delta$),(3$_\lambda$,4$_\psi$),pos1)
...

The predicates used to describe examples are part of the pattern definitions (e.g., *contents/4*). When instantiations of these patterns are collected for each example description, PAL identifies the instantiations of the description predicates (i.e., the components involved in the instantiation of the pattern). The *lgg* between compatible literals is guided by the associated labels to produce a smaller number of literals, as *lgg*s are produced only between compatible literals with common labels (a simple matching procedure is used for this purpose). In chess this constraint means that legal moves of non–corresponding pieces are not considered in the *lgg* algorithm. Without this constraint, PAL can produce, as other ILP systems based on *lgg*, very long clauses, requiring of additional constraints to achieve practical results. This constraint has been successfully used in three different domains, where a pattern–based approach can be used.

For instance, the above chess position has 5 legal moves for the *Bishop* and 2 for the *Knight*. A second position where the Bishop is changed to a black Pawn at *square*(4,2), and the Pawn to a white Knight at *square*(5,4), has 1 legal move for the Pawn and 8 for the Knight. The labels used in the first position for the first piece are associated with the first piece of the second example. The *lgg* algorithm without labels produces 63 literals for the legal move predicate (i.e., $(5+2)\times(1+8)$). With labels it produces 21 ($5\times1+2\times8$). To recognize the corresponding pieces, the example generator (described below) knows which pieces are changed and associates their corresponding labels. Examples which are manually provided or selected at random require that the corresponding components are presented in the same order. This is equivalent to give the examples as sets of attribute–value pairs and not allow to shift the values of the attributes in the description of the examples. PAL's learning algorithm is described in Table 2.

3.3. Automatic example generator

PAL can follow an experimentation process by automatically generating positive and negative examples (validated by the user). PAL's example generator changes the values of the

given:

- a logic program $\mathcal{K}$
- a set of pattern definitions $\mathcal{P}$, such that $\mathcal{P} \subseteq \mathcal{K}$
- a set of positive ($\mathcal{E}^+$) and negative ($\mathcal{E}^-$) examples each one described as a set of ground unit clauses

select an example $E_1 \in E^+$
construct a new clause (*NC*) defined as: $NC \equiv C_1 \leftarrow E_1 \cup A_{1,1}, \cup A_{1,2}, \ldots$ where:

- $\mathcal{K} \wedge E_1 \vdash A_{1,i}$, and $A_{1,i}$ is a ground instance of a pattern definition in $\mathcal{P}$
- C_1 is a head clause constructed from the arguments used in E_1 and a new predicate name PN

set $E^+ = E^+ - \{E_1\}$
while $E^+ \neq \{\emptyset\}$

- **select** a new example description $E_j \in E^+$
- **construct** a new clause (C'_j) defined as: $C'_j \equiv C_j \leftarrow E_j \cup A_{j,1}, \cup A_{j,2}, \ldots$ where:
  - $\mathcal{K} \wedge E_j \vdash A_{j,i}$, and $A_{j,i}$ is a ground instance of a pattern definition in $\mathcal{P}$
  - C_j is a head clause constructed from the arguments used in E_j and predicate name PN
- **set** *NC* = *clgg*(C'_j,*NC*) where *clgg* is an *lgg* between literals with the same labels
- **if** NC covers an example in E^-, **then** reject NC, save E_j in a disjunct list DL, and **continue**

output *NC*
if there are examples in DL not covered by an *NC*,
then set $E^+ = DL$ and start the whole process again.

Table 2. PAL's learning algorithm

arguments involved in the example description to produce new examples. For instance, it changes the place of a piece, its side, or changes one piece for another. Each new example is tested against the current concept definition and only presented to the user when it is not an instance of the current definition. PAL uses domain dependent knowledge of the possible values of the arguments used to describe positions (e.g., possible pieces, places and sides). If the perturbation method generates a negative example, then the system analyses which literals failed on that example and tries to construct a new example that will succeed on at least one of them. If a positive example is generated, then PAL constructs a clause as defined above, and generalizes it using the constrained *lgg* algorithm, with the current concept definition. If the system cannot generate a new example (i.e., a new generalization of the current definition will require producing an example that involves changing different arguments), then PAL continues with a different set of arguments. PAL first considers sets with the smallest number of arguments, e.g., it changes the position of each piece at a time,

before changing the positions of several pieces at a time. PAL stops when there are no more sets left, or when the user decides to terminate the process (a more detailed description is given in (Morales, 1992)).

The *lgg* algorithm constructs a single clause (the least general generalized clause) from a set of clauses. In order to learn disjunctive definitions (i.e., definitions which required more than one clause), systems which use *lgg* must rely on the negative examples produced by the system and/or provided by the user. This is also true for most inductive learning algorithms. In PAL each new definition is checked against the current negative examples. If a definition covers a negative example, it is rejected and the example is stored. When the perturbation process finishes or all the examples have been processed, the final definition is checked against the stored examples. Those examples which are covered are eliminated and those which are not covered are tried again. PAL selects the first uncovered example and the whole process is repeated until all the positive examples have been covered without covering any negative example (see Table 2). In this way, PAL learns disjunctive concepts by learning each clause separately. Alternative methods for handling disjunctive concepts, like storing intermediate hypotheses and allowing some form of backtracking, are left for future research. As the perturbation process cannot guarantee to produce an instance of each particular disjunct, the user must provide at least one example for each disjunct in advance, if he wants to be sure that all the disjuncts will be learned.

