First-Order Induction of Patterns in Chess

Eduardo Morales
Computer Science
The Turing Institute - University of Strathclyde
Submited 1992

Abstract

It has been argued that much of human intelligence can be viewed as the process of matching stored patterns. Cognitive studies of the play of Chess suggest that the analysis of positions by people is guided by the recollection of patterns. Several Chess playing systems have implemented a pattern-based approach to play. The research focus has been on how to play Chess with existing patterns rather than on the acquisition of patterns.

This thesis investigates the use of inductive logic programming (ILP) to acquire Chess patterns that can be used for play. Patterns are expressed in a subset of Horn clause logic.

A learning mechanism (PAL) based on relative least general generalisation (rlgg) is described. This mechanism permits background Chess knowledge to be used during learning. The learning process is driven by an automatic example generator which minimises the user's guidance of the learning process.

It is shown that PAL can learn Chess concepts such as fork, discovered check, skewer and pin, which are beyond the current state of the art for machine learning systems. The utility of the acquired patterns is demonstrated by learning patterns sufficient for correct play in the King and Rook vs. King endgame. It is demonstrated that the approach is not restricted to Chess, by showing how PAL can be used to learn qualitative models of simple dynamic systems.

A comparison with previous work in ILP is made. The limitations of this approach are discussed, and future research directions outlined.

Acknowledgements

I am greatly indebted to my supervisor, Tim Niblett, for his encouragement and continual support during the development of this research. His perceptive suggestions, advice, and "be precise" comments helped me to clarify the ideas presented in the thesis. This thesis also benefit from useful comments and suggestions of Robin Boswell, Steve Muggleton and Ashwin Srinivasan. Peter Clark and all the members of the Machine Learning Group at the Turing Institute contributed with additional feedback during the development of this work. I am thankful to Ian Watson and to Lisa Mac Mannus who helped to correct the English mistakes of the thesis.

My special gratitude goes to Fernanda for her continual caring and understanding and to my parents for their moral (as well as financial) support during our stay in Glasgow.

This research was funded with a grant from CONACYT (México) and a supplement from *Banco de México* obtained through the *Instituto de Investigaciones Eléctricas*. Finally, this research would not have been possible without the excellent facilities provided by the Turing Institute.

Chapter 1

Introduction

1.1 Motivation

Artificial Intelligence (AI) is devoted to designing and programming machines to accomplish tasks that people use their intelligence for. Perhaps the most distinctive feature of intelligent behaviour is learning. From the introduction of computers, researchers have tried to program computers with different learning capabilities [Tur63]. The development of successful machine learning methods is one of the long term goals of AI. Instructing a computer to perform a task is a time-consuming process which requires hand-coding a complete and correct algorithm. From a practical point of view, machine learning tries to ease this burden.

AI attempts to understand how human beings think, by studying the behaviour of machine designs and programs that model some aspects of the human cognitive process. From this point of view, few domains offer such substantial psychological evidence of human reasoning as does Chess. Despite considerable attention from researchers in AI, little progress has been made in applying machine learning to Chess.

The idea of a Chess playing machine capable of beating the best human player has been one of the most seductive challenges since the introduction of computers. In trying to fulfill this, most computer Chess programs have deviated from studies in cognition, and followed an “engineering” approach, where emphasis has been made on playing by searching a huge number of Chess positions.

Studies in cognitive science suggest that human players remember posi-

tional patterns to analyse Chess positions and derive their playing strategies with little search involved [CS88, dG65]. Pattern-based reasoning is not exclusive to Chess and it has been argued that a large part of human intelligence can be viewed as the process of matching stored patterns [Cam66, Lor73]. For example, a doctor recognises symptoms in patients to suggest a diagnosis, a mechanic identifies some machine behaviour to suggest a failed component, etc. In a similar way, a Chess player analyses a position by recognising some positional relations in the game to suggest a move.

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 pattern. For example given the position of Figure 1.1 (left), a Chess player recognises (illustrated to the right) that:

- The white Rook *threatens* the black Queen.
- The black Bishop is *pinned*.
- The white Queen is *threatened* by the 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).

1.2 The aim of the thesis

> This thesis 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.

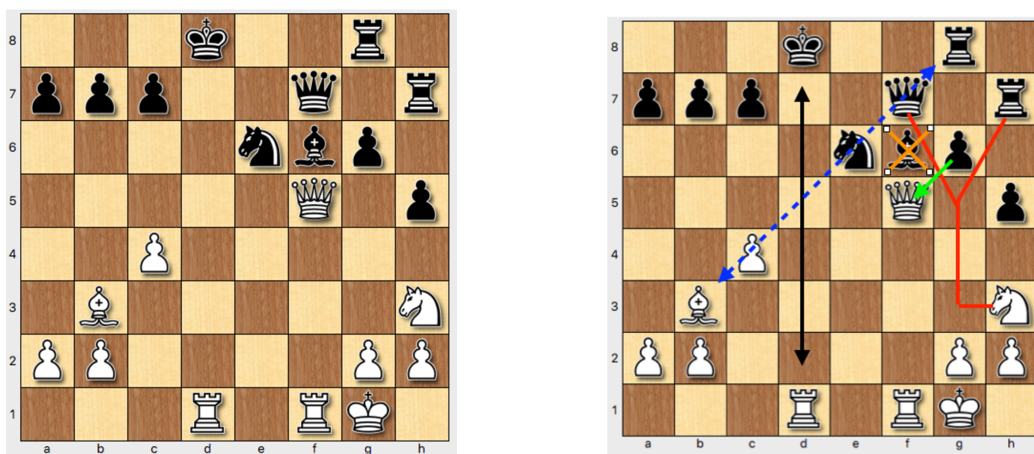


Figure 1.1: An example position and patterns in it

Previous work has shown how playing strategies can be constructed following a pattern-based approach [Ber77, Bra77, Bra82, Hub68, Pit77, Wil79]. 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 [Qui83, Sha87]. In this case too, most of the work is dedicated to the definition of attributes adequate to describe the examples and from which the target concept can be constructed. We want to investigate whether Chess patterns can be learned effectively from simple descriptions of Chess positions together with the rules of the game, rather than investigating the means by which patterns can be used for play.

Broadly speaking, concept learning in AI depends on the representation language and on the examples provided to the system. In choosing a representation language we must consider not only its representational power (e.g., the language must be adequate to represent patterns in Chess), but also its practical effectiveness – the language must be learnable. One of the obstacles to effective use of previous machine learning techniques in learning concepts in Chess has been an inadequate hypothesis language. A comparative study between different machine learning formalisms has concluded that the ability to produce high performance in a domain like Chess is almost entirely dependent on the ability to express first order predicate relationships [MBHMM89]. Such relationships are not expressible in many of the systems

used to learn Chess concepts.

Inductive logic programming (ILP) is an area of AI which combines logic programming with machine learning [Mug91a]. It provides a mechanism and language which should be able to represent and learn patterns in Chess. The characteristics of the background knowledge that an inductive system is provided with, have a strong influence on the effectiveness of the learning process, and most ILP systems have used a limited background knowledge to achieve practical results. The use of background knowledge is important as it allows us to represent examples in a simpler way and reduces the inductive steps require to learn particular concepts. Concepts in Chess can be expressed in terms of relations between pieces (e.g., threats, checks), relations between pieces and places (e.g., legal moves, distance to a particular place), relations between places (e.g., adjacent squares), etc. This makes Chess a particularly good domain where the power of ILP can be tested as a varied set of concepts can be induced over a more realistic background knowledge.

The minimum Chess knowledge required to start playing consists of the rules of the game. Chess players are able to increase their pattern vocabulary after seeing a large number of positions. Similarly, we would like to learn patterns in Chess, from descriptions of Chess positions together with the rules of the game. Rather than trying to learn concepts in the way humans represent them, our interest lies on inducing symbolic representations of concepts which could be used to design playing strategies following a pattern-based approach. To simplify the task, our approach is to learn single concepts from descriptions of Chess positions, rather than learning, perhaps several concepts, from traces of games.

The aim of this thesis is to demonstrate that Chess patterns, capable of play, can be expressed compactly and economically in first order logic and that such patterns can be learned efficiently from examples and general purpose background knowledge about Chess.

1.3 Chess and Inductive Logic Programming

Learning the Chess concepts we are interested in is a non-trivial task and beyond the capability of existing ILP systems. This thesis identifies the characteristics of simple Chess concepts that make them difficult to learn and demonstrates a learning mechanism (PAL) that substantially overcomes these problems. We show, by applying PAL to a different domain that, from

the ILP viewpoint, our results can be generalised.

In section 1.2 above we discuss the usefulness of a first order representation. There are several technical issues:

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. 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 ILP 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.

Issue (1) is addressed by a specification of which facts deducible from the background knowledge are relevant in a particular state. Broadly speaking these are facts about the relationships between the components of the state (Chess position). This issue is also addressed by use of novel constraint based on labelling the different components (pieces) which are used to describe examples (Chess positions) to guide and constrained the *lgg* algorithm, as the *lgg* is only computed between literals which involve the same components (Chess pieces). This constraint is described in section 5.2.3, and provides a considerable reduction of the search space.

These two refinements also address issue (3) since the learning mechanism automatically produces a set of relevant ground facts from the background knowledge together with specific information about the current state, rather than requiring the user to generate these examples.

Issue (2) is addressed by PAL which can learn a limited class of non-determinate concepts.

PAL contains an automatic example generator, which puts the onus of providing examples on the system rather than the user. The user has just to classify states and accept or reject hypotheses. We demonstrate that efficient learning is possible using this example generator. This addresses issue (4).

In summary, the thesis shows that several Chess concepts, which are outside the scope of current ILP systems, can be learn by PAL. It is shown as well, that the learned concepts can be used to construct a correct playing strategy. Furthermore, chapter 8 demonstrates that the approach is not restricted to Chess, by showing how PAL can be used in another domain, where other ILP systems have similar difficulties to those they experience in the Chess domain.

