Learning Features by Experimentation in Chess

Eduardo Morales
The Turing Institute, 36 North Hanover St., Glasgow G1 2AD
Email: eduardo@turing.ac.uk

Abstract

There are two main issues to consider in an inductive learning system. These are 1) its search through the hypothesis space and 2) the amount of provided information for the system to work. In this paper we use a constrained relative least-general-generalisation (RLGG) algorithm as method of generalisation to organise the search space and an automatic example generator to reduce the user's intervention and guide the learning process. Some initial results to learn a restricted form of Horn clause concepts in chess are presented. The main limitations of the learning system and the example generator are pointed out and conclusions and future research directions indicated.

Keywords: LGG, experimentation, chess, Horn clause

1 Introduction

Suppose we want a system to learn the definition of the concept of a piece threatening another piece in chess, neither of which is a king. We provide the system with a description of a position where a piece is threatening another one, but we do not tell the system what concept we want to learn or which arguments are involved in the new concept.

The position of Figure 1 can be completely described by a three-place predicate (contents/3) stating the position of each piece in the board.

contents(black,king,square(1,8)).
contents(black,rook,square(4,4)).
contents(white,king,square(1,1)).
contents(white,pawn,square(4,7)).

The system uses the above description with its current background knowledge to recognise a set of "features" and construct a possible definition. If the background vocabulary of the system consists of:

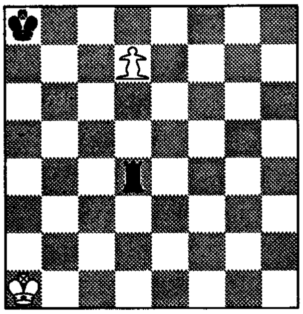


Figure 1: Example Position

contents(Side,Piece,Place).	Describes the position of each piece.
sliding_piece(Piece).	Piece is a rook, bishop or queen.
straight_slide(Piece).	Piece is a rook or queen.
all_but_K(Piece).	Piece is anything but king.
other_side(Side1,Side2).	Side1 is the other side of Side2
legal_move(Side,Piece,Place,NewPlace).	Piece of Side in Place can move to NewPlace.

Then the system produces the following highly specialised definition for that particular example:

```
concept(black,king,square(1,8),black,rook,square(4,4),
        white,king,square(1,1),white,pawn,square(4,7)) ←
            contents(black,king,square(1,8)),
            ...
            other_side(black,white),
            other_side(white,black),
            all_but_K(pawn),
            all_but_K(rook),
            straight_slide(rook),
            sliding_piece(rook),
            legal_move(black,king,square(1,8),square(2,8)),
            ...
            legal_move(black,rook,square(4,4),square(1,4)),
            ...
            legal_move(white,king,square(1,1),square(2,1)),
            ...
            legal_move(white,pawn,square(4,7),square(4,8)).
```

The system then follows an experimentation process by automatically generating positive and negative examples (validated by the user) from which other features are deduced and similar definitions constructed. Following a generalisation process between definitions, eventually, the system recognises that the two kings are irrelevant to the concept and arrives to the following definition:

```
concept(A,B,square(C,D),E,F,square(G,H)) ←
        contents(A,B,square(C,D)),
        contents(E,F,square(G,H)),
        other_side(E,A),
        other_side(A,E),
        all_but_K(B),
        all_but_K(F),
        legal_move(A,B,square(C,D),square(G,H)).
```

We have built a system which arrives to the same definition after generating 21 positive and 11 negative examples. It has been able to learn a more general definition of threat, forks, possible attacks and possible checks in chess. It learns concepts expressed in a subset of Horn clauses after generating a small number of examples.

Section 2 describes the generalisation method based on an RLGG algorithm. Section 3 discusses the automatic example generator method based on "perturbations". The learning algorithm is summarised in Section 4 and some examples in chess are presented in Section 5. Finally, Section 6, summarises and suggests future research directions.

2 Constrained RLGG

2.1 Introduction

Due to the requirements of searching a large hypothesis space, systems that induce first-order predicates have been of limited success since they have been forced to constrain their search space in such a way that only simple concepts can be learned. More recently, a model of generalisation based on relative least-general-generalisation (RLGG) [Plotkin, 1971a] has been used successfully to learn new concepts using a Horn clause framework ([Muggleton & Cao, 1990]).[1]

Plotkin [Plotkin, 1971b, Plotkin, 1969] describes how to construct the least general generalisation (LGG) of two clauses in terms of Θ-subsumption. Clause C_1 is more general than clause C_2 if C_1 Θ-subsumes C_2 (i.e., $C_1\sigma \subseteq C_2$ for some substitution σ). The least general generalisation of two clauses is a generalisation which is less general than any other generalisation. The LGG of two

[1]Muggleton [Muggleton, 1990] provides a unified framework for his Inverse Resolution method [Muggleton & Buntine, 1988] and Plotkin's RLGG.

clauses C_1 and C_2 is defined as: $\{l : l_1 \in C_1 \text{ and } l_2 \in C_2 \text{ and } l = LGG(l_1, l_2)\}$. The LGG of two terms or literals is defined for two terms or literals with the same predicate name and sign (compatible). The algorithm proceeds as follows: If L_1 and L_2 are compatible, then find terms t_1 and t_2 that have the same place in L_1 and L_2 such that $t_1 \neq t_2$ and both t_1 and t_2 either begin with different function letters or at least one of them is a variable. If there is no such pair t_1, t_2, then finish. Else replace t_1 and t_2 by a new variable V and, whenever t_1 and t_2 appear in L_1 and L_2 in the same place, replace them by V.