4. Learning patterns in chess

In order to learn patterns in chess, PAL was provided with the following pattern–based background vocabulary:

contents(*Side,Piece,Place,Pos*):
Describes the position of each piece.
other_side(*Side1,Side2*):
Side1 is the opponent side of Side2.
sliding_piece(*Piece,Place,Pos*):
Piece is Bishop, Rook or Queen.
in_check(*Side,Place,OPiece,OPlace,Pos*):
King in Place is in check by OPiece in OPlace.
check_mate(*Side,Place,Pos*):
Side with King in Place is check mated.
legal_move(*Side,Piece,Place,NPlace,Pos*):
Piece in Place can move to NPlace.
stale(*Side,Piece,Place,Pos*):
Piece in Place cannot move.
make_move(*Side,Piece,Place,NPlace,Pos1,Pos2*):
Piece in Place moves to NPlace.

The domains of the arguments used to describe chess positions and a representative example for each concept were also given.

```
domain(piece,[pawn,knight,bishop,rook,queen,king]).
domain(side,[black,white]).
domain(place,[square(1,1),square(1,2),...,square(8,8)]).
```

With the above information PAL was used to learn several chess concepts. For instance the following is a specialization of the concept of *fork* learned by PAL. Its interpretation is that a piece (P3) threatens another piece (P2) and checks the King at the same time:

```
fork(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos) ←
    contents(S1,king,(X1,Y1),Pos),
    contents(S1,P2,(X2,Y2),Pos),
    contents(S2,P3,(X3,Y3),Pos),
    other_side(S2,S1),
    in_check(S1,(X1,Y1),P3,(X3,Y3),Pos),
    legal_move(S2,P3,(X3,Y3),(X2,Y2),Pos).
```

An example of a pattern involving a 1–ply movement (*discovery check*) is given below. A check by piece (P2) can be "discovered" after moving another piece (P1) to (X4,Y4).

```
disc_check(S1,P1,(X1,Y1),S1,P2,(X2,Y2),S2,king,(X3,Y3), (X4,Y4),Pos1) ←
    contents(S1,P1,(X1,Y1),Pos1),
    contents(S1,P2,(X2,Y2),Pos1),
    contents(S2,king,(X3,Y3),Pos1),
    other_side(S1,S2),
    sliding_piece(P1,(X1,Y1),Pos1),
    legal_move(S1,P2,(X2,Y2),(X4,Y4),Pos1,Pos2),
    ¬ in_check(S2,(X3,Y3),P1,(X1,Y1),Pos1),
    make_move(S1,P2,(X2,Y2),(X4,Y4),Pos1,Pos2),
    in_check(S2,(X3,Y3),P1,(X1,Y1),Pos2).
```

Table 3 lists some of the concepts learned by PAL. The concepts name and their arity are given on the first column. The number of examples produced by the example generator (second column) is compared with those produced by PAL when additional knowledge about symmetries is taken into account (third column) and with those produced by a user (fourth column). The number of examples that PAL presents to the user can be reduced by taking advantage of the symmetric properties of the board. In particular, some concepts in chess do not depend on any particular orientation of the board, and for each example 7 equivalent examples can be generated, considering reflections along the horizontal, vertical and diagonal axes. PAL can take advantage of this knowledge to produce further generalizations between all the "symmetric" examples before presenting the user with a new example (the exceptions being with concepts involving Pawns or *castlings*). The user can inform the system which axes of symmetry to consider. The minimum number of examples that a trained user (the author) needed to produce the same definitions was also recorded. The additional background knowledge that is used to learn some concepts is given in the last column.

Table 3. Table of results for chess concepts

Concept	Generated Examples	G.E. Symm.	G.E. User	Add. Back. Knowledge
threat/7	23 + 8 –	10 + 7 –	2 +	—
fork/10	22 + 67 –	9 + 16 –	3 +	—
can_check/8	20 + 18 –	2 + 1 –	2 +	—
can_threat/8	23 + 7 –	6 + 9 –	3 +	threat
can_fork/11	34 + 29 –	3 + 0 –	3 +	fork
disc_check/11	17 + 44 –	5 + 6 –	4 +	—
disc_threat/11	26 + 48 –	4 + 2 –	3 +	threat
pin/10	22 + 42 –	4 + 3 –	3 +	threat
skewer/11	19 + 55 –	4 + 22 –	3 +	threat
Average (Tot)	58.22	12.55	2.88	

The number of examples generated by PAL compares very favorably with the size of example space (e.g., the example space for 3 pieces is approximately $\approx 10^8$ examples). Using additional knowledge about the symmetric properties of the board, can reduce the number of examples presented to the user to almost one fifth in average. In general, at least for the chess concepts that we are interested in learning, there is a very small proportion of positive examples in the example space. The greater proportion in Table 2 occurs in a concept like *threat* where it is roughly 1/6 (one positive for every six negative examples). For a concept like *fork* involving three pieces is about 1/(6 * 64).

To test if the patterns learned by PAL could be used for designing a playing strategy, PAL was used to learn patterns for the King and Rook against King endgame. The patterns learned by PAL were used in the designed of a correct playing strategy for this endgame, in the sense that regardless of the movements of the side with only a King, the side with the Rook will always checkmate (see (Morales, 1994) for more details).