1.4 A guide through the thesis

Chapter 2 analyses Chess as a domain and looks at the different approaches employed by Chess programs. It reviews the supporting psychological evidence which suggests that human players follow a pattern-based approach and argues for the desirability of learning concepts in the form of patterns.

Chapter 3 presents a general overview of concept formation in machine learning. It reviews the different machine learning techniques and how they have been used in Chess. It provides evidence that a relational language is suitable for expressing concepts in Chess.

Chapter 4 reviews concept formation within a first order formalism. It describes a generalisation model for Horn clauses introduced by Plotkin [Plo69, Plo71a] which has been used by several first order systems. It also analyses why current first order learning systems are inadequate for learning patterns in Chess.

Chapter 5 describes PAL, a first order inductive system capable of learning concepts in Chess. It describes in detail the generalisation method used by the system, an automatic example generator based on “perturbations”, and how it is used to guide the learning process.

Chapter 6 compares PAL with other first order systems and discusses their applicability in Chess by looking at their performance over a simple concept. It then shows how PAL is used to learn several Chess concepts, such as those in Figure 1.1, e.g., *pin*, *skewer*, *discovered-check*, *discovered-*

attack, *fork*, etc. It discusses the robustness of PAL and points out its main deficiencies.

Chapter 7 shows that the patterns learned by PAL can be used to construct a correct playing strategy for a simple endgame (King and Rook against King). It discusses the strategy which was followed to design the strategy, how the strategy was proved to be correct, and how it can be improved by adding extra background knowledge.

Chapter 8 shows how PAL can be applied to other domains. In particular, PAL is used to learn a qualitative model of a simple dynamic system from its state descriptions. The performance of PAL is compared with that of Golem in a domain where Golem was originally tested. The performance of PAL in this domain helps to asses the general applicability of the approach.

Finally, conclusions and future work are given in chapter 9.

Chapter 2

Problem solving in Chess

Problem solving, as part of human intelligence, has been widely studied by the Artificial Intelligence (AI) community. Chess, in particular, has received the attention of computer scientists and psychologists alike. Computer problem solving in Chess has followed two general approaches. The engineering or search intensive approach, where a large number of positions are explored with little Chess knowledge involved, and the cognitive or knowledge intensive approach in which, closer to psychological studies, a large amount of knowledge is used to perform a more selective search.

This chapter looks at Chess as a test domain for AI, and in particular for machine learning. It reviews the psychological studies, which suggest that human players use knowledge in the form of patterns to analyse positions and derive their playing strategies. It examines previous pattern-based approaches to Chess and provides an insight for the desirability of learning patterns in Chess.

2.1 Chess as a machine learning domain

Chess has been the game *par excellence* in Artificial Intelligence and has served as a convenient vehicle for studying cognition and perception [Cha77, dG65, Nie91, SG73]. It is constrained but not trivial, it has many similarities to real world domains, where special cases exist, the space is too large to perform exhaustive search, it definitely needs expertise (at least for humans) as well as search, and the results of actions can be only partially predicted. The rules of play can be easily formulated, experiments are easy to perform

and it has been widely studied (a recent panel discussion on the role of Chess in AI research is given in [LHS+91]).

Michie [Mic82] identifies five key features in Chess.

1. The game constitutes a fully defined and well-formalised domain.
2. The game challenges the highest levels of human intellectual capacity.
3. The challenge extends over a large range of cognitive functions such as logical calculation, rote learning, concept formation, analogical thinking, imagination, deductive and inductive reasoning.
4. A massive and detailed corpus of knowledge has been accumulated.
5. A generally accepted numerical scale of performance is available.

Together, these make Chess a good domain for AI research and, in particular, for machine learning research.

2.2 Historical background

The idea of a Chess playing machine dates back to the 1760's when a Hungarian inventor, Wolfang von Kempelen, astounded Europe with his Maelzel Chess Automaton, later discovered to be operated by a diminutive Chess master hidden in a secret compartment. A more honest attempt was made in the 1914 by a Spanish inventor, L. Torres y Quevedo, who constructed a device that played the King and Rook against King endgame (starting from certain positions). However, the seminal work on which most of the current Chess programs are based was done by Shannon [Sha88], building on the discoveries of John von Neumann and Oskar Morgenstern, who in their general theory of games had devised the minimax algorithm by which the best move can be calculated. Shannon did not present a particular program; it remained for A.M. Turing [Tur53] to describe a program along these lines that was sufficiently simple to be simulated by hand.

The basic idea behind Shannon's proposal is as follows. As he observed, in Chess there are a finite number of positions each with a finite number of alternative moves. Thus Chess can be completely described as a branching tree. The nodes corresponding to positions and the branches to the alternative moves from each position. It is clear that for a player who could view

the entire tree with all its outcomes, Chess becomes a simple game. Starting from the terminal positions, he can work backwards, determining on each node which is the best branch for him, or his opponent, until he arrives at an alternative for his next move. As a complete search tree for Chess requires exploring around 10^{120} positions [Sha88], current systems explore up to a certain depth, estimate the value of those positions, and combine them back to choose the move with the highest score. An evaluation function is used to estimate the winning chances from the point of view of one of the players. The higher the value, the higher the player's chances to win, the lower the value, the higher the opponent's chances to win. One player tends to move to high value positions (which is called MAX), while the other tends to low value positions (which is called MIN). Whenever MAX is to move he or she chooses a move that maximises its value, while on the contrary, MIN chooses a move that minimises its value. Given the values at certain depth-level positions, this principle (called minimax) determines the values of all the other positions in the search tree.

2.3 Search intensive approach

Most computer Chess programs have followed variants of the minimax algorithm. In the late 70's, the depth of search was found to correlate almost linearly with the program's rating [New88]. Each additional ply[1] added about 200 rating points to the computer Chess strength and it was estimated that by 1992 a computer would be better than any human. Research concentrated on the design of special-purpose machines to explore the search tree. The analytical speed of the current Chess champion machine, called Deep-Thought, was 750,000 positions per second in 1990 and its performance rating exceeded 2600 [HACN90]. It is estimated that a new implementation will increase the speed of analysis by more than 1,000-fold, to about 1 billion positions per second and that it will be ready by 1992 (which might fit with earlier predictions). Deep-Thought's search gives a solid horizon of 10 ply in the opening phase, 9 ply in the middle game, rising to 10 ply and above in the endgame. A singular extension feature searches forceful variations out to a depth of 16 to 18 ply [Mic89]. While the search intensive approach has achieved significant performance in Chess, the approach is of restricted applicability in other domains, and its model of "expertise" is opaque to human players.

[1] A ply is defined as a single move of one of the players.

2.4 Psychological evidence

Chess has been subject of several cognitive studies in the past. Most of the psychological evidence comes from the experimental research of de Groot [dG65]. In one set of experiments, subjects of differing Chess skill were shown Chess positions for a few seconds and then were asked to reconstruct the positions. On positions taken from actual games, experienced players performed significantly better than novices. However, on random, but legal Chess positions, novices recalled positions as well as experienced players. The explanation being that the player builds up a library of Chess "patterns" (configurations of pieces) that he sees frequently on the board. When presented with a new position, he can code it in terms of a few appropriate patterns from his library. De Groot also asked Chess players to find the best move of an interesting position and to talk aloud while thinking. From an analysis of the verbal protocols, de Groot also concluded that it is not in depth nor in breadth that Grandmasters necessarily calculate any more extensively than experienced competition players, say one class below the master level, do. Much more spectacular is the grandmaster's superiority in Chess perception, namely in registering, understanding and reproducing real game positions after an exposure of a few seconds [dG86]. Chase and Simon [CS88] concluded as well, on later experiments, that the basic ability underlying Chess skill relies on the capacity to perceive familiar patterns quickly. This suggests that one reasonable measure of a person's Chess knowledge is the number of patterns he or she has stored in memory, since the number appears to grow with the competence of the player.

Using patterns to guide the human reasoning process is not exclusive to Chess. In an engineering domain, a pattern might refer to a particular behaviour in sampling data, which can then be used for fault diagnosis. In the context of Chess we refer to a pattern as a relation between pieces and places in the board, which can then be used to suggest a move. In general, a pattern-based approach will be applicable to domains where a search intensive approach, like minimax, will fail to work or otherwise be inapplicable.

2.5 Playing Chess with patterns

Following psychological evidence, some systems have used knowledge in the form of patterns to guide their playing strategies in certain end-games [Bra77, Bra82, Hub68] and tactically sharp middle-game positions [Ber77, Pit77, Wil79]. The idea is that although the use of knowledge takes time, it is compensated for by a smaller and more directed search.

Perhaps the most successful system in terms of depth analysis is Wilkins' PARADISE system [Wil79]. It consists of around 200 production rules, constructed around the recognition of patterns, and is able to elaborate plans of up to 19-ply depth. Each rule has a pattern to match (i.e., an interrelated set of features based on relations between pieces and places in the board) as its condition. More primitive patterns are used as building blocks for more complex patterns. For each instance of the condition found, the production posts zero or more concepts in a data base (which can be seen as a blackboard [HR85]) for use by the system in its reasoning processes. A search tree with additional rules is used to show that one move suggested by the pattern-based analysis is in fact the best. PARADISE was tested on tactically sharp middle-game positions, i.e., positions where the success can be judged by the gain of material rather than a positional advantage.

Bramer [Bra77] describes a model, based on the recognition of patterns, for the King and Rook against King (KRK) and the King and Pawn against King (KPK) endgames. In Bramer's model, all the immediate successors of a position are matched against ranked patterns. The chosen move is the one which matches the pattern with the highest score. Occasional ties are resolved with additional associated functions, but no further search is performed.

Huberman [Hub68], constructed programs for three endgames, King and Rook against King (KRK), King and two Bishops against King (KBBK), and King, Bishop, and Knight against King (KBNK). Her model is only intended to be applied to positions in which the winning party can construct a *forcing tree*. That is, a tree in which the winning party can always force a better position for him/her. This is done with two functions, *better* and *worse*, that are used to prune a breadth first search.