Plotkin [Plotkin, 1971a, Plotkin, 1971b] also introduces a notion of LGG relative to some background knowledge $\mathcal{K}$. Given $\mathcal{K}$, two examples e_1 and e_2 for which $\mathcal{K} \not\vdash e_1$ and $\mathcal{K} \not\vdash e_2$. C is the LGG of e_1 and e_2 relative to $\mathcal{K}$ whenever C is the least general clause for which $\mathcal{K} \wedge C \vdash e_1 \wedge e_2$. We can construct C by replacing $\mathcal{K}$ with a set of ground atoms $a_1 \wedge a_2 \wedge \ldots$, representing a model of $\mathcal{K}$ (see also [Buntine, 1988, Muggleton & Cao, 1990]), and taking the LGG (as described above) of two clauses C_1 and C_2 defined as:

$$C_1 = (\overline{a_1} \vee \overline{a_2} \vee \ldots) \vee e_1$$

$$C_2 = (\overline{a_1} \vee \overline{a_2} \vee \ldots) \vee e_2$$

2.2 Constraints on the Background Knowledge

In general, if e_1 and e_2 are unit clauses and only a finite number of ground atoms (constructed with symbols in $\mathcal{K}$, e_1 and e_2) are logical consequences of $\mathcal{K}$, then the LGG of e_1 and e_2 relative to $\mathcal{K}$ exists. A key issue in RLGG is how to choose adequate constraints to produce a finite set of "relevant" atoms derived from $\mathcal{K}$. Buntine [Buntine, 1988] suggests using a finite subset of the least Herbrand model of $\mathcal{K}$. Muggleton and Feng [Muggleton & Cao, 1990] substitute $\mathcal{K}$ by an h-easy model constructed from a restricted form of Horn clauses. Rather than generating and storing a large number of relevant atoms, we use a restricted form of Horn clauses, supported by a variable-typed logic, from which only a finite number of ground atoms can be derived.[2]

2.3 Knowledge Representation

Our final research direction aims to use a learning strategy in conjunction with a planning system to deal with reactive environments such as chess [Morales, 1990]. We assume that the planning skills of a chess player are linked to the number of "features" he/she can recognise from a chess position and that their skills can improve when learning to recognise new features. With this aim in mind, our research is oriented towards learning new feature definitions from existing ones. The learning algorithm relies on an oracle which provides an initial example description and classifies

[2]Our clauses are more restricted than Muggleton and Feng's [Muggleton & Cao, 1990].

the examples generated by the perturbation algorithm (described in Section 3). Depending on the initial background knowledge and on the particular example description, the system is able to derive (recognise) more or less atoms (features). We propose to start with some basic background knowledge and incrementally extend the domain knowledge by learning "simple" concepts first.

Unlike other systems, the relevant arguments of the target concept do not need to be pre-defined. We define a feature as an atom which is true for the current board position description. A board position or example description is specified by a set of ground unit clauses. A feature definition is a restricted Horn clause which takes the example description to test for particular features. A feature definition has the following format:

$$H \leftarrow D_1, D_2, ..., D_n, F_1, F_2, ..., F_n.$$

where D_is are "input" predicates used to describe positions and F_is are "feature" predicates which are either provided as background knowledge or learned by the system. We define as input predicates those which are used to describe the current example but which depend on at most one piece. In the example of Section 1, all the predicates except `legal_move` are input predicates. Feature predicates are dependent on the position of other pieces or provided as background knowledge. For example, legal moves, checks, check mates, ..., etc.

This format instantiates the arguments required by the clause with arguments of the current example description constraining the possible instantiations of the head and producing only relevant atoms to the current example.

For example, the following feature definition is used to recognise checks in chess. The input predicates are contents/3 and other_side/2 and the feature predicate is piece_move/4.

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

The example description is included to the theory and a set of relevant atoms are derived from the feature definitions (representing a model). These atoms constitute a feasible body of the new concept definition. Since we do not specify exactly which arguments are involved in the concept definition, a "tentative" head is constructed with the arguments used in the "input" predicates. This initial clause is gradually generalised using an LGG algorithm between this clause and similar clauses constructed from other example descriptions generated by the perturbation algorithm (Appendix 1 has a complete sequence of gradual generalisations produced when learning a special case of the concept of fork in chess).

```
Given a new example description (ground input predicates),
and a set of feature definitions (representing the domain theory)
Add the description to the domain theory and
Construct a new clause
        The body being the set of atoms derived from the feature definitions
        with the input predicates
        The head constructed with the arguments used in the input predicates
Make an LGG between this clause and the current concept clause
          the resulting clause being of the form $H \leftarrow D_1, D_2, ..., D_n, F_1, F_2, ..., F_n$.
          where $D_i$s are input predicates
Remove the arguments in the head that do not appear in any $F_i$
Remove any literal with a variable argument which do not appear in any
other place of the concept definition
```

Table 1: Generalisation Algorithm

Once a generalisation is produced, the system tries to reduce the number of arguments involved in the head of the new concept by keeping only those which appear in a literal (different than the input predicates) in the new concept definition. New compatible heads are constructed taking into account the current concept head.