4.1. Discussion

Most machine learning approaches have been used in chess. Quinlan used it as a test domain for his ID3 algorithm (Quinlan, 1983). In a domain like chess, a well defined set of attributes is hard to specify even for experts. Quinlan reports 2 man–months work required to define the 3–ply attributes for the King–Rook vs. King–Knight endgame (Quinlan, 1983). Shapiro reports an estimated 6 man weeks effort for the King–Pawn vs. King endgame (Shapiro, 1987). Explanation–based learning (de Jong & Mooney, 1986, Mitchell, Keller & Kedar-Cabelli, 1986) has also been used in this domain (e.g., (Flann & Dietterich, 1989, Tadepalli, 1989, Minton, 1984)). EBL offers the advantage of allowing the system to use as much knowledge as possible from the domain. Its disadvantages include the potentially high cost of generating proofs, the difficulty in having a complete domain theory, and the likelihood that the domain theory will contain errors. Perhaps the most successful EBL/G system for learning concepts in chess has been Flann's Induction–Over–Explanation (IOE) (Flann & Dietterich, 1989). The obvious objection to this approach is

that it must start with a stronger domain theory with at least a general (and very close) definition of the target concept definitions that we want to learn in the first place. Levinson and Snyder (Levinson & Synder, 1991) report a parameter–adjustment system, Morph, which learns weighted patterns, consisting of networks of connections, from traces of games. Morph is limited to learn patterns which express attack/defend relations. For instance, it is unable to learn if two Rooks are in diagonal or if a Rook is in a border. All the patterns in Morph are constructed from a fixed set of relations (links). Once a new pattern is learned, it cannot be used to construct other patterns (i.e., to be used as another link). This however is compensated by a mechanism able to learn from traces of games.

We have shown how PAL can learn several chess concepts from simple example descriptions and general purpose chess knowledge. The patterns learned by PAL can be used for designing playing strategies. In the sections to follow, we will show how the same pattern–based approach can be used in other domains.

5. Learning qualitative models

In a recent paper, Bratko et al. (Bratko, Muggleton & Varsek, 1992) report how Golem (another ILP system), was used to learn a qualitative model of a simple dynamic system (the U–tube). QSIM (Kuipers, 1986), a qualitative formalism used for simulating dynamic models, was taken as the basis for the experiments with Golem. In QSIM, a qualitative simulation of a system starts with a description of the physical parameters (or qualitative variables), a set of constraints describing how those parameters are related to each other and an initial state, and produces possible future states of the system. The constraints are designed to permit a large class of differential equations to be mapped into qualitative constraint equations. In the QSIM formalism, six constraints are allowed: *add* (i.e., $X + Y = Z$), *mult* (i.e., $X \times Y = Z$), *minus* (i.e., $X = -Y$), *m_plus* (i.e., $X = M^{+}(Y)$, that is, X monotonically increases with Y), *m_minus* (i.e., $X = M^{-}(Y)$, X monotonically decreases with Y), and *deriv* (i.e., $dX/dt = Y$). Each qualitative variable has a set of landmark values. The qualitative state of a variable consists of its value or range of values and its direction of change, i.e., *inc* (increasing), *std* (steady) or *dec* (decreasing).

Following (Bratko, Muggleton & Varsek, 1992), the learning task is: given general constraints such as *deriv* or *add* as background knowledge and some qualitative states of a system (examples), induce its model. This can be expressed as follows:

$$\text{QSIM-Theory} \wedge \text{Qual-Model} \vdash \text{Example-Behaviors}$$

The target concept consists of defining a predicate *q_model* in the form:

```
q_model(...) :–
        constraint1(...),
        constraint2(...),
        ...
```

where *constraint*$_i$ is one of *add*, *mult*, *minus*, *m_plus*, *m_minus*, or *deriv*.

5.1. Learning the U-tube

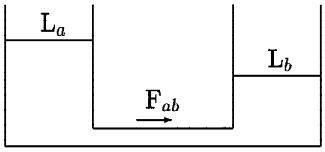

Figure 2. The U-tube

The U–tube (illustrated in Figure 2) consists of two containers, A and B, connected with a pipe and filled with water to their corresponding levels La and Lb. The standard qualitative model used in (Bratko, Muggleton & Varsek, 1992) for the U-tube can be written in Horn clause notation as follows:

```
legalstate(La,Lb,Fab) ←
    add(Lb,Diff,La, [c(lb0,diff0,la0)]),
    m_plus(Diff, Fab, [c(0,0), c(diff0,fab0)]),
    minus(Fab, Fba, [c(fab0,fba0)]),
    deriv(La, Fba),
    deriv(Lb, Fab).                                    (4)
```

where [c(...), c(...), ...] represent a list of corresponding values. In the *add* constraint they say that whenever $Lb = lb0$ and *Diff* (difference between levels) = *diff0*, $La = la0$ (i.e., $lb0$ + *diff0* = $la0$). The *minus* constraint says that the flow from A to B (Fab) is minus the flow from B to A (Fba).

In PAL, the qualitative states (examples) are described by two–place atoms: *qvar*(*Name:Value/Deriv,State*). For instance, a qualitative behavior (at time = t0) can be described as follows:

```
qvar(la:la0/dec,t0).
qvar(lb:lb0/inc,t0).
qvar(fab:fab0/dec,t0).
qvar(fba:fba0/inc,t0).
qvar(diff:diff0/dec,t0).
```

The background knowledge for PAL consisted of definitions for the qualitative constraints: *deriv*, *add*, *minus*, *m_minus*, and *m_plus* (roughly 100 lines of Prolog code). They were taken from the original Prolog code used by Bratko et at. to generate the background facts for Golem[5]. As in Golem, the corresponding values for the constraints were ignored.