Michie and Bratko extended Huberman's ideas by introducing an advice language, in AL1/AL3, based on 'pieces-of-advice' to construct forcing-trees [Bra82, Mic76]. Each piece-of-advice has a better-goal to achieve, a holding-goal to maintain, and move-constraints that select a subset of all legal moves.

Similarly to Huberman's definitions for the *better* and *worse* functions, the goal predicates used in the definition of pieces-of-advice are based on the recognition of patterns. In AL1/AL3, a plan succeeds when there is a forcing tree which represents a detailed strategy for the achievement of the goals of a piece of advice. When a plan fails, a combination of plans is considered. Counter-plans (plans from the opponent's point of view) are also considered during this process.

Gadwal et al. have recently extended the advice language used in AL1/AL3 to construct plans at different levels (expert and student) which are then used in a tutoring system for helping students to learn how to play Bishop-Pawn endgames [GGM91]. Student plans are created by weakening expert plans. Different strategies are compiled along with information about the possible responses, the best responses, whether or not the better-goal and the holding-goal are satisfied, etc. In the tutoring mode, the system tries to identify the plan behind the student moves and offers advice accordingly, based on its stored information.

Levinson and Snyder [LS91] report a system, Morph, which uses weighted patterns to guide its playing strategy. Morph makes a move by generating all legal successors of the current position, evaluating each position using its current weighted patterns, and choosing the position that is considered less favourable to the opponent. Positions are evaluated by combining the weights of the most specific patterns that match a position using an evaluation function. Morph is able to adjust the weights of the patterns and learn new patterns from traces of games and it will be further discussed in chapter 3, where different machine learning approaches for Chess are reviewed.

Patterns in Chess have been used to suggest moves (e.g., [Bra77, LS91]), suggest goals from which to generate plans (e.g., capture a piece as in [Wil79] or achieve/maintain a goal as in [Bra82, Mic76]), and have been used to focus the search by suggesting only a subset of possible moves (e.g., [BE90]).

2.6 Why learn patterns?

A pattern-based approach has proven to be adequate as a reasoning strategy in several domains. In particular, production systems or expert systems, which has been applied to a wide range of domains can be regarded as pattern-based systems, where each production rule has a pattern to match as conditions. The emphasis in Chess systems which follow a pattern-based

approach, has been in developing ways of using and combining patterns to produce plans, while a substantial programming effort is devoted to the selection of the *right* patterns for the task. This makes the approach difficult to implement, susceptible to errors and of restricted applicability. The need for automatically generating patterns has long been recognised [Mic77, Qui83], yet little progress has been made within the machine learning community (different machine learning approaches will be reviewed in chapter 3).

Extracting skill knowledge has been the fundamental bottleneck in the construction of expert systems [FM83] and has been one of the major incentives for machine learning research. From cognitive studies it appears that practice is the major independent variable in the acquisition of Chess skill [CS88], and that this is linked to the number of patterns that a player has stored in memory. Although the patterns used by computer programs are not necessarily meant to be equivalent to those used by humans, their definitions often requires the extraction of knowledge from experts. The difficulties in the extraction of knowledge from human experts makes the mechanisation of learning patterns a desirable goal.

In general, the use of a pattern-based reasoning process has been restricted by the time-consuming process of defining the required patterns for the task. From a programming perspective, machine learning can ease this limitation by automating the acquisition of patterns. Concept learning from examples, a large subarea of machine learning, provides a framework in which general descriptions of classes of objects are induced from examples. In principle, this framework can be used to learn pattern definitions from a set of example positions. The next chapter provides a general characterisation for concept learning and analyses why different machine learning approaches have been inadequate for Chess.

Chapter 3

Machine learning

Perhaps the most distinctive characteristic of human intelligence is learning. It includes the acquisition of knowledge, development of skills through instruction or practice, organisation of knowledge, discovery of facts, etc. In the same way, Machine Learning (ML) is devoted to the study and computer modelling of learning processes in their multiple manifestations. Instructing a computer to perform a task often requires a time-consuming and difficult process of hand-coding a complete and correct algorithm. Machine Learning strives to ease the burden by providing tools to automate this process.

A large part of machine learning is devoted to the construction of concept descriptions from a set of positive and negative examples. Despite the relatively easy access to a large number of examples and the existence of recognised expertise, Chess still remains as a challenging domain for concept formation.

This chapter is divided in two sections. Section 3.1 provides a general background for concept learning. Section 3.2 describes different machine learning techniques used in Chess, shows their problems, and provides supporting evidence for an adequate learning framework.

3.1 Concept learning

A large subarea of machine learning is devoted to inferring rules from examples. General descriptions of classes of objects, obtained from a set of examples, can be used for classification and prediction. Its main interest is not in learning concepts the way humans do, but rather to induce symbolic

representations of them (e.g., see [Sim81]). The availability of a large number of examples together with relatively easy access to recognised Chess experience, provide an adequate basis in which an inductive inference framework can be applied to learn patterns in Chess. Angluin and Smith [AS83] list five items that must be specified to characterise an inductive inference problem.

1. The Class of Rules.
2. The Hypothesis Space.
3. The Set of Examples and Presentation.
4. The Class of Inference Methods.
5. The Criteria for a Successful Inference.

The rule class denotes the class of functions or languages under consideration. For instance, all regular sets over a particular alphabet, context-free languages, recursively enumerable functions, Prolog programs, etc. The hypothesis space is a set of descriptions such that each rule in the class has at least one description in the hypothesis space. Different hypothesis spaces can be used for the same rule class. For instance, if the class of rules is all regular sets over the alphabet {0,1}, the hypothesis space can be all the regular expressions over the same alphabet, deterministic finite acceptors, or context-free grammars. The hypothesis space must have descriptions for all the rules in the class, but it may contain descriptions of other things as well. For convenience, we will assume that the language the hypothesis space describes (i.e., the hypothesis language) is the same as the class of rules, and introduce the following definitions:

Hypothesis Language : The syntax used in constructing hypotheses.

Hypothesis Space : Set of all possible hypotheses within the hypothesis language.

3.1.1 The hypothesis space

The hypothesis language determines the hypothesis space from which the inference method selects its rules. The chosen language imposes a constraint (or 'bias') on what can be learned and what reasoning strategies are allowed.

In choosing a language, we must consider not only what we want the system to do, but also what information must be specified in advance to allow the system to solve the problem, whether the system will be able to solve the problem, and in what time. The hypothesis language broadly depends on the application area. Once a particular language is chosen, most of the developer's time is spent in carefully defining *adequate* knowledge structures for the learning task. This time-consuming work becomes even more critical when the hypothesis language restricts the expressiveness in such a way that the domain knowledge needs to be 'squashed' into the adopted formalism. This is further discussed in section 3.2.5, where an adequate hypothesis language for Chess is considered, and in chapter 5, where constraints are introduced on this language to improve tractability.

The induction process can be viewed as a search for hypotheses or rules [Mit82]. The whole space of possible rules can be systematically searched until a rule is found that agrees with all the data so far. Given a particular hypothesis space, suppose that there is an enumeration of descriptions, say $d_1, \ldots, d_n$, such that each rule in the hypothesis space has one or more descriptions in this enumeration. Given any collection of examples of a rule, the method of *identification by enumeration* goes down this list to find the first description, say d_i, that is compatible with the given examples and then conjectures d_i [Gol67]. This method, although general and powerful, is impractical for all but very limited cases, due to the size of the hypothesis space.

For learning to take place efficiently, it is often crucial to structure the hypothesis space. This can be done with a model of generalisation, although there are other ways too (some of which are discussed in chapter 4). This thesis is largely concerned with hypothesis search of clauses[1]. Roughly, a clause C_1 is more general than another clause C_2 (or C_2 is more specific than C_1) if C_2 can be deduced from C_1. A formal definition of generalisation (which is used as the principal structuring technique in the thesis) will be given in chapter 4. Clauses can be organised in a hierarchy, where all the clauses below a node in a specialisation (generalisation) hierarchy are specialisations (generalisations) of the node. This allows pruning of complete branches during search knowing that all the specialisations or generalisations of a clause inherit some property. The most common properties are those of failing to prove a fact known to be true (e.g., a positive example) or prov-

[1] A formal definition of a clause will be given in chapter 4.

ing a fact known to be false (e.g., a negative example). Knowing particular generalisations and specialisations of a particular clause can also delimit the space of potential clauses [Mit82].

3.1.2 Set of examples and presentation

Gold [Gol67] considers different types of data presentation and their effects on the inference of languages. A presentation can consist of positive examples only, or positive and negative examples. Most inference methods require admissible presentations. That is, for every false rule that is consistent with the positive examples, there is a negative example to falsify it. This reflects Popper's methodological requirement that theories should be refutable by facts [Pop59]. Examples are used for testing and forming hypotheses. In practice, a selection of examples is made over the example space. This selection can be made by an oracle, left to the environment, randomly selected, or proposed by the system. A "good" selection of examples can be used to improve the system's performance, and in some cases, an inference method can be highly dependent on it. This is further discussed in chapter 4 when automatic example generators are reviewed. In particular, an example selection can be improved with knowledge about the domain. This will be discussed in chapter 5 where domain knowledge is used to constrain the generation of examples and in chapter 6, where the symmetric properties of the domain are used to improve the learning rate.

A reasonable example presentation for learning a *single* pattern in Chess is a set of Chess positions with/without the particular pattern that we are interested in learning. In practice, Chess players learn patterns without being explicitly told that instances of patterns are being shown. Learning an unknown number of patterns from traces of games is a more challenging problem which is outside the scope of the thesis. However, some comments on ways to learn playing rules from traces of games will be made in chapter 9. Even for single concept learning, the number of possible Chess positions is so big, that restrictions on the pieces involved and/or a careful mechanism for selecting examples are needed for an effective inference to take place.

3.1.3 Inference methods

Intuitively an inference method is a computational process of some kind that reads in examples and outputs guesses from the hypothesis space. Several

properties can be associated with them. Gold introduced the concept of *identification in the limit* [Gol67]. That is, if M is an inductive inference method that is attempting to describe correctly some unknown rule R, when M is run repeatedly on larger and larger collections of examples of R, an infinite sequence of M's conjectures is generated, say, $h_1, h_2, \ldots$. If there exists some number m such that h_m is a correct description of R and $h_m = h_{m+1} = h_{m+2} = \ldots$, then M is said to identify R correctly in the limit on this sequence of examples.