Even if we produce a finite set of atoms to construct a clause, the RLGG algorithm can generate clauses with a large number of literals. The length of the clauses is constrained by deleting all the literals whose variable arguments do not appear on any other place in the concept definition (see Table 1).

The constrained RLGG algorithm has been able to learn concept like forks and attacks in chess. The knowledge representation syntax, which follows our intuitive definition of a feature in chess, can produce only a finite set of relevant atoms for an example description.

3 Perturbation Method

3.1 Introduction

One key issue to consider is the information on which the system relies for its "correct" behaviour. In some cases, the learning process is highly dependent on the user's intervention. This is more noticeable in an incremental learning system, where the user often has to be careful in selecting the examples or training instances to ensure that the system will succeed on its learning task (e.g. [Winston, 1977, Sammut & Banerji, 1986]). This dependency or *hidden knowledge*, requires a good understanding of the system's internal characteristics and severely questions the system's learning capabilities. Experimentation (or active instance selection) has been employed in several machine learning systems [Feng, 1990, Carbonell & Gil, 1987, Dietterich & Buchanan, 1983, Lenat, 1976, Porter & Kibler, 1986] to reduce this dependency and guide the learning process.

There are several strategies that can be adopted to generate an example. Ruff and Dietterich [Ruff & Dietterich, 1989] argue that there is no essential difference between an example generator that uses a "clever" (although computationally expensive) strategy to divide the hypothesis space and a simple example generator that randomly selects examples. As we construct clauses from positive examples only, a random strategy is of very little use, especially when the target concept covers a small part of the example space, as it can generate a huge number of negative examples before generating a positive, slowing down the learning process. Another alternative is to provide a hierarchy of concepts and generate new examples from instances of concepts higher or at the same level of the hierarchy [Porter & Kibler, 1986, Lenat, 1976]. While applicable in some domains, some others domains are not so easily structured and alternative methods must be employed. Feng [Feng, 1990] provides the theoretical basis for choosing a new example based on information theory. His next-best-verification algorithm chooses the next example which is the best to verify a hypothesis based on information content. In practice, he requires a set of heuristics to define a sequential number for the examples (the best example being the one which follows in the sequence), which in general is not easy to do as several "sequences" along different "dimensions" can exist.

3.2 A Framework for Describing the Example Space

In an automatic example generator, the space of examples depends on the number of arguments required to describe an instance of the target concept and on the size of their domains. If an example can be described by instantiating N arguments, we can have 2^N - 1 different perturbation classes distributed in N perturbation levels. Each perturbation level represents the number of arguments to change at the same time to generate a new example and each perturbation class shows the particular arguments to change, representing a class of instances. For example, if we can describe an instance of the concept of threat between two pieces with four arguments, e.g., threat(P1,L1,P2,L2) (meaning that piece P1 in place L1 threatens piece P2 in position L2), we can structure the perturbation space in four levels (see Figure 2).

At each perturbation class, we can generate $D_i \times D_j \times \ldots \times D_n$ examples, where each D_k is a particular argument domain at that level. For instance, the perturbation class `[L1,P2]` represents the class of examples that can be generated by changing the position of the attacking piece and the piece which is being threatened. In the example of Section 1, the attacking piece (rook) can be in 60 different legal positions and the piece being attacked can be changed for knight, bishop, rook or

```
4                P1,L1,P2,L2

3     P1,L1,P2   P1,L1,L2   P1,P2,L2   L1,P2,L2

2   P1,L1   P1,P2   P1,L2   L1,P2   L1,L2   P2,L2

1                P1   L1   P2   L2
```

Figure 2: Pertubation Space

queen. Clearly the example space grows exponentially with the number of arguments involved.

We can apply several domain constraints to reduce this space. In particular, not all the perturbations generate legal examples. For instance, the first (last) rank can be eliminated from the domain of the positions of the white (black) pawns, we can use the knowledge that two and only two kings (one on each side) must be at any position to constrain the domains on the possible values for the pieces and avoid changing sides on one king without changing in the other, ..., etc. We can choose a particular order in which to traverse this space. Like changing all the arguments that involve the sides of the pieces first (this corresponds to a particular perturbation class). Similarly, in other domains like the 8-puzzle, we can constrain the perturbation space to perturbation classes that involve only "swapping" tiles. Despite these constraints, the example space can still be huge (e.g., in chess, two kings alone can be in 3612 different legal positions, which corresponds to a perturbation class at level 2).

3.3 A Perturbation Algorithm

Our example generation strategy is guided by the current concept definition, starting the perturbation process at the lower levels of the previously described structure and moving gradually upwards trying to reduce the arguments and their domains in the process. The perturbation classes are generated dynamically, i.e., we do not produce a new perturbation class unless required. After exploring a perturbation class (this is described below), only its immediate perturbation classes above are generated. Similarly, if an argument is eliminated from the head of the concept definition (as described in Section 2) or when its domain is reduced to the empty list, all the perturbation classes where the argument appears are removed from the perturbation space.