A modification was made to the main predicates (i.e., the constraints) to transform them into pattern definitions suitable for PAL. For instance, in the following definition for the constraint *add*, the *qvar*/2 predicates (i.e., the "input" predicates) were added (indicated by the comment "New") to change it into a pattern definition.

```
add(F1:M1/D1,F2:M2/D2,F3:M3/D3,State) :–
    qvar(F1:M1/D1,State), % New
    qvar(F2:M2/D2,State), % New
    qvar(F3:M3/D3,State), % New
    verify_add_inf_consistence(M1, M2, M3),
    verify_add_mag(F1, F2, F3, M1, M2, M3),
    verify_add_der(D1, D2, D3).
```

The same change was made to the other qualitative constraints predicates (i.e., the pattern–based background knowledge), while the rest of the code remained unchanged.

The same positive examples that were provided to Golem were manually given to PAL. With them, PAL obtains the following definition[6]:

```
legalstate(la:A/B,lb:C/D,fab:E/B,fba:F/D,S) :–
    qvar(la:A/B,S),
    qvar(lb:C/D,S),
    qvar(fab:E/B,S),
    qvar(fba:F/D,S),
    deriv(la:A/B,fba:F/D,S),
    deriv(lb:C/D,fab:E/B,S),
    deriv(fab:E/B,fba:F/D,S),
    deriv(fba:F/D,fab:E/B,S),
    m_minus(la:A/B,lb:C/D,S),
    m_minus(la:A/B,fba:F/D,S),
    m_minus(lb:C/D,fab:E/B,S),
    m_minus(fab:E/B,fba:F/D,S),
    m_plus(lb:C/D,fba:F/D,S),
    m_plus(la:A/B,fab:E/B,S),
    minus(fab:E/B,fba:F/D,S),
    add(lb:C/D,fab:E/B,la:A/B,S),
    add(la:A/B,fba:F/D,lb:C/D,S).
```

This model, has the principal components of the model for the U-tube. In general terms, a U-tube model must show that $Fab \propto (La - Lb)$ (shown by the last two *add* literals) and that $dLa/dt = -Fab$ (first *deriv* literal) or that $dLb/dt = Fab$ (second *deriv* literal). The other two *deriv*s (i.e., *deriv*(fba:F/D,fab:E/B,S), and *deriv*(fab:E/B,fba:F/D),S)), say that when the change in Fab is negative then the flow Fba is negative and vice versa. The rest of the constraints follow directly from the physics of the modeled system.

Running PAL with the same examples but only described with the first three variables (i.e., La, Lb, Fab) produces the following definition:

```
legalstate(la:A/B,lb:C/D,fab:E/B,S) :–
    qvar(lb:C/D,S),
    qvar(la:A/B,S),
    qvar(fab:E/B,S),
    deriv(lb:C/D,fab:E/B,S),
    m_minus(la:A/B,lb:C/D,S),
    m_minus(lb:C/D,fab:E/B,S),
    m_plus(la:A/B,fab:E/B,S),
    add(fab:E/B,lb:C/D,la:A/B,S).
```

which again captures the essential features of a model for the U-tube.

5.2. Discussion

The models induced by Golem and PAL were compared against the standard model by evaluating the models on possible states of the U-tube. Golem's model is equivalent to the standard model only in a dynamic sense. From some initial states, the Golem model produces the same behavior as the standard model. The model induced by Golem fails in states where all the water is in one of the containers. It is of interest to note that the models induced by PAL (with three and four variables) accept all the legal states for the U-tube and that PAL can learn a specialization (w.r.t. θ-subsumption) of the standard model if the qualitative variable *diff* is provided (Morales, 1992).

Following (Bratko, Muggleton & Varsek, 1992), other authors have worked on this problem. Although the U-tube looks relatively simple, Bratko et al. (Bratko, Muggleton & Varsek, 1992) and Dzeroski (Dzeroski & Bratko, 1992) report how ILP systems like Foil (Quinlan, 1990) and Linus (Lavrac, Dzeroski & Grobelnik, 1991) are not suited to the task. Each variable has 4 landmark values and 3 time intervals, which gives 7 possible qualitative values for each variable. Combining these with the three possible directions of change give 21 possible qualitative values for each variable. With three variables, the total number of states for the U-tube is 21^3 = 9,261. An additional problem for ILP systems that search heuristically in a general–to–specific way, is that they tend to have problems in learning clauses involving a large number of literals. As in the chess domain, some more interesting qualitative models will tend to have large definitions.

Perhaps the best results for learning qualitative model have been obtained by Coiera with a system called Genmodel (Coiera, 1989). Genmodel is specifically designed for learning qualitative models and has learned, among others, a model for the U-tube. Genmodel has the advantage of using the full power of QSIM by considering the corresponding values to filter out a larger number of constraints. Although PAL was not originately designed to this domain, it is interesting to note that it can achieved an equivalent performance to Genmodel for the U–tube.

We have demonstrated that the learning mechanism employed by PAL generalizes to this qualitative physics domain, and that the results are comparable with those of Golem in terms of quality, and that in some respects PAL's performance is superior.

6. Learning counterpoint rules

The concept of musical counterpoint emerge in the 14th. century and evolve up to *Gradus ad Parnassum* by Johann Joseph Fux published in 1725 (Man, 1971). This is the first book which synthesize in form of rules the art of polyphony considered to be correct by that time. Those rules can be considered as the culmination of musical analysis from the 14th. until 18th. century.