Patterns in Chess do not have to be correct in the sense that two players can have different patterns, and yet both be competent players. Therefore, in choosing an inference method for Chess, we are more interested in having some guarantee on the 'quality' of the hypotheses that are produced. This could be in terms of compactness, simplicity, efficiency, predictiveness, etc. Human understandability plays a central role in choosing particular inference systems, which brings us back to the adopted hypothesis language.

3.1.4 Criteria of success

An important component in specifying an inference problem is what is considered as the criterion for success. Identification in the limit is one of them. However, a person using an inference method to identify an unknown rule usually cannot tell whether the method has converged. More recently, Valiant [Val84] has proposed a general criterion of correct identification of a rule from examples in a stochastic setting. The idea is that after randomly sampling positive and negative examples of a rule, an identification procedure should conjecture a rule that with "high probability" is "not too different" from the correct rule. In practice, it is desirable to have some guarantee of the quality of the hypothesis at finite stages. The most common are based on completeness and correctness. A hypothesis is *complete* if it accounts for all the positive examples[2]. A hypothesis is *correct* if it does not account for any negative example. Often, in systems where the user plays an active role in the induction process by providing adequate examples to the system, he or she determines when to stop the process. The stopping criterion for systems which generate their own examples is determined when the system cannot generate any new example that will create an inconsistency in the hypothesis. In Chess, a large number of examples is relatively easy to generate, and

[2] A formal definition in terms of coverage will be given in chapter 4.

the stopping criterion will depend on how the examples are generated and provided to the learner.

3.2 Machine learning approaches in Chess

As a favourite domain in artificial intelligence, it is not surprising to find that most machine learning techniques have been applied to Chess. A brief description of each approach with a critical discussion of their utility for learning Chess patterns is given below.

3.2.1 Propositional similarity-based learning (PSBL)

PSBL systems learn classification rules from a set of training examples. Each example corresponds to an instance of a class and is described by a set of attribute/value pairs plus the corresponding class of which the example is a member. The hypothesis language of PSBL systems is at most equivalent to propositional logic, a finite set of positive and negative examples are presented in a 'batch' mode, and the criterion of success is based on completeness and correctness (with perhaps some exceptions). Their limitations have been well recognised, namely:

1. A restricted representation, inadequate for expressing relational knowledge.
2. An inability to use background knowledge.

Several attempts have been made to use PSBL in Chess. Quinlan used ID3 [Qui83] to induce classification rules in the form of decision trees for lost in 2 and 3 ply positions for the King and Rook against King and Knight (KRKN) endgame. Shapiro [Sha87, SN82] took this approach further to classify win/draw positions in the King and Pawn against King (KPK) and King and Rook against King and Pawn on the rook's file and seventh rank (KRKPa7) endgames using a structured version of ID3. The decision trees are constructed using an information-based approach aimed to minimise the expected number of tests to classify an object. Muggleton used DUCE [Mug87] to generate new attributes and structure a rule set in the KRKPa7 endgame. DUCE takes as input a set of conjunctive productions or rules, in a propositional framework, and uses six operators to progressively transform

them by asking an oracle to validate them. In the KRKPa7 endgame, Shapiro used 36 primitive board attributes to describe each position. Similarly, the initial rule base given to DUCE for this endgame consisted of examples of the form:

$$\textit{won-for-white} \leftarrow \textit{attribute}_1 \wedge \textit{attribute}_2 \wedge \ldots \textit{attribute}_{36}$$

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 KRKN endgame [Qui83]. Shapiro reports an estimated 6 man weeks effort for the KPK endgame [Sha87]. Although systems like ID3 and DUCE were originally tested on Chess, their relative success can be attributed to a careful selection of attributes [MBHMM89]. Muggleton et al. report a recent comparative study of machine learning formalisms in Chess [MBHMM89]. The study involved Quinlan's C4 [Qui87], Bratko et al. 's Assistant86 [CKB87], Muggleton's Duce [Mug87], and Muggleton's CIGOL [MB88]. The learning problem involved deciding the legality of positions in the Chess endgame King and Rook against King. The experiments involved learning from piece-on-place attributes and learning with an extended hypothesis vocabulary over randomly generated sets of training instances (this is more extensively described in chapter 6). They concluded that the ability to produce high performance in this domain was almost entirely dependent on the ability to express first order predicate relationships. We can imagine an experiment in Chess with a propositional system that could learn 16 classification rules for a particular concept, one for each column and rank of a Chess board, instead of one or maybe two rules with variables ranging from 1 to 8. This kind of experiment shows the importance of a first order representation language which can be emphasised by imagining a Chess board with 1,000 by 1,000 squares. Trying to simplify and reduce the number of rules produced by an PSBL algorithm, by choosing more appropriate attributes, is not an easy task (e.g., Shapiro reports 55 lines of C code, using a high-level pattern library, for one of the attributes he used for a Chess endgame [Sha87]).

3.2.2 Chunks and macros

Chunking [LNR87] and Macro-learning [FHN72, Iba89, Kor85, Mor90] have been used to improve a problem solver's performance, based on the principle of storing and generalising sequences of actions to use them in the future

without search. Solutions of subsequent problems can be speeded up by treating the chunks or macros as starting primitive rules or operators and solving subsections of the problems with a small number of them. Although successful in certain artificial domains, it is not clear how to apply them in a more realistic environment, like Chess, where the problem conditions can change while solving a problem.

Campbell took an alternative approach and used chunking to group interrelated Pawns and use them in the design of a playing strategy for Pawn endgames [Cam88]. In his approach, a chunk is constructed by an *influence relation.* A Pawn P1 influences a Pawn P2 if:

- they are on the same or adjacent files and they are of the same colour
- they are on the same or adjacent files, they are of different colour, and the rank of the black pawn is greater than the rank of the white pawn
- P1 and P2 are influenced by some Pawn P3

The influence relation is used to partition a pawn position into chunks. Several domain dependent properties, obtained from databases, depth-first search, etc., are assigned to each chunk to characterise it. A pre-defined set of operators is used to move between states in a chunk space. Different moves are used to form plans and counter-plans in a similar way to AL3 [Bra82]. The mechanism for selecting meaningful chunks assignments for the Kings is directed by the plans available to each player. His system is only applicable in domains where entities can be grouped (chunked) with limited interaction between groups (chunks) and from which a planning strategy can be constructed. While he succeeds in Pawn endgames by treating in a special way Kings' interactions, his approach is clearly not applicable in general where clear-cut chunks are no longer easy/possible to construct.

More recently, Walczak [Wal91] have used a geometrical relation between pieces to construct chunks in complete Chess games. Instead of trying to define chunks with limited interaction to form plans in Pawn endgames, as in Chunker, Walczak is interested in grouping pieces to predict the opponent's moves in complete Chess games. Two pieces are *geometrically related* if at least one of them can move into the square occupied by the other piece and the two pieces are contiguous. Chunks are defined between at least two geometrically related pieces of the same colour. Following each move made by the opponent, Walczak's system, IAM, searches for all the opponent's

pieces and chunks all the related pieces. The size of the chunk is limited to a 16 square area. IAM also records the first five opening moves of the opponent. All the chunks are compared with those used in other games (IAM requires at least two traces of games of the opponent). Chunks which match other chunks (considering reflections and rotations of the board) from other games are saved. The number of times that each chunk is observed in several games is recorded. To predict the opponent's moves, IAM finds all the chunks that could be matched after a single move. The move which ends in the largest chunk which has been used the highest number of times before, is the one that it is predicted. The hypothesis is that Chess masters tend to reduce the complexity of a game position by moving to board positions that contain familiar patterns. Walczak reports to have predicted with IAM over 13% in average of the moves of Kasparov from the first half of the 1990 World Championship. IAM achievements are difficult to evaluate considering that the first 5 opening moves, which are likely to be repeated, are recorded and that its actual results account for 5.4 predicted moves in average. IAM uses a fixed relation to chunk pieces, i.e., chunks can only be constructed with pieces that are geometrically related (as defined above). It is not clear whether extensions to this definition will be needed and how it will affect its performance.

3.2.3 Explanation-based learning/generalisation

The basic idea of EBL/G [dJM86, MKKC86] is to start with a domain theory that defines a target concept. The learning process consists of repeatedly accepting a training example, applying the domain theory to prove that the example is an instance of the target concept, and then extracting the weakest pre-conditions of that proof to form an efficient chunk that provides an easy-to-evaluate sufficient condition for the target concept. This approach 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. EBL/G has been used to generalise single instances of specific plans or solutions into plan schemata that can then apply to similar problems [Ben89, Moo90, Sha89]. This approach suffers from problems similar to those of macro-learning and chunking. In complex domains like Chess, it is simply not possible to learn schemata for every possible tactic.

Several authors like, Minton [Min84], Tadepalli [Tad89], and more recently, Flann [FD89], have applied EBL/G to Chess. Minton uses EBL/G to learn simple plans from move sequences in which all the opponent's moves are forced. Tadepalli's system uses a planning strategy similar to AL3 [Bra82]. His system learns over-general plans from examples. When a new counter example appears it learns a counter plan. As the planning strategy is based in generating plans and counter plans, it tends to improve the plan's accuracy at the expense of increasing planning effort. Perhaps the most successful EBL/G system for learning concepts in Chess has been Flann's Induction-Over-Explanation (IOE) [FD89]. IOE learns from a set of examples with a more "conservative" generalisation method, which allows to learn more specific concept definitions. It has been able to derive definitions for *skewer*, *sliding-fork*, *knight-fork*, and *general-king-fork*, from a definition of *check-with-bad-exchange*. 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. The definition of such concepts is not always an easy task and defeats the purpose of the induction problem as most of the relevant knowledge needs to be given in the first place.