For example, following the hypothetical concept of a threat with 4 arguments, the perturbation space will initially consist of 4 perturbation classes:

```
[[P1], [L1], [P2], [L2]]
```

As soon as we finish exploring the possible places of the attacking piece [P1], the new perturbation space becomes:

```
[[L1], [P2], [L2]], [[P1,L1], [P1,P2], [P1,L2]]
```

Similarly, after exploring the possible attacking pieces [P1], we have:

```
[[P2], [L2]], [[P1,L1], [P1,P2], [P1,L2], [L1,P2], [L1,L2]].
```

If the attacked piece "P2" is removed from the head of the concept or if its domain becomes empty, then the new perturbation space becomes:

```
[[L2]], [[P1,L1], [P1,L2], [L1,L2]]
```

Recognising irrelevant arguments is important since they represent significant cuts in the search space.[3]

The perturbation algorithm picks the first perturbation class and generates new examples by picking new values from the domains of the arguments involved. If no new values can be generated (i.e., it has finished exploring a perturbation class), it changes the perturbation space (as described above), otherwise it checks which literals (features) from the concept definition fail with the new values. After selecting new values, if none of the literals fail, it considers those values as irrelevant and removes them from the domain. If at least one literal fails, it constructs a new example with the new values. When a negative example is generated, the system tries to construct an example that will succeed on an least one of the literals that failed on that example. Whenever an argument is eliminated from the head of the concept definition or if its domain becomes empty, it is removed from the perturbation space. The perturbation process ends when there are no more perturbation classes left (see Table 2).

Following the example given in Section 1, at the first level of perturbation, a new example can be generated by replacing the attacked piece (pawn) with a knight. This perturbation fails the literal: legal_move(white,knight,square(4,7),square(4,8)) and a new clause is constructed. However, if the attacker (rook) is replaced with a knight, generating a negative example, then the system will try to construct an example that will succeed on at least one of the failed features (e.g., replacing the attacker with a queen).

The perturbation method has been used to guide the learning process of the RLGG algorithm described in Section 2. In general, this strategy will converge faster to a concept definition than a random example generator, especially in concepts where a small number of positive examples exist in a large example space. It has a clear termination criterion to stop the generation of examples and produces a smaller set of examples because it can reduce the example space during the learning process.

[3]An argument which is removed from the perturbation space is not necessarily removed from the definition.

```
DO UNTIL all the perturbation classes has been explored
    or stopped by the user
    IF a new definition is constructed
    THEN pick the first perturbation class and try to generate an
        example that will fail on at least one of its literals
    IF a negative example is generated
    THEN see which literals failed with that example and try to
        generate an example which that succeed on at least one
        of them
    IF we cannot generate a new example that will fail
        (or succeed) on any literal,
    THEN generate the next perturbation classes and continue
END DO
```

Table 2: Perturbation Algorithm

4 The Learning Algorithm

We can now summarise the learning method using the description of the previous two sections. Initially, the system is provided with some background knowledge, the domain values of the arguments involved to describe an example, and a description of a "typical" example of the target concept. The system first constructs an initial concept definition and an initial perturbation level. The system then calls the example generator method to create new examples (see Section 3). Each time a positive example is created the system uses the constrained RLGG algorithm (see Section 2) to create a new concept definition. The example generator tries to fail on at least one of the concept literals by changing (perturbing) the arguments involved in the current perturbation class. 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 the system cannot generate a new example (i.e., a new generalisation of the current definition will require producing an example that involves changing different arguments), then it changes the perturbation space and continues. The process ends when there are no more levels left, or when the user decides to terminate it.

Each new definition is checked against the current negative examples (the user is also asked for confirmation). This is to avoid over-generalisations, which can occur when learning disjunctive definitions. If a definition covers a negative example, then it is rejected and the example is stored. When the perturbation process finishes, the final definition is checked against the stored examples,

```
Given an initial example description
Construct an initial clause (as described in Section 2) and
an initial perturbation level (as described in Section 3)
DO UNTIL no more perturbation levels or stopped by the user
    CALL PERTURBATION-METHOD to generate a new example
    IF the example is positive
    THEN CALL RLGG
            IF the new definition covers a negative example
               (or if it is rejected by the user)
            THEN reject the definition and store that example
END DO
Check the final definition with the stored examples
IF some examples are not covered,
THEN start again
ELSE add the new definition to the background knowledge
and finish
```

Table 3: Learning Algorithm

those which cannot succeed are tried again and the whole process is repeated. In this way the systems is able to learn disjunctive concepts although each clause is learned separately (see Table 3).

5 Examples

We applied the previously described system to learn some concepts in chess. We provide the system with the same background knowledge described in the introduction. The input predicates being for each example, contents/3, straight_slide/1, sliding_piece/1, all_but_K/1 and other_side/2. Feature predicates definitions to recognise legal moves, checks and check-mates were also given. We provided as well domain values for the arguments used in the input predicates (i.e., Side, Piece, Place),

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

and specification of which arguments have which domain.