Counterpoint rules can be classified between two voices (sequences of notes) into several species, according to the number of notes involved at the same time on each voice:

- 1st: one note on one voice against one note on the other
- 2nd: two notes on one voice against one note on the other
- 3rd: four notes on one voice against one note on the other
- 4th: a whole note (of four times) on one voice against half notes (of two times) on the other
- 5th or florid: three or more notes in combination with the previous species

Our goal is to obtain similar rules as those described by Fux from examples of counterpoint musical pieces and basic musical knowledge from traditional music. This can be expressed as follows:

$$\text{Music-Theory} \wedge \text{Counterpoint-Rules} \vdash \text{Counterpoint-pieces}$$

Musical knowledge includes the classification of intervals (distances in height between two notes) into: *consonances* and *dissonances*. *Unison*, *fifth* and *octave* are *perfect consonances* while *third* (major and minor) and *sixth* (major and minor) are *imperfect consonances*. *Second* (major and minor), *fourth*, *augmented fourth*, *diminished fifth* and *seventh* (major and minor) are *dissonances*.

These are the elements which account for all harmony in music. The purpose of harmony is to give pleasure by variety of sounds through progressions from one interval to another. Progression is achieved by motion, denoting the distance covered in passing from one interval to another in either direction, up or down. This can occur in three ways: direct, contrary or oblique:

- *direct motion*: results when two or more parts ascend or descend in the same direction
- *contrary motion*: results when one part ascends and the other descends, or vice versa.
- *oblique motion*: results when one part moves while the other remains stationary

With these concepts the counterpoint rules of the 1st. species are defined as follows:

First rule: from one perfect consonance to perfect consonance one must proceed in contrary or oblique motion

Second rule: from a perfect consonance to an imperfect consonance one may proceed in any of the three motions

Third rule: from an imperfect consonance to a perfect consonance one must proceed in contrary or oblique motion

Fourth rule: from one imperfect consonance to another imperfect consonance one may proceed in any of the three motions

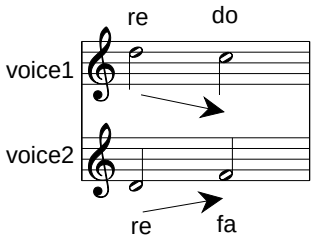

Figure 3. An example of the first rule

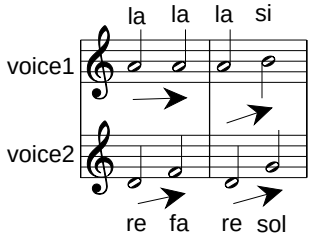

Figure 4. An example of the second rule

In the case of music, a musical score can be completely described by the tone and height of each note involved, its time interval and the voice where it belongs. For counterpoint rules of the first species, time intervals can be ignored[7] and the examples were described by two–place atoms (*note/2*) stating the tone and height of each note and its voice. For instance, *note(c/4, voice1)* states that a "*c*" note in the center of the piano scale (*4*) belongs to voice one. Other notes of the same or different voices can be described in the same way.

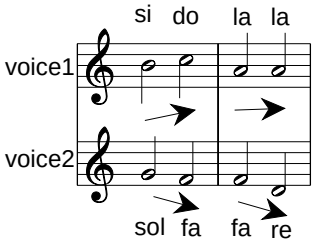

Figure 5. An example of the third rule

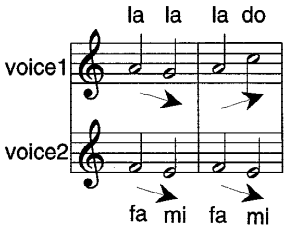

Figure 6. An example of the fourth rule

6.1. Experiments and results

The following musical knowledge was provided to PAL[8]:

- *inter_class1*(*Note1*,*Note2*,*Valid*): describes if two notes from the same voice have a valid/invalid interval. Where valid intervals can be consonances or dissonances which follow the same modality[9] of the *cantus firmus* (sequence of single notes to which counterpoint rules are applied to generate harmonic notes).

- *inter_class2*(*Note1*,*Note2*,*Conso*): describes if two notes of different voices form a perfect or imperfect consonance or a dissonance.

As in the qualitative model domain, the original musical knowledge was adapted to follow a pattern–based formalism. For instance, the definition of *inter_class2/3* was given to PAL as follows:

```
inter_class2(Note1,Note2,Type) ←
        note(Note1/_, Voice1),
        note(Note2/_, Voice2),
        interval(Inter,Note1,Note2),
        int_class(Valid,Inter,Type).
```

where *interval/3* and *int_class/3* are background knowledge definitions that return the musical interval between two notes, and if the interval is valid/invalid with its type (*perf_cons, imperf_cons, diss*).

PAL was given manually the examples for each rule[10]. The number of examples required to learn each rule is given below:

Rules	Rule1	Rule2	Rule3	Rule4
Number of examples	6	4	5	5

The first rule induced by PAL is shown below (the other three rules are very similar changing only in the different combinations of *perf_cons* and *imperf_cons*).

```
rule(Note1/Height1, Note2/Height1, voice1, Note3/Height2,
     Note4/Height2, voice2)  :–
          note(Note1/Height1, voice1),
          note(Note2/Height1, voice1),
          note(Note3/Height2, voice2),
          note(Note4/Height2, voice2),
          inter_class1(Note1, Note2, valid),
          inter_class1(Note3, Note4, valid),
          inter_class2(Note1, Note3, perf_cons),
          inter_class2(Note2, Note4, perf_cons).
```