3.2.4 Parameter adjustment

Self-tuning methods like neural networks, genetic algorithms and simulated annealing, adjust their parameters (e.g., coefficients in algebraic expressions) over time in order to improve the system's performance. In neural networks, the back propagation algorithm, although it was a breakthrough in the application of multilayer perceptrons, has important drawbacks. Namely, long training times, sensitivity to the presence of local minima and having to know in advance the network topology: the exact number of units in a particular hidden layer, as well as the number of hidden layers. Similarly, there have been some criticisms of the use of genetic algorithms for parameter optimisation in computer Chess because of its inefficiency [vT91, vTH91].

Levinson and Snyder [LS91] report a parameter-adjustment system, Morph, which learns weighted patterns, consisting of networks of connections, from traces of games. An evaluation function is used to decide the next move by looking at all the possible next positions and combining the weights of the applicable patterns to those positions. The next move is that which ends in the least favourable position for the opponent. Each position in a

game is represented as a directed graph with nodes created for all pieces and all the unoccupied squares adjacent to the Kings, and edges constructed for attack-and-defend relationships (direct, indirect, discovered) between the pieces. Patterns are constructed as the smallest connected subgraph made up of those edges which do not appear in adjacent positions, and then augmented by adding to it all the edges adjacent to their initial set of nodes. Updates in the weights of the patterns are made after arriving at a final position. Determining the exact method by which the pattern values should be combined, i.e., tuning the evaluation function, is a difficult and critical problem (Levinson et al. report some of these problems [LS91]). 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.

3.2.5 A rationale for using a first order framework

Expressing relations between pieces, which can be naturally represented in first order logic, seems to play a key role in the definition of Chess patterns. The hypothesis is that first order patterns will be short, easy to understand, and powerful enough to play with. In addition, 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 in Chess. An obvious candidate for the background knowledge is the rules of the game.

Until recently, first order induction systems have been applied to very simple domains. Their main drawback comes from the exploration of a large hypothesis space, which has forced them to restrict it in such a way that only very simple concepts can be induced (e.g. [dRB88, Fen90, MB88, Sha81, SB86, Rou91]). Concepts in Chess can be expressed in terms of relations between pieces (e.g., threats, checks), relations between pieces and places (e.g., legal moves, distance to a particular place), relations between places (e.g., adjacent squares), etc. A reasonably varied set of concepts within a

more realistic background knowledge marks Chess as an ideal domain to test the power of first order systems. The next chapter analyses several machine learning efforts using a first order formalism, discusses their applicability to Chess, and describes a particular model of generalisation.

Chapter 4

First order learning

Evidence suggests that an adequate representation of patterns in Chess requires a relational formalism, where the exact position of pieces is not as important as their relation to other pieces and places in the board [MBHMM89]. So it is important to see if first order induction is applicable to Chess.

Section 4.2 provides a general overview of first order learning. Section 4.3 reviews a model of generalisation for first order clauses introduced by Plotkin [Plo71b]. Finally, section 4.5 shows that, although a relational language is regarded as an adequate formalism for Chess, current first order systems are still not adequate for this domain. First, some definitions from logic are presented. Some of the concepts and notation will be used in the sections to follow.

4.1 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. The negation symbol is $\neg$. A *clause* is a disjunction of a finite set (possibly empty) of literals of the form:

$$\forall X_1 \ldots \forall X_s (A_1 \vee A_2 \vee \ldots \vee A_k \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n)$$

where the A_is and B_js are literals, $\forall$ is the universal quantifier, where $\forall X$ denotes "for all X", and $X_1 \ldots X_s$ are all the variables occurring in the A_is and B_js. The following notation is equivalent:

$$A_1, A_2, \ldots, A_k \leftarrow B_1, B_2, \ldots, B_n$$

Thus, all the variables are assumed to be universally quantified, the commas in the B_js denote conjunction and the commas in the A_is denote disjunction. Clauses can also be represented as sets of literals. For instance, the above clause can be represented as $\{A_1, A_2, \ldots, A_k, \neg B_1, \neg B_2, \ldots, \neg B_n\}$. Set operations, such as set-difference and subset can then be applied between clauses. For instance, if $C_1 = \{A, B\}$ and $C_2 = \{A, B, C\}$, $C1 - \{A\} = \{B\}$ and $C_1 \subseteq C2$.

A *Horn clause* is a clause that contains at most one positive literal (e.g., $H \leftarrow B_1, B_2, \ldots, B_n$). The positive literal (H) is called the *head*, the negative literals (all B_i's) the *body*. A set of Horn clauses is a *logic program*. Literals, clauses and logic programs are well formed formulae (wff). A wff is said to be *ground* if it has no variables. An *interpretation* of a logic program consists of some domain of discourse and assignments of each constant to an element of the domain, each function to a mapping on the domain, and each predicate to a relation on the domain. A *model* of a logic program is an interpretation for which the clauses express true statements. A wff F is *satisfiable* if there exists a model for F. Let F_1 and F_2 be two wff's. 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. The *least Herbrand model* of a logic program P, is the set of ground atoms which are logical consequences of the program. We say that F_1 *syntactically entails* F_2 (or $F_1 \vdash F_2$) iff F_2 can be derived from F_1 using the set of deductive inference rules. The set of inference rules is said to be *sound* and *complete* iff $F_1 \vdash F_2$, whenever $F_1 \models F_2$. We say that F_1 is *more general than* F_2 iff $F_1 \models F_2$ and $F_2 \not\models F_1$ (every model of F_1 is a model of F_2). This definition imposes a lattice on the set of all formulae. The top element (most general) is the empty clause ($\square$) and the bottom (most specific) is the empty program.

A *substitution* $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ 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. Every sub-term within a given term or literal can be uniquely referenced by its *place*. A place

within a term or literal C is denoted by an n-tuple of natural numbers and defined recursively as follows. The term at place $\langle i \rangle$ within $f(t_0, \ldots, t_m)$ is t_i. The term at place $\langle i_0, \ldots, i_n \rangle$ within $f(t_0, \ldots, t_m)$ is the term at place $\langle i_1, \ldots, i_n \rangle$ in t_{i_0}. Let t be a term found at place p in literal L, where L is a literal of C. The place of term t in C is denoted by the pair $\langle L, p \rangle$. If C is a clause or a term and $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ a substitution, the inverse substitution $\theta_C^{-1} = \{\langle t_1, \{p_{1,1}, \ldots, p_{1,m_1}\}\rangle/v_1, \ldots, \langle t_n, \{p_{n,1}, \ldots, p_{n,m_n}\}\rangle/v_n\}$. The inverse substitution is applied by replacing all t_i at places $p_{i,1}, \ldots, p_{i,m}$ within C by v_i. Clearly $C\,\theta\,\theta^{-1} = C$. For example, if literal $L =$ likes(X, brother(X)) and $\theta = \{X/\text{john}\}$, then $L\theta =$ likes(john, brother(john)), $\theta_L^{-1} = \{$<john, $\{< 1 >, < 2, 1 >\}$>$/X\}$, where $< 1 >$ and $< 2, 1 >$ are the places where the variable X is found, and $L\theta\theta^{-1} = L$. The substitution θ is said to be a unifier of the atoms A and A' whenever, $A\,\theta = A'\,\theta$. μ is the *most general unifier* (mgu) of A and A' if and only if for all unifiers γ of A and A', there exists a substitution δ such that $(A\mu)\delta = A\gamma$.

If F_1 and F_2 are two wff's, F_1 and F_2 are said to be *standardised apart* whenever there is no variable which occurs in both F_1 and F_2. Resolution is a deductive inference rule that allows us to infer a new clause from two given clauses. $((C - \{A\}) \wedge (D - \{\neg A'\}))\theta$ is said to be the resolvent of the clauses C and D, whenever C and D are standardised apart, $A \in C$, $\neg A' \in D$, and θ is the mgu of A and A'. Resolution is used for deriving the consequences of a logical program. Given a program P, C is a logical consequence of P iff $P \wedge \neg\, C$ can be shown by resolution to derive the empty clause ($\Box$).

4.2 Inductive logic programming

Inductive Logic Programming (ILP) is a fast growing research area which combines Logic Programming with Machine Learning to induce first order logic programs from examples [Mug91a]. In inductive logic programming the system's current knowledge consists of data and background knowledge expressed as a logic program. These are assumed to be true in the intended interpretation. A formula is true if it is a logical consequence of the current knowledge. The inductive problem is to find a hypothesis, a set of clauses, consistent with the current background knowledge and capable of explaining the data in the sense that all the positive literals but no negative literals in the data are deducible from the hypothesis and the background knowledge. Given background knowledge $\mathcal{K}$ and some examples $\mathcal{E}^+$ and $\mathcal{E}^-$, a potential

induction hypothesis $\mathcal{H}$ explaining the data must be such that: $(\mathcal{K} \wedge \mathcal{H}) \models \mathcal{E}^+$ and $\mathcal{K} \wedge \mathcal{H} \wedge \mathcal{E}^-$ is satisfiable. ILP involves producing logic programs which satisfy the above conditions from examples[1]. These conditions define a search space on hypotheses which can be organised with a model of generalisation. Induction can be achieved by searching through those clauses more general than a known specialisation of a clause, or through more specific clauses than a known generalisation. These search spaces are usually organised into hierarchies, where all the clauses below (above) a particular clause are specialisations (generalisations) of that clause. In general, there are infinite ascending and descending chains within the hierarchy, constructed by conjoining and disjoining formulae. The next section introduces a model of generalisation for ILP.

4.3 Generalisation

Plotkin [Plo71b] (who did not restrict himself to Horn clause logic) was the first to analyse in a rigorous manner the notion of generalisation based on θ-subsumption. Clause C θ-subsumes clause D (denoted as $C \leq D$) iff there exists a substitution σ such that $C\sigma \subseteq D$ (i.e., there exists a substitution that makes clause C a subset of clause D). Clause C is more general than clause D if C θ-subsumes D. It is possible to have clauses that are equivalent under subsumption, i.e., $C\sigma \subseteq D$ and $D\sigma' \subseteq C$. Plotkin also investigated the existence and properties of least general generalisations (*lgg*). The *lgg* of two terms or literals is a generalisation which is less general than any other generalisation. Two terms or literals are *compatible* if they have the same predicate name and sign. The *lgg* of two terms or literals is defined for two compatible terms or literals. The *lgg* algorithm replaces all the different terms that have the same place[2] within compatible literals by new variables [Plo69, Plo71b] (see Table 4.1). Plotkin shows that any set of compatible literals have an *lgg*.