This time, we decided to broaden the concept of a threat and accept as positive examples those which include as well a king threatening a piece. Using the same description of the initial example given in the introduction, the system produces the following definition (which follows our more general definition) after generating 48 positive and 14 negative examples:

```
threat(A,B,square(C,D),E,F,square(G,H)) ←
      contents(A,B,square(C,D)),
      contents(E,F,square(G,H)),
      other_side(E,A),
      other_side(A,E),
      all_but_K(B),
      all_but_K(F),
      legal_move(A,B,square(C,D),square(G,H)).

threat(A,B,square(C,D),E,king,square(F,G)) ←
      contents(A,B,square(C,D)),
      contents(E,king,square(F,G)),
      other_side(E,A),
      other_side(A,E),
      all_but_K(B),
      legal_move(E,king,square(F,G),square(C,D)).
```

The first clause is the same one given in the introduction and represents a threat between two pieces. The second clause represents a threat between a king and a piece. The system can learn in the same way a threat between a piece and a king (i.e., a check), but since the concept of being in check was initially given as a feature definition the examples where a check occurred were classified as negative instances. The total number of examples generated by the system compares very favourable against an example space of approximately $\approx 10^8$ possible examples.

Similarly the learning algorithm produced the following definition of the concept of a *fork* after learning the concept of threat and after generating 13 positive and 27 negative examples. This is a restricted version of a fork in chess which occurs whenever a piece threatens another piece and checks the king at the same time. Appendix 1 describes the learning sequence involved to learn this concept showing only the positive examples generated by the system and all the intermediate generalisations:

```
fork(A,king,square(B,C),A,D,square(E,F),G,H,square(I,J)) ←
      contents(A,king,square(B,C)),
      contents(A,D,square(E,F)),
        contents(G,H,square(I,J)),
        in_check(A,square(B,C),H,square(I,J)),
        other_side(G,A),
        other_side(A,G),
        all_but_K(D),
        all_but_K(H),
        legal_move(G,H,square(I,J),square(E,F)),
        threat(G,H,square(I,J),A,D,square(E,F)).
```

Although not implemented, we can use the definition of threat to reduce this definition to:

```
fork(A,king,square(B,C),A,D,square(E,F),G,H,square(I,J)) ←
        contents(A,king,square(B,C)),
        contents(A,D,square(E,F)),
        contents(G,H,square(I,J)),
        in_check(A,square(B,C),H,square(I,J)),
        threat(G,H,square(I,J),A,D,square(E,F)).
```

Again the total number of generated examples is several orders of magnitude smaller than the example space.

The learning algorithm as it stands has several limitations. In particular, it cannot deal with exceptions or negation (i.e., it cannot learn things like "without feature")[4]. It also cannot learn recursive concepts. This is partly due to the example representation, although in principle we could include previously generated heads into the list of relevant atoms to allow it to learn recursive concepts (although they will have to be updated with changes in the current number of arguments).

6 Conclusions and Future Research Directions

In an inductive learning system we need to consider the search through the hypothesis space and the amount of information provided by the user. We have addressed both problems by using a constrained RLGG algorithm as a model of generalisation coupled with an automatic example generator to learn a restricted form of Horn clauses. The problem of selecting a relevant set of atoms derived from the background knowledge for the RLGG algorithm has been solved by using a restricted form of Horn clauses which follows closely to our intuitive notion of a feature definition in domains like chess. We have also relaxed the example representation used in other systems by describing each example with a list of features, rather than specifying which of the arguments are relevant to the concept definition. We have reduced the user's intervention over the system for its

[4]Although negation is used in one of the concepts of the background knowledge

"correct" behaviour by presenting an automatic example generator which converges rapidly to the concept definition. The examples space is structured dynamically, allows to include domain rules if necessary to explore particular perturbations and is fairly independent of the learning algorithm. Finally, we have demonstrated the feasibility of the approach with some initial results in chess. We plan to continue this research by learning concepts which involve one or more moves, like in discovery attacks.

Acknowledgements.

I would like to thank Tim Niblett, Steve Muggleton and Peter Clark for helpful comments in the development of this work. This research was made possible by a grant from CONACYT (México).

References

Buntine, W. (1988). Generalised subsumption an its applications to induction and redundancy. *Artificial Intelligence*, (36):149–176.

Carbonell, F. and Gil, Y. (1987). Learning by experimentation. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 256–265.

Dietterich, T. and Buchanan, B. (1983). The role of experimentation in theory formation. In *Proceedings of the Second International Workshop on Machine Learning*, pages 147–155.

Feng, C. (1990). *Learning by Experimentation.* PhD thesis, Turing Institute - University of Strathclyde.

Lenat, D. B. (1976). *AM: an artificial intelligence approach to discovery in mathematics as heuristic search.* PhD thesis, Stanford University, Artificial Intelligence Laboratory. AIM-286 or STAN-CS-76-570.

Morales, E. (1990). Thesis proposal. (unpublished).

Muggleton, S. (1990). Inductive logic programming. In *First International Workshop on Algorithmic Learning Theory (ALT90)*, Tokyo, Japan.