The rules learned by PAL were tested for analysis on simple counterpoint pieces. We add an extra argument to each rule to distinguished them from the rest. The analysis was made with the following program:

```
analysis([N1,N2|RVoice1],[N3,N4|RVoice2],[NumRule|Rules]) ←
     rule(N1, N2, voice1, N3, N4, voice2, NumRule),
     analysis([N2|RVoice1],[N4|RVoice2],Rules).
analysis([ _ ],[ _ ],[ ]).
```

For instance, for the piece below (figure 7), we obtained the following analysis:

```
?- analysis([d/4,f/4,e/4,d/4,g/4,f/4,a/4,g/4,f/4,e/4,d/4],
            [a/4,a/4,g/4,a/4,b/4,c/5,c/5,b/4,d/5,cs/5,d/5],
            Rules).

  Rules = [r1,r4,r3,r2,r3,r2,r4,r4,r4,r2].
```

The same program can be used for musical generation. For example in figure 8, given the *cantus firmus*(the first voice), we can generate the required counterpoint notes (the second voice):

```
?- analysis(Notes,[d/4,f/4,e/4,d/4,g/4,a/4,g/4,f/4,e/4,d/4],[r1,r3,r3,...]).

  Notes = [d/3,d/3,a/3,f/3,e/3,d/3,f/3,c/4,d/4,cs/4,d/4].
```

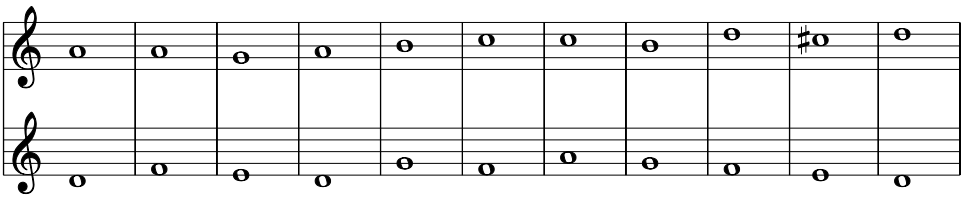

Figure 7. An example of counterpoint analysis

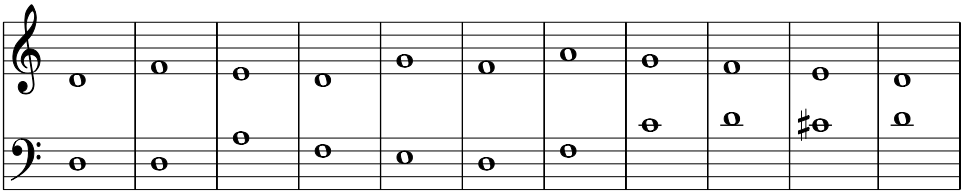

Figure 8. An example of counterpoint generation

6.2. Discussion

In (Widmer, 1992), Widmer describes a system capable of learning counterpoint rules using an Explanation–based learning approach (de Jong & Mooney, 1986). Unlike PAL, a generalization of the target counterpoint rules is required as background knowledge, from which the more specific counterpoint rules are derived. By contrast, PAL uses a much simpler background knowledge to induce equivalent rules.

It is shown in this section how PAL can effectively learn simple counterpoint rules from general purpose musical knowledge and simple example descriptions. The learned rules can be used for musical analysis and generation.

7. Comparison with other related work

To achieve efficiency some ILP systems (such as Golem(Muggleton & Feng, 1990) or Foil (Quinlan, 1990)) have used a set of ground atoms as background knowledge. These systems, however, suffer from the problem of preparing such background facts. Defining background facts for some concepts is a time consuming and difficult process. Specially since it is sometimes unworkable for the systems to include all the background facts, even if they are finite (e.g., the concept of legal moves in chess, or the concept of addition of qualitative variables). In such cases, appropriate subsets need to be selected to maintain efficiency, which requires a prior knowledge of the training example set. In particular, Golem required to simplified the definition of *add* into three more "economical" predicates to reduce the number of tabulated facts and learned a model for the U–tube[11]. These systems cannot be

used in an incremental way, which means that once a concept is learned, ground facts need to be selected again if it is going to be used in the induction of a new concept.

ILP systems which use non–ground clauses as background knowledge, such as ITOU (Rouveurol, 1991) or Clint (De Raedt & Bruynooghe, 1988, De Raedt & Bruynooghe), have been applied to very limited domains (e.g., family relations, the concept of "arch", etc) despite the use of several constraints. In particular, CLINT can use integrity constraints to specify properties about the target concept and reduce the search space. ITOU uses partial graphs to represent saturated examples and guide the generalization process.

There have been other recent systems that impose a particular structure to the learned clauses. For instance, GRENDEL (Cohen, 1992) limits the hypothesis space by requiring all the hypotheses to be sentences of a particular grammar called Antecedent Description Grammar. GRENDEL starts with the start symbol of the grammar and repeatedly specializes it by applying rewrite rules of the grammar and using Foil's information–gain metric to guide this process.

Progol (Muggleton, 1995) is a most recent ILP system which uses mode declaration and depth–bounded resolution steps to construct a most–specific clause. It then uses a refinement operator, similar to MIS (Shapiro, 1983), to construct its hypotheses considering the predicates used in the most–specific clause to focus its search.

PAL can learn relatively long clauses in the presence of large amounts of background knowledge[12] which are difficult to learn by systems which heuristically search in a general to specific way. Additionally, PAL's learning task is non standard, in the sense that the name and the number of arguments of the target concept are not explicitly stated in the examples. The introduction of a pattern–based background knowledge representation and the introduction of a novel constraint based on labels have allow PAL to learn concepts which are beyond the existing capabilities of current inductive systems, despite its simplicity. On the other hand, PAL is unable to learn concepts which do not follow a pattern–based format.