In some sense the least general generalisation of literals is the dual of the most general unifier. For example, if:

L_1 = foo(a,f(a),g(X,b),f)

[1]It is clear that $\mathcal{H}$ could be replaced by $\mathcal{E}^+$ (that is, if $\mathcal{K} \wedge \mathcal{E}^+ \not\models \mathcal{E}$ for all $\mathcal{E} \in \mathcal{E}^-$). However, in general, we look for compactness of data into general descriptions of classes of objects which can be used for classification and prediction.

[2]See section 4.1 for the definition of place.

If L_1 and L_2 are two compatible terms or literals
1. Let $P_1 = L_1$ and $P_2 = L_2$. 2. Find two terms, t_1 and t_2, in the same place in P_1 and P_2, such that $t_1 \neq t_2$ and either both have a different function letter or at least one of them is a variable. 3. If there is no such pair, then finish. $P_1 = P_2 = lgg(L_1, L_2)$. 4. Else, choose a variable X distinct from any variable occurring in P_1 or P_2, and whenever t_1 and t_2 appear on the same place in P_1 and P_2, replace them with X. 5. Go to 2.

Table 4.1: The *lgg* algorithm for terms

and
$L_2 =$ foo(Y,f(Y),g(c,b),h(g))
then
$lgg(L_1, L_2) =$ foo(Z,f(Z),g(W,b),V).

Similarly, a *lgg* of two *clauses* is a generalisation which is less general than any other generalisation. 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 $l = lgg(l_1, l_2)\}$. This extends to sets of clauses. A set of literals $S = \{L_i \mid i \in [1 \ldots n]\}$ is a *selection* of a set of clauses $H = \{C_i \mid i \in [1 \ldots n]\}$ iff L_i is in C_i $(i \in [1 \ldots n])$, and any two literals in S are compatible. The main result is that every set of clauses has an *lgg* which is not empty iff the set of clauses has a selection.

Plotkin [Plo71a, Plo71b] also introduces the notion of relative least general generalisation of clauses (or *rlgg*).

Definition 1 *Clause C θ-subsumes clause D relative to $\mathcal{K}$, denoted as $C \leq_K D$, if $\mathcal{K} \vdash C\theta \rightarrow D$ for some substitution θ.*

Definition 2 *If H is a set of clauses, C is the* rlgg *of H relative to K, if for every D in H, $C \leq_K D$, and for a C' such that $C' \leq_K D$ for every D in H, then $C' \leq_K C$.*

In general, there is no *rlgg* of two clauses relative to some background knowledge. This is true for Horn-clauses as well [Nib88]. However, Plotkin shows that *rlgg* exists for ground theories. One problem with Plotkin's definition is that one clause which refers to one concept can be more general than another clause that refers to a different concept. Under Plotkin's definition $P \leftarrow Q$ is more general than $R \leftarrow S, Q$ relative to $R \leftarrow P, S$.

A model-theoretic characterisation of θ-subsumption, called generalised subsumption, was introduced by Buntine [Bun88] for Horn clauses, with an additional condition on the heads of the clauses.

Definition 3 *Clause $C \equiv C_0 \leftarrow C_1, C_2, \ldots, C_n$, covers atom A with respect to interpretation J, iff there exists a substitution θ for C such that $C_0\theta = A$ and $C_i\theta$ is true in J for all $i \leq n$.*

Definition 4 *A clause C subsumes clause D w.r.t. program P, denoted as $C \leq_P D$, iff for every interpretation J which makes P true and every ground atom A, C covers A w.r.t. J whenever D does.*

Buntine also laid down the foundations for an algorithm to test for subsumption:

Theorem 1 *Clause $C \equiv C_{head} \leftarrow C_{body}$, subsumes $D \equiv D_{head} \leftarrow D_{body}$ (where the variables in C and D are distinct) w.r.t. program P ($C \leq_P D$) iff there exists a minimal substitution σ such that, $C_{head}\sigma = D_{head}$ and for any ground substitution θ for D which substitutes new constants not occurring in P, C or D for variables it is true that $P \cup D_{body}\theta \models \exists\,(C_{body}\sigma\theta)$.*
Proof*: see Buntine [Bun88].*

Roughly, when comparing two clauses, it is necessary to show that the more general clause can be converted to the other by repeatedly:

1. turning variables into constants or other terms
2. adding atoms to the body, or
3. partially evaluating the body by resolving some clause in the program P with an atom in the body.

Testing for generalised subsumption is only semi-decidable, i.e., termination is guaranteed only when in fact $C \leq_P D$. If the program has no recursion termination is guaranteed, similarly for a program without function

symbols. The main difference between logical implication ($P \models (C \rightarrow D)$) and Buntine's generalised subsumption ($C \leq_P D$), arises when the heads of the clauses are not compatible or when C is recursive. Roughly, $C \leq_P D$ when the body of D implies the body of C in the context of P.

While Plotkin's definition is proof-theoretic, Buntine's is model-theoretic. Buntine's definition of relative subsumption is strictly weaker that Plotkin's (see Niblett [Nib88] for a more thorough discussion). In effect, generalised subsumption (Buntine's) simplifies to θ-subsumption when there is no background knowledge.

Similarly, a least general generalisation of two clauses relative to some background knowledge, is a generalisation which is less general than any other generalisation. Buntine suggests a method for constructing *rlgg*s using Plotkin's *lgg* algorithm between two clauses as follows:

Theorem 2 *Let C and D be two clauses with disjoint variables and P a logic program. Let θ_1 be a substitution grounding the variables occurring in C_{head} to new constants, θ_2 grounding the remaining variables in C, and likewise ϕ_1 and ϕ_2 for D. If $lgg_P(C, D)$ exists, it is equivalent w.r.t. P to the $lgg(C', D')$, where $C' \equiv C\,\theta_1 \cup \{\neg A_1, \ldots, \neg A_n\}$, and $D' \equiv D\,\phi_1 \cup \{\neg B_1, \ldots, B_m\}$. Where for $1 \leq i \leq n$, $P \wedge C_{body}\theta_1\theta_2 \models A_i$, and A_i is a ground atom constructed from symbols occurring in P, C, θ_1, θ_2, and D. Likewise for each B_j.*
Proof: *see Buntine [Bun88].*

This however assumes the deduction of all the ground facts from the theory. Similarly, there is, in general, no *rlgg* between two clauses relative to some background knowledge. The problem arises from the fact that an infinite number of facts can be logically implied by the theory. Again, an *rlgg* exists when the logic program is a set of facts (i.e., for ground theories) or when it is a DATALOG program (i.e., without function symbols).

Even with a finite set of facts, the *lgg* algorithm of two clauses can generate a very large number of literals. If the length of a clause C is represented by $|C|$, then the length of the *lgg* of two clauses C_1 and C_2 can be $|C_1| \cdot |C_2|$. For example, following Buntine's method, the *rlgg* of two ground unit clauses: *member*(1,[0,1]) and *member*(0,[1,2,0]), relative to the set of ground unit clauses: *member*(1,[1]), *member*(0,[2,0]), and *member*(1,[1,0]), is the *lgg* of C_1 and C_2 defined as follows:

$$C_1 = member(1,[0,1]) \leftarrow member(1,[1]),\ member(0,[2,0]),\ member(1,[1,0]).$$

$$C_2 = member(0,[1,2,0]) \leftarrow member(1,[1]),\ member(0,[2,0]),\ member(1,[1,0]).$$

which produces,

$$member(X,[Y,Z|T]) \leftarrow member(1,[1]),\ member(X,[Z|T]),\ member(1,[1|T]),\ member(Y,[W|R]),\ member(0,[2,0]),\ member(Y,[W,0]),\ member(1,[1|R]),\ member(X,[Z,0]),\ member(1,[1,0]).$$

Although this contains the recursive definition for *member/2*, it contains many other literals as well. Several authors report clauses of thousands of atoms long for very simple concepts [Bun88, Fen90, MF90, Rou91]. Plotkin [Plo69, Plo71a] suggests the use of logical clause reduction to remove redundant literals from a clause. A literal L is logically redundant in a clause C if $\mathcal{K} \wedge C \vdash C - \{L\}$. However, this is again semi-decidable and inefficient as it requires theorem proving. The main problem is that *lgg* produces very small generalisation steps and some heuristics are required to converge faster into a solution. Furthermore, 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. The particular strategy followed in this thesis for an effective use of *rlgg* will be discussed in chapter 5 where the description of PAL is given.

The notion of generalisation described above illustrates some of the problems of current ILP systems. Some implications are that most ILP systems either use very restricted theories, or very strong constraints, which limit their applicability to very simple domains. In the next section, an overview of the different approaches taken by ILP systems is presented.

4.4 An overview of ILP systems

ILP systems are reviewed under 3 criteria: the hypothesis search strategy, the example selection, and their background knowledge. A classification of constraints used to limit the hypothesis space is given in section 4.4.4.

4.4.1 Hypothesis search: top-down vs bottom-up

ILP systems which search a generalisation hierarchy (or bottom-up) include, Marvin [SB86], Cigol [MB88], Relex [Fen90], Golem [MF90], Itou [Rou91],

and Clint [dRB88, dRB90]. Bottom-up systems start with a very specialised clause and gradually generalise it, turning constants into variables, removing conditions (literals from the body of the clause), or transforming the body of the clause using background knowledge. In general, an infinite number of facts deduced from the theory can be used to construct "very specialised" definitions, and even with finite length hypotheses, there is a large number of ways in which to generalise them.