Muggleton, S. and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–353. Kaufmann.

Muggleton, S. and Cao, F. (1990). Efficient induction of logic programs. In *First International Workshop on Algorithmic Learning Theory (ALT90)*, Tokyo, Japan.

Plotkin, G. (1969). A note on inductive generalisation. In *Machine Intelligence 5*, pages 153–163. Meltzer B. and Michie D. (Eds).

Plotkin, G. (1971a). *Automatic Methods of Inductive Inference.* PhD thesis, Edimburgh University.

Plotkin, G. (1971b). A further note on inductive generalisation. In *Machine Intelligence 6*, pages 101–124. Meltzer B. and Michie D. (Eds).

Porter, B. and Kibler, D. (1986). Experimental goal regression. *Machine Learning*, (1):249 - 286.

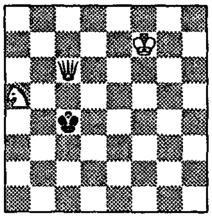
Ruff, R. and Dietterich, T. (1989). What good are experiments? In *Proc. of the Sixth International Workshop on Machine Learning*, pages 109–112, Conell Univ., Ithaca New York. Morgan Kaufmann.

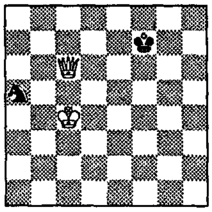
Sammut, C. and Banerji, R. (1986). Learning concepts by asking questions. In *Machine Learning: An artificial intelligence approach (Vol 2)*. R. Michalski, J. Carbonell and T. Mitchell (eds).

Winston, P. (1977). Learning structural descriptions from examples. In *The Psychology of computer vision*. Winston, P.H. (Ed), MacGraw-Hill.

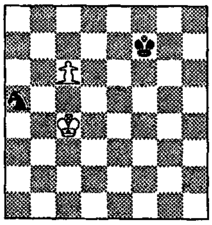
Appendix 1 **This is the sequence for learning a restricted concept of fork. Only the positive examples generated by the perturbation method are shown.**

```
|?- go.
```

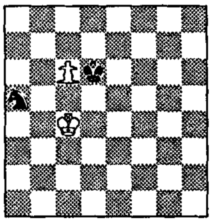




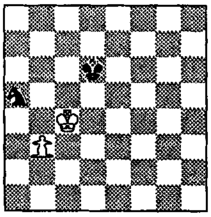
```
tmp(A,king,square(3,4),A,queen,square(3,6),
    B,king,square(6,7),B,knight,square(1,5)) ←
  contents(A,king,square(3,4)),
  contents(A,queen,square(3,6)),
  contents(B,king,square(6,7)),
  contents(B,knight,square(1,5)),
  in_check(A,square(3,4),knight,square(1,5)),
  other_side(B,A),
  other_side(A,B),
  all_but_K(queen),
  all_but_K(knight),
  diagonal_slide(queen),
  straight_slide(queen),
  sliding_piece(queen),
  legal_move(A,king,square(3,4),square(4,5)),
  legal_move(A,king,square(3,4),square(2,5)),
  legal_move(A,king,square(3,4),square(4,3)),
  legal_move(A,king,square(3,4),square(3,5)),
  legal_move(A,king,square(3,4),square(3,3)),
  legal_move(A,king,square(3,4),square(4,4)),
  legal_move(A,king,square(3,4),square(2,4)),
  legal_move(B,king,square(6,7),square(7,8)),
  legal_move(B,king,square(6,7),square(6,8)),
  legal_move(B,king,square(6,7),square(7,7)),
  legal_move(B,king,square(6,7),square(5,7)),
  legal_move(B,knight,square(1,5),square(2,7)),
  legal_move(B,knight,square(1,5),square(2,3)),
  legal_move(B,knight,square(1,5),square(3,6)),
  threat(B,knight,square(1,5),A,queen,square(3,6)).
```



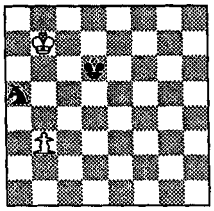
```
tmp(A,king,square(3,4),A,B,square(3,6),
    C,king,square(6,7),C,knight,square(1,5)) ←
  contents(C,knight,square(1,5)),
  contents(C,king,square(6,7)),
  contents(A,B,square(3,6)),
  contents(A,king,square(3,4)),
  in_check(A,square(3,4),knight,square(1,5)),
  other_side(A,C),
  other_side(C,A),
  all_but_K(knight),
  all_but_K(B),
  legal_move(C,knight,square(1,5),square(3,6)),
  legal_move(C,knight,square(1,5),square(2,3)),
  legal_move(C,knight,square(1,5),square(2,7)),
  legal_move(C,king,square(6,7),square(5,7)),
  legal_move(C,king,square(6,7),square(7,7)),
  legal_move(C,king,square(6,7),square(6,8)),
  legal_move(C,king,square(6,7),square(7,8)),
  legal_move(A,king,square(3,4),square(2,4)),
  legal_move(A,king,square(3,4),square(4,4)),
  legal_move(A,king,square(3,4),square(3,3)),
  legal_move(A,king,square(3,4),square(3,5)),
  legal_move(A,king,square(3,4),square(4,3)),
  legal_move(A,king,square(3,4),square(2,5)),
  legal_move(A,king,square(3,4),square(4,5)),
  threat(C,knight,square(1,5),A,B,square(3,6)).
```