8. Conclusions and future work

Pattern–based reasoning has been used by several computer systems to guide their reasoning strategies[13]. For chess, in particular, a pattern–based approach has been used with relative success in simple end–games (Bramer, 1977, Bratko, 1982, Huberman, 1968) and tactically sharp middle games (Berliner, 1977, Pitrat, 1977, Wilkins, 1979). Our aim has been to learn chess patterns from simple example descriptions together with the rules of the game. To achieve this goal, we have used an Inductive Logic Programming (ILP) framework as it provides an adequate hypothesis language and mechanism where patterns and examples can be simply expressed and, in principle, learned in the presence of background knowledge. The limitations of current ILP systems are more clearly exposed in domains like chess, where a large number of background definitions can be required, it is common to have non–deterministic concepts, some background knowledge definitions can cover a very large number of facts all of which are required, and concepts can be several literals long. The same restrictions appear in other domains, like qualitative modeling. In PAL, examples are given as descriptions of states of a system (e.g., a description of the pieces in a chess board) and instances of patterns definitions are derived from such descriptions. Together they are

used to construct new pattern definitions, which can then be used in new examples. In order to 'recognize' instances of patterns from such state descriptions, we have introduced a pattern–based knowledge representation. This approach makes a more selective use of the background knowledge by considering only those definitions which apply to the current example description, reducing most of the problems encountered by other ILP systems. The approach has been successfully applied in chess, qualitative reasoning, and music, and it is suspected to be useful to other domains where a pattern–based approach can be applied. A novel mechanism for labeling the arguments of the components used in the state descriptions and the atoms which are derived from the background knowledge have been used for two main purposes. On one hand it guides and constraints the generalization process as only compatible literals of the same components are considered. On the other hand, it is used to construct the example space and guide the example generator by indicating which literals are affected by which arguments.

8.1. Future work

There are several research areas which are being considered at the moment. In particular, we would like to extend the expressiveness of our pattern definitions (e.g., allow recursive patterns). We have not dealt with noisy examples and/or incomplete knowledge. One important question is how much starting knowledge to include in order to learn a wide range of concepts. In the presence of noise in the training examples, the generated hypotheses must be allowed to accept some negative examples and/or to reject some positive examples. If both noise and incomplete knowledge are considered, a balance between them must be established. That is, recognize when the induced concept is incorrect due to noisy information or because the lack of knowledge.

The *lgg* of clauses is limited to learning single clauses (i.e., it cannot learn disjunctive definitions), cannot include negation of literals, and cannot introduce new terms. PAL can learn disjunctive definitions using negative examples to check for over–generalizations and by providing an instance of each disjunct. With insufficient or unrepresentative negative examples, PAL can produce over–generalizations and does not have an adequate mechanism to recover from them. Similarly, it cannot detect if insufficient knowledge is provided.

The strategy followed by PAL has been to do 'simple' perturbations first. Additional improvements in the learning rate can be obtained by including domain dependent knowledge to the example generator (e.g., symmetries, shifts of positions, etc.). Although not implemented, the invariance of a clause with respect to an axis of symmetry or to shifted positions could be deduced from the concept definitions after seeing several examples.

Acknowledgments

This research was partly supported by a grant from CONACyT (México). I would like to thank Tim Niblett for all his insightful comments in the development of this work, Roberto Morales who provided me with all the musical knowledge, and the anonymous reviewers for their helpfull comments on an earlier version of this paper.

Notes

1. Clauses whose literals are not completely determined by instantiations of the head.
2. A single white or black piece is legally moved from one square to another.
3. The complement of P is $\neg P$ and vice versa.
4. Background definitions given as ground unit clauses are also considered as relevant.
5. I am grateful to Saso Dzeroski for providing me with the code.
6. Redundant literals produced by symmetry and associativity of the constraints (e.g., *m_plus*(A, B) and *m_plus*(B, A), and *add*(A, B, C) and *add*(B, A, C)), were removed from the definition.
7. We are beginning to investigate how to include time intervals in the descriptions of scores.
8. I am grateful to Roberto Morales who provided all the musical background knowledge.
9. The same musical scale.
10. The examples were suggested by Roberto Morales without knowing the exact functioning of the system.
11. For this domain, the *add* constraint requires several thousand facts for each triple of variables.
12. Both the chess and the qualitative model domains include several pages of Prolog code as background knowledge.
13. In particular, the conditions used in the production rules used in most expert systems can be regarded as particular patterns to match.

References

Berliner, H.J. (1977). "A representation and some mechanisms for a problem-solving chess program". In M.R.B. Clarke (Eds.), *Advances in Computer Chess 1*, Edinburgh: Edinburgh University Press.

Bramer, M. A. (1977). "Representation of knowledge for chess endgames: Towards a self–improving system". *PhD Thesis* Open University, Milton Keynes.

Bratko, I. (1982). "Knowledge-based problem-solving in AL3". In J.E. Hayes, D. Michie, & Y.H. Pao (Eds.), *Machine Intelligence 10*, Horwood.

Bratko, I., Muggleton, S., & Varsek, A. (1992). "Learning qualitative models of dynamic systems". In S. Muggleton (Eds.), *Inductive Logic Programming*, London: Academic Press.