Systems which search a specialisation hierarchy (or top-down), include, MIS [Sha81], Foil [Qui90], Linus [LDG91], and Focl [PK90]. Top-down systems start with a very general clause and gradually specialise it, replacing variables with terms, adding literals to the body, or transforming the body using background knowledge. Similarly, there is a large number of ways in which to specialise a clause.

Both approaches suffer from a combinatorial explosion in the search for hypotheses. Most systems trade efficiency for applicability by applying strong restrictions to limit the hypothesis space.

4.4.2 Example selection: user vs batch vs automatic

A characterisation of inductive inference methods can be based on the amount of information on which a system relies for its "correct" behaviour. An important component of such characterisation is the particular example selection suitable for the inference method. The efficiency of an inference method can be measured by the number of examples that it requires to converge to a correct rule. Some systems require a large number of examples to "justify" the construction of an hypothesis. Other systems might depend on a very careful example presentation. In general, the learning performance of an inference method changes with the example presentation. Selection of "good" examples, has allowed some systems to learn concepts that would otherwise be infeasible for them to learn [MB88, Rou91, SB86, Win85]. Examples can be provided by the environment, selected by an informed oracle, can be a subset of the example space, randomly generated, automatically generated by the system, etc. In some domains, examples might not be immediately available. The most common example presentations are either interactively by the user, or an example generator, or in 'batch', selected from an existing sample set (often generated by the user). In general, the termination criterion can be linked to the example presentation.

- Systems which rely for their success on a careful presentation of examples by the user, include MIS [Sha81], Cigol [MB88], Itou [Rou91], and Marvin [SB86]. The user provides the “right” examples and guides the learning process. The user is aware of the target concept and determines the criterion of success.

- Systems which accept a set of examples in batch, include Golem [MF90], Foil [Qui90], and Linus [LDG91]. The termination criterion is based on completeness and correctness (with perhaps some exceptions) of the hypotheses.

- Systems which generate their own examples, include Clint [dRB88] and Relex [Fen90]. The termination criterion is either determined by the user or when the system is unable to generate a new example that will produce an incorrect or incomplete hypothesis.

4.4.2.1 Automatic example generators

This section deviates so as to look in general into some automatic example generators. Systems which induce concepts from a batch of examples often require an “artificial” generation of them by the user as not all domains have them readily available for the process. Systems which accept examples interactively by the user are often highly dependent on an “adequate” sequence selection. This dependency, or *hidden knowledge*, requires a good understanding of the system’s internal characteristics and calls into question its learning capabilities. It is believed that “careful” experiment selection is more effective for concept formation than a random experiment selection (e.g. see [Ang88, Lin91]). Experimentation (or active instance selection) has been employed in several machine learning systems (e.g., [CG87, DB83, Fen90, Len76, PK86]) to reduce the dependency on the user and guide effectively the learning process.

Example generators in concept learning provide two basic functions: test the hypotheses and gather information to constrain the hypothesis generator. In general, over large hypothesis spaces, “clever” experimentation is required to search this space efficiently. Several strategies have been suggested. Some systems, like AM [Len76], LEX [MUB83], or PET [PK86], provide a hierarchy of concepts. Selection of examples involves choosing instances of concepts which have a relation in the hierarchy with the current hypothesis (or concepts involved in the hypothesis). A considerable amount

of knowledge about the domain is required to provide an initial hierarchy of concepts and examples, and not all the domains can be easily structured in this way.

Feng [Fen90] provides a theoretical basis for choosing a new example based on information theory. His next-best-verification algorithm chooses the next example as 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. The definition of a sequential number is not easy to do as several “sequences” along different “dimensions” can exist.

Clint [dRB88] starts with the most specific function-free clause constructed in the concept description language that covers a positive example w.r.t. the current background knowledge (the starting clause). It then constructs new clauses, by incrementally selecting and deleting subsets of literals from the original clause, which have a greater coverage of positive examples without covering any negative. To validate such new clauses, it generates examples which are covered by the clauses, but not by the original clause. If the example is validated as positive by an oracle, the new clause is kept as the current hypothesis and the process continued. Otherwise, a new subset of literals is selected (i.e., a new clause is constructed). The process stops when no more literals can be eliminated from a clause without covering a negative example. A similar example generation strategy has been used by other systems such as Marvin [SB86]. In a clause of length N Clint’s original search is of 2^N subsets, however, in practice not all the subsets need to be considered. If a clause covers a negative example when a subset S_1 of literals is eliminated from the current clause, then any superset of S_1 is no longer considered as a candidate for elimination. The space is also reduced as literals are eliminated. The number of examples generated by Clint depends on the number of literals in the starting clause. Even with function-free clauses, and with a restricted hypothesis language and background knowledge definitions, the starting clause can be huge and difficult to compute.

In [SF86], Subramanian and Feigenbaum show that a substantial improvement in the number of generated examples can be achieved on factorable concepts by considering each component independently. A concept is factorable if credit or blame can be assigned independently to each component of the concept. The components of a concept expressed as a Horn clause are the different literals in its body. Generating new examples correspond to changes in the arguments used in the head of the clause. Different ar-

guments affect different literals. In this context, a concept is factorable if changes in some arguments of the head of the clause affect some literals in the body independently from changes in the other arguments of the head. In general, knowing that credit or blame can be assigned independently to each component is not known/possible. In particular, several concepts in Chess like *fork*, *pin* or *skewer* cannot be factored into independent components as changes in the description of one of the pieces will affect the others.

Ruff and Dietterich [RD89] report some experiments using Boolean truth tables as hypotheses with different example selection strategies. They argue that there is no essential difference between an example generator that uses a "clever" (although computationally expensive) strategy which divides the hypothesis space in half and an example generator that guarantees to eliminate at least one hypothesis, or even a simple example generator that randomly selects examples (without repeating experiments). They did not provide any evidence that their results could scale or generalise. This is not applicable for a domain like Chess, where some concepts can cover a relatively small subset of the example space. In such cases, a random strategy can prove to be useless as a large number of negative examples can be generated before a positive example is selected (examples will be given in chapter 6). It is thus imperative to have a guided experimentation strategy, however, selecting the "best" example can be infeasible. Chapter 5 introduces an automatic example generator for Chess which is guided by the concept definition.

4.4.3 Background knowledge: clauses vs facts

One of the main arguments against existing propositional systems is based on their lack of use of background knowledge. Background knowledge is important as it allows us to represent examples in a simple way and can be used to learn several concept definitions. In some ILP systems, the background knowledge (generally represented as Horn clauses) is carefully selected by the user. Recent systems (i.e., Foil, Focl, Linus, and Golem), have represented their background knowledge with a set of ground facts. ILP systems with ground and non-ground background knowledge are analysed below.

- All ILP systems which represent their background knowledge as non-ground clauses suffer from search problems, need a careful selection of background knowledge definitions (and sometimes of examples), and have been applied to very restricted domains. Their main problem

arises from the combinatorial explosion in the search for hypotheses, which is severely affected by the size and characteristics of the background knowledge. Their main advantages include its concise representation, they can incorporate in principle any previously known concepts without requiring any transformation process, and the new learned concepts are immediately accessible to the system in the next inductive cycle.

- In trying to improve applicability, recent systems have replaced the representation of the background knowledge with a set of ground facts. This approach has been used in larger domains [Mug91a] as it eliminates some problems of previous systems. In particular, for bottom-up systems, *rlgg* exists for ground theories. Ground theory systems require a careful selection of a "representative" subset of ground facts in advance, which is often tailored to the nature of the examples over which the induction is made. A huge memory space is sometimes needed to store ground theories and none of the current systems is used in an incremental way.

There is a fundamental space-time vs applicability tradeoff when choosing a particular background knowledge representation. Systems which use ground theories are limited by the number of facts that they can store. In some domains, generating and storing facts to achieve the required results can be impractical, and sometimes a reformulation of the background facts is required to keep the number of facts down into workable conditions. Bratko et al. [BMV92] report this problem when using Golem to induce qualitative models and is further described in chapter 8. In general, the generation of appropriate background facts can be a time-consuming process, highly dependent on the examples. This memory-greedy and time-consuming generation process is compensated by an efficiency gain over systems which use non-ground theories and has been applied to larger domains. The combinatorial hypothesis search of non-ground theory systems has limited strongly their applicability. Regardless of the approach, problems can only be magnified when adding extra background knowledge and in general only a very limited number of background concepts have been used by ILP systems.

4.4.4 Constraints on the hypothesis space

Structuring the hypothesis space with a generalisation/specialisation hierarchy provides only a guideline to ILP systems and different heuristics have been used to restrict the space and provide a notion of "relevance" between competing background knowledge. A suitable selection of background knowledge by the user provides an implicit notion of relevance to the system; however this is not enough to achieve practical results. The following constraints have been used by several ILP systems.

- *Functional Restriction*: The system is provided with information which states which arguments are determined (output arguments) in a predicate if the rest are known (input arguments). This information can be used to form directed graphs which link input/output arguments and guide the construction of hypotheses (e.g., [MF90, Qui90, Rou91]). The linkage between input and output arguments can be constrained as well by using typed variables, where input/output links can be formed only between arguments with the same type.

- *Variable Connection Restriction*: Consider only clauses in which all variables appear at least twice in the clause (e.g., [dRB88, dRB90, MF90, Rou91]) or introduce a new literal to the body only if at least one existing variable is used (e.g., [Qui90]).

- *Rule Schema Restriction*: Construct hypotheses only from a class of clauses defined through rule models (i.e., consider only hypotheses which "match" a particular rule schema [dRB92, Mor89, Thi89, Wro89] or with particular "refinement" operators as in MIS [Sha81].

- *Information Content Restriction*: Construct hypotheses which produce a compression in information content w.r.t. the examples (e.g., [MB88]), hypotheses with the Minimum Description Length (MDL) [MSB91], or guide the hypothesis search with a measure of information gain based on the discrimination between positive and negative examples (e.g., [Qui90]).

- *Explicit Relevance*: Use additional predicates to determine which background knowledge predicates are relevant to the current hypothesis (e.g., with determinations [RG88] or with integrity constraints [dRBM91]).