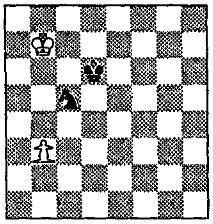
```
tmp(A,king,square(3,4),A,B,square(3,6),
    C,king,square(D,E),C,knight,square(1,5)) ←
  contents(A,king,square(3,4)),
  contents(A,B,square(3,6)),
  contents(C,king,square(D,E)),
  contents(C,knight,square(1,5)),
  in_check(A,square(3,4),knight,square(1,5)),
  other_side(C,A),
  other_side(A,C),
  all_but_K(B),
  all_but_K(knight),
  legal_move(A,king,square(3,4),square(2,5)),
  legal_move(A,king,square(3,4),square(4,3)),
  legal_move(A,king,square(3,4),square(3,3)),
  legal_move(A,king,square(3,4),square(4,4)),
  legal_move(A,king,square(3,4),square(2,4)),
  legal_move(C,king,square(D,E),square(5,7)),
  legal_move(C,king,square(D,E),square(5,E)),
  legal_move(C,knight,square(1,5),square(2,7)),
  legal_move(C,knight,square(1,5),square(2,3)),
  legal_move(C,knight,square(1,5),square(3,6)),
  threat(C,knight,square(1,5),A,B,square(3,6)).
```



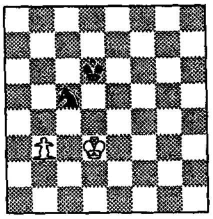
```
tmp(A,king,square(3,4),A,B,square(C,D),
    E,king,square(F,G),E,knight,square(1,5)) ←
  contents(E,knight,square(1,5)),
  contents(E,king,square(F,G)),
  contents(A,B,square(C,D)),
  contents(A,king,square(3,4)),
  in_check(A,square(3,4),knight,square(1,5)),
  other_side(A,E),
other_side(E,A),
all_but_K(knight),
all_but_K(B),
legal_move(E,knight,square(1,5),square(C,D)),
legal_move(E,knight,square(1,5),square(3,6)),
legal_move(E,knight,square(1,5),square(2,3)),
legal_move(E,knight,square(1,5),square(2,7)),
legal_move(E,king,square(F,G),square(5,G)),
legal_move(E,king,square(F,G),square(5,7)),
legal_move(A,king,square(3,4),square(2,4)),
legal_move(A,king,square(3,4),square(4,4)),
legal_move(A,king,square(3,4),square(3,3)),
legal_move(A,king,square(3,4),square(4,3)),
legal_move(A,king,square(3,4),square(2,5)),
threat(E,knight,square(1,5),A,B,square(C,D)).
```



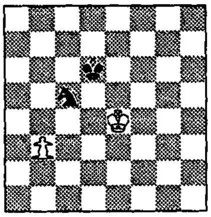
```
tmp(A,king,square(B,C),A,D,square(E,F),
    G,king,square(H,I),G,knight,square(1,5)) ←
  contents(A,king,square(B,C)),
  contents(A,D,square(E,F)),
  contents(G,king,square(H,I)),
  contents(G,knight,square(1,5)),
  in_check(A,square(B,C),knight,square(1,5)),
  other_side(G,A),
  other_side(A,G),
  all_but_K(D),
  all_but_K(knight),
  legal_move(G,king,square(H,I),square(5,7)),
  legal_move(G,king,square(H,I),square(5,I)),
  legal_move(G,knight,square(1,5),square(2,3)),
  legal_move(G,knight,square(1,5),square(3,6)),
  legal_move(G,knight,square(1,5),square(E,F)),
  threat(G,knight,square(1,5),A,D,square(E,F)).
```



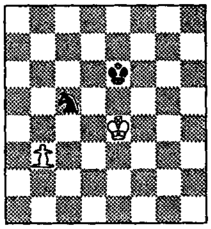
```
tmp(A,king,square(B,C),A,D,square(E,F),
    G,king,square(H,I),G,knight,square(J,5)) ←
  contents(G,knight,square(J,5)),
  contents(G,king,square(H,I)),
  contents(A,D,square(E,F)),
  contents(A,king,square(B,C)),
  in_check(A,square(B,C),knight,square(J,5)),
  other_side(A,G),
  other_side(G,A),
  all_but_K(knight),
  all_but_K(D),
  legal_move(G,knight,square(J,5),square(E,F)),
  legal_move(G,knight,square(J,5),square(2,3)),
  legal_move(G,king,square(H,I),square(5,I)),
  legal_move(G,king,square(H,I),square(5,7)),
  threat(G,knight,square(J,5),A,D,square(E,F)).
```