Buntine, W. (1988). "Generalized subsumption and its applications to induction and redundancy", *Artificial Intelligence*, 36(2), 149–176.

Campbell, D. T. (1966). *Pattern Matching as an Essential in Distal Knowing*, New York: Holt, Rinehart and Winston.

Charness, N. (1977). "Chess skill in man and machine", In P.W. Frey (Eds.), *Human chess skill*, Springer-Verlag.

Cohen, W. (1992) "Compiling prior knowledge into an explicit bias", *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 485–489). San Mateo: CA: Morgan Kaufman.

Coiera, E. (1989) "Generating qualitative models from example behaviors", *DCS Report No. 8901*, School of Electrical Engineering and Computer Science, University of New South Wales, Sydney, Australia.

de Groot, A. (1965). *Thought and Choice in Chess*, The Hague: Mouton.

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

de Raedt, L. & Bruynooghe, M. (1988). "On interactive concept-learning and assimilation", *Proceedings of the Third European Working Session on Learning*, (167–176), London: Pitman.

de Raedt, L. & Bruynooghe, M. (1990). "Indirect relevance and bias in inductive concept-learning", *Knowledge Acquisition*, 2(4), 365–390.

Dzeroski, S. & Bratko, I. (1992). "Handling noise in inductive logic programming", *Proceedings of the Seciond International Workshop on Inductive Progamming*, (pp. XX-YY), Tokyo, Japan, ICOT TM-1181, Institute for New Generation Computer Technology.

Flann, N. S. & Dietterich, T. G. (1989). "A study of explanation-based methods for inductive learning", *Machine Learning*, 4(2), 187–226.

Huberman, B. J. (1968). "A program to play chess end games", *CS-106*, Computer Science Department, Stanford University, Stanford, CA.

Kuipers, B. (1986). "Qualitative simulation", *Artificial Intelligence*, 29 (3), 289–338.

Lavrac, N., Dzeroski, S., & Grobelnik, M. (1991). "Learning nonrecursive definitions of relations with linus", *Proceedings of the European Working Session on Learning*, (265–281), Berlin: Springer-Verlag.

Levinson, R. & Snyder, R. (1991) "Adaptive Pattern-Oriented Chess", *Proceedings of the Ninth National Conference on Artifical Intelligence*, (601–606), Boston: AAAI Press - The MIT Press,

Lorenz, K. (1973) *Behind the Mirror*, New York: Harcourt Brace Jovanovich.

Man, A. (1971) *The study of counterpoint from Johann Joseph Fux's Gradus ad Parnassum*, W.W. Norton & Company.

Minton, S. (1984). "Constraint–based generalization: learning game–playing plans from single examples", *Proceedings of the National Conference on Artificial Intelligence*, (251–254), Menlo Park, CA: Kaufmann.

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

Morales, E. (1992) "First order induction of patterns in Chess", *PhD Thesis*, The Turing Institute - University of Strathclyde, Glasgow.

Morales, E. (1994). "Learning patterns for playing strategies", *ICCA Journal*, 17(1), 15–26.

Muggleton, S. (1995) *New Generation Computing Journal* 13: 245–286.

Muggleton, S. (1992) *Inductive Logic Programming*, London: Academic Press.

Muggleton, S. & Feng, C. (1990). "Efficient Induction of Logic Programs", In S. Muggleton (Eds.), *Inductive Logic Programming*, London: Academic Press.

Niblett, T. (1988). "A study of generalisation in logic programs", *Proceedings of the Third European Working Session on Learning*, (131–138), London: Pitman.

Pitrat, J. (1977). "A chess combination program which uses plans", *Artificial Intelligence*, 8, 275–321.

Plotkin, G. D. (1969). "A note on inductive generalization", In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5*, Edinburgh: Edinburgh University Press.

Plotkin, G. D. (1971a). "A further note of inductive generalization", In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 6*, Edinburgh: Edinburgh University Press.

Plotkin, G. D. (1971b). "Automatic methods of inductive inference", *PhD Thesis*, University of Edinburgh, Edinburgh.

Quinlan, J. R. (1983). "Learning efficient classification procedures and their application to chess end games", In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine Learning: an artificial intelligence approach*, Palo Alto, CA: Tioga.

Quinlan, J. R. (1990). "Learning logical definitions from relations", *Machine Learning*, 5(3), 239–266.

Rouveurol, C. (1991) "ITOU: Induction of first order theories", In S. Muggleton (Eds.), *Proceedings of the International Workshop of Logic Programming*, Viana de Castelo, Portugal.

Shapiro, E.Y. (1983) *Algorithmic program debugging*. MIT Press.

Shapiro, A. D. (1987) *Structured induction in expert systems*, Wokingham: Turing Institute Press in association with Addison-Wesley.

Tadepalli, P. (1989) "Planning in games using approximately learned macros", In B. Spatz (Eds.), *Proceedings of the Sixth International Workshop on Machine Learning*, (221–223), San Mateo, CA: Morgan Kaufmann.

Widmer, G. (1992). "The importance of basic musical knowledge for effective learning", In M. Balaban, J. Ebcioğlu & O. Laske (Eds.), *Understandig Musica with AI: Perspectives on Music Cognition*, Cambridge - Menlo Park: AAAI Press/MIT Press.

Wilkins, D. E. (1979) "Using patterns and plans to solve problems and control search", *(AIM-329 ; STAN-CS-79-747)*, Stanford University, Artificial Intelligence Laboratory, Stanford, CA.

Received September 1995
Accepted June 1996
Final Manuscript January 1997