- *Initial Clause Restriction*: Provide an initial clause and add only literals to the body which can be deduced from the body of the clause and the background knowledge (e.g., [Rou91, SB86]).

4.5 A review of some ILP systems

Most ILP systems have been used to infer simple concepts, such as list definitions like *member* or *append*, the concept of an *arch*, *family relations*, etc., with restricted background knowledge (e.g. [dRB88, MB88, Rou91, SB86, Sha81]). By contrast, concepts in Chess which can be used in a playing strategy, might involve background definitions for threats, checks, legal moves, distances to places, etc. These concepts comprise a more realistic background knowledge with a larger hypothesis space from which relatively varied concepts can be induced. The suitability for learning concepts in a domain like Chess, is reviewed below for ground and non-ground theory systems.

4.5.1 Non-ground theory systems

Most ILP systems under this category require a very careful selection of examples, tend to produce over-generalisations [MB88, SB86], or are lost in the combinatorics of specialisations [Sha81]. MIS [Sha81], Cigol [MB88], and Itou [Rou91] are briefly reviewed below.

4.5.1.1 MIS

MIS [Sha81] is a top-down system which starts with a set of positive and negative examples (provided by the user) and one or more *refinement operators* to specialise clauses. MIS starts with the most general theory and stops when all the positive examples are deduced from the theory without deducing any of the negatives. MIS is an interactive system where examples are provided by the user. At each step, while MIS can derive a false fact given some resource bounded computation, it applies the contradiction backtracing algorithm (see below) to find a false clause and remove it from the hypothesis. When MIS cannot derive a true fact given some resource bounded computation, it generates a new hypothesis (i.e., adds a new clause to the hypothesis) using a refinement operator (see below). MIS continues

```
Set T to {□}

repeat
    read the next fact
    repeat
        while the conjecture T is too strong
                (i.e., it implies a negative example)
                apply the contradiction backtracing algorithm,
                and remove from T the refuted hypothesis.
        while the conjecture T is too weak
                (i.e., it does not imply a positive example)
                add to T refinements of previously
                refuted hypotheses.
    until the conjecture T is neither too strong
          or too weak (w.r.t. the facts read so far).
forever
```

Table 4.2: MIS algorithm

until neither of the while loops is entered (see Table 4.2). Roughly, Q is a refinement of P if P implies Q and $size(P) < size(Q)$, where *size* is a function that maps from clauses to natural numbers. A refinement operator is said to be complete over a set of sentences, if we can obtain all the sentences by successive refinements from the empty clause. A refinement operator induces a partial order over the hypothesis language. This can be 'illustrated' by a refinement graph where every node in a lower level is a specialisation of previous upper levels. Given a refinement operator complete for the hypothesis language, MIS traverses its refinement graph until a complete axiomatisation for the target concept represented in the language is found.

The backtracing algorithm starts from a derivation tree of a refutation proof[3] and interactively tests the terms resolved upon by constructing ground atoms for each term and asking the user to validate them. The algorithm is only applied when a logic program succeeds but with a wrong answer. If the logic program fails, MIS can only add new clauses.

MIS was applied to simple domains with different refinement operators.

[3] See section 4.1.

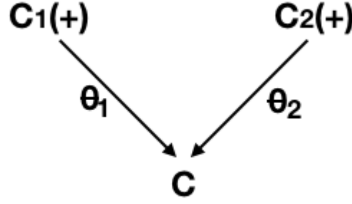


Figure 4.1: A resolution step

In general, as the background knowledge grows, so do the combinatorics of its search space. Because of this, MIS is unable to learn concepts with a large background knowledge within reasonable time limits.

4.5.1.2 CIGOL

Cigol [MB88] constructs Horn clauses from examples using three operators, *truncation*, *absorption*, and *intra-construction*. Cigol was designed around the idea of inverting resolution. Resolution allows us to infer a new clause C from two clauses C_1 and C_2 with literals L_1 and L_2, with a most general unifier (mgu) (see section 4.1). If $\neg L_1$ and L_2 are the literals of C_1 and C_2 with an mgu θ, θ can be expressed as $\theta = \theta_1\theta_2$, and the resolvent clause C will be:

$$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2 \tag{4.1}$$

Absorption constructs C_2 given C and C_1, assuming that the literal resolved on is positive in C_1 and negative in C_2 (see Figure 4.1).
Rearranging equation 4.1, we get,

$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{L_2\} \tag{4.2}$$

Cigol provides a partial solution for this equation, by considering only the case where C_1 is a unit clause (i.e., $C_1 = L_1$). This means that the operator can only be applied when one of the clauses resolved on is a unit clause, which in practice corresponds to the examples provided by the user. Noticing that $\neg L_1\theta_1 = L_2\theta_2$, and assuming that $C_1 = L_1$, we obtain:

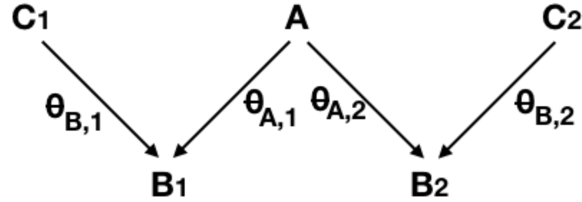


Figure 4.2: Two resolution steps with common clause A

$$C_2 = (C \cup \{\neg L_1\}\theta_1)\theta_2^{-1} \tag{4.3}$$

From equation 4.3 there are 2 sources of indeterminacy (since $C_1 = L_1$), namely θ_1 and θ_2^{-1}, where θ_2^{-1} is an inverse substitution in the domain of C_2 defined as a list of terms, the places where they should be replaced, and the variables to replace them with (see section 4.1).

Cigol needs to identify which clauses from the current examples are C and C_1. Then it has to construct a substitution in the domain of C_1 and an inverse substitution in the domain of C_2.

Intra-Construction works by combining two resolution steps back-to-back (see Figure 4.2). If we assume that C_1 and C_2 resolve on a common literal L within A to produce B_1 and B_2, this operator constructs A, C_1, and C_2 given B_1 and B_2 with L negative. Since the common literal L in A is resolved away, the clauses A, C_1 and C_2 can contain a predicate symbol not found in B_1 and B_2. In this way new predicates are introduced by Cigol.

The third operator in Cigol is *truncation*. This operator constructs an *lgg* of a set of unit clauses (see section 4.3).

Cigol only applies *truncation* to new unit clause examples. It then applies the inverse-resolution operators (*absorption* and *intra-construction*) and selects that which produces the best compression on the information content. The information content of each clause is assigned by the size of the syntactic objects which compose it. Redundant clauses are removed using Buntine's redundancy algorithm [Bun88]. The algorithm is summarised in Table 4.3.

Roughly, the *absorption* operator (first introduced in Marvin [SB86]) replaces in the body of one clause the body of another clause by its head. The literals that have been replaced are lost and can have harmful effects if the wrong choice is made for the clauses involved or if different orders of *absorptions* are made. For example, if we have in the theory:

```
while examples are provided by the user
      apply truncation (as a filter)
      apply intra-construction and absorption
      select one which produces the most information
            compression contents
      ask the user for confirmation
      reduce clauses
```

Table 4.3: Cigol algorithm

$A \leftarrow B, C.$
$D \leftarrow C, E.$

with the following example:

$F \leftarrow B, C, E.$

absorption produces with the first clause:

$F \leftarrow A, E.$

and with the second clause:

$F \leftarrow B, D.$

Literals C or E are removed (depending on the clause which is chosen) thus preventing one more *absorption* with the other clause. The search space of Cigol grows as the background knowledge grows, its operators work only with unit clauses, and the user (oracle) must be very careful in the sequence of training examples. All of which makes it only applicable to very limited domains.

4.5.1.3 ITOU

Itou [Rou91] is a recent ILP system with some improvements over systems like Marvin [SB86] and Cigol. Itou starts with a clause provided by the

while examples are provided by the user
  accept a new example by the user
  if the example is not explained by the theory
    perform exhaustive elementary saturation
    repeat until the user does not accept a generalisation
      perform truncation (remove literals from the saturated body).
    perform *intra-construction*

Table 4.4: Itou algorithm

user and performs an exhaustive *elementary saturation* process to construct a starting clause, which is later generalised by removing literals.

Elementary saturation is defined as follows. Given $H_1 \leftarrow B_1$ and a background knowledge clause $H_2 \leftarrow B_2$, a new clause $H_1 \leftarrow B_1 \wedge H_2\theta$ is produced iff $B_2\theta$-subsumes B_1 (i.e., $B_2\theta \subseteq B_1$). This is equivalent to doing the *absorption* operator but without removing the resolvent literals from the clause, i.e., avoiding some of the problems of Marvin and Cigol. Computing exhaustive elementary saturations is equivalent to *rlgg* [Mug91a] and may never stop. Even in domains with finite theories, this approach can produce very large clauses, which has constrained the applicability of Itou to simple domains.

The generalisation process, also called truncation, drops literals from the saturated clause. For each possible generalisation, the user is asked for confirmation. Truncation in Itou can remove any literal in the clause, however, preference is given to literals which introduced other literals during the saturation process. The saturated clauses are reduced using functional and variable connection restrictions (see section 4.4.4). Itou also uses the *intra-construction* operator of Cigol to reformulate the theory. The algorithm is summarised in Table 4.4.

Itou uses a “flattening” and “unflattening” strategy to transform clauses with function symbols to clauses without function symbols and vice versa (this has been previously done manually in Marvin). To flatten a clause, Itou follows the following process. For every functor of arity n, $f(t_1, \ldots, t_n)$, that appears in a clause:

- Introduce a new predicate symbol f_p of arity $n+1 : f_p(t_1, \ldots, t_n, X)$. The new predicate f_p is defined by $f_p(t_1, \ldots, t_n, X) \leftrightarrow X = f(t_1, \ldots, t_n)$.

- For each occurrence of a term $f(t_1, \ldots, t_n)$ in a clause C of the logic program P, replace it by a variable X, with the predicate $f_p(t_1, \ldots, t_n, X)$ in the body of C.

For example:

$member$(a,$cons$(a,nil)) ←