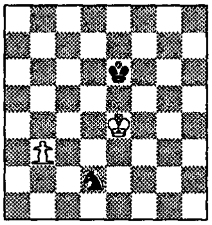
```
tmp(A,king,square(B,C),A,D,square(E,F),
    G,king,square(H,I),G,knight,square(J,5)) ←
  contents(A,king,square(B,C)),
  contents(A,D,square(E,F)),
  contents(G,king,square(H,I)),
  contents(G,knight,square(J,5)),
  in_check(A,square(B,C),knight,square(J,5)),
  other_side(G,A),
  other_side(A,G),
  all_but_K(D),
  all_but_K(knight),
  legal_move(G,king,square(H,I),square(5,7)),
  legal_move(G,king,square(H,I),square(5,I)),
  legal_move(G,knight,square(J,5),square(2,3)),
  legal_move(G,knight,square(J,5),square(E,F)),
  threat(G,knight,square(J,5),A,D,square(E,F)).
```



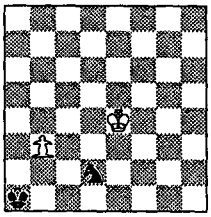
```
tmp(A,king,square(B,C),A,D,square(E,F),
    G,king,square(H,I),G,knight,square(J,5)) ←
  contents(G,knight,square(J,5)),
  contents(G,king,square(H,I)),
  contents(A,D,square(E,F)),
  contents(A,king,square(B,C)),
  in_check(A,square(B,C),knight,square(J,5)),
  other_side(A,G),
  other_side(G,A),
  all_but_K(knight),
  all_but_K(D),
  legal_move(G,knight,square(J,5),square(E,F)),
  legal_move(G,knight,square(J,5),square(2,3)),
  legal_move(G,king,square(H,I),square(5,I)),
  legal_move(G,king,square(H,I),square(5,7)),
  threat(G,knight,square(J,5),A,D,square(E,F)).
```



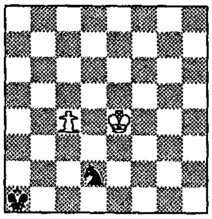
```
tmp(A,king,square(B,C),A,D,square(E,F),
    G,king,square(H,I),G,knight,square(J,5)) ←
  contents(A,king,square(B,C)),
  contents(A,D,square(E,F)),
  contents(G,king,square(H,I)),
  contents(G,knight,square(J,5)),
  in_check(A,square(B,C),knight,square(J,5)),
    other_side(G,A),
    other_side(A,G),
    all_but_K(D),
    all_but_K(knight),
    legal_move(G,king,square(H,I),square(5,7)),
    legal_move(G,knight,square(J,5),square(2,3)),
    legal_move(G,knight,square(J,5),square(E,F)),
    threat(G,knight,square(J,5),A,D,square(E,F)).
```



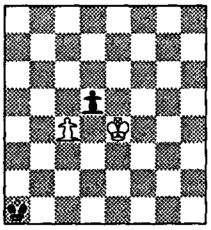
```
tmp(A,king,square(B,C),A,D,square(E,F),
      G,king,square(H,I),G,knight,square(J,K)) ←
    contents(G,knight,square(J,K)),
    contents(G,king,square(H,I)),
    contents(A,D,square(E,F)),
    contents(A,king,square(B,C)),
    in_check(A,square(B,C),knight,square(J,K)),
    other_side(A,G),
    other_side(G,A),
    all_but_K(knight),
    all_but_K(D),
    legal_move(G,knight,square(J,K),square(E,F)),
    legal_move(G,knight,square(J,K),square(2,3)),
    legal_move(G,king,square(H,I),square(5,7)),
    threat(G,knight,square(J,K),A,D,square(E,F)).
```



```
tmp(A,king,square(B,C),A,D,square(E,F),
      G,knight,square(H,I)) ←
    contents(A,king,square(B,C)),
    contents(A,D,square(E,F)),
    contents(G,knight,square(H,I)),
    in_check(A,square(B,C),knight,square(H,I)),
    other_side(G,A),
    other_side(A,G),
    all_but_K(D),
    all_but_K(knight),
    legal_move(G,knight,square(H,I),square(2,3)),
    legal_move(G,knight,square(H,I),square(E,F)),
    threat(G,knight,square(H,I),A,D,square(E,F)).
```



```
tmp(A,king,square(B,C),A,D,square(E,F),
      G,knight,square(H,I)) ←
    contents(G,knight,square(H,I)),
    contents(A,D,square(E,F)),
    contents(A,king,square(B,C)),
    in_check(A,square(B,C),knight,square(H,I)),
    other_side(A,G),
    other_side(G,A),
    all_but_K(knight),
    all_but_K(D),
    legal_move(G,knight,square(H,I),square(E,F)),
    legal_move(G,knight,square(H,I),square(2,3)),
    threat(G,knight,square(H,I),A,D,square(E,F)).
```



```
tmp(A,king,square(B,C),A,D,square(E,F),
      G,H,square(I,J)) ←
    contents(A,king,square(B,C)),
    contents(A,D,square(E,F)),
    contents(G,H,square(I,J)),
    in_check(A,square(B,C),H,square(I,J)),
    other_side(G,A),
    other_side(A,G),
    all_but_K(D),
    all_but_K(H),
    legal_move(G,H,square(I,J),square(E,F)),
    threat(G,H,square(I,J),A,D,square(E,F)).

How would you like to call the concept? fork.

yes

| ?-
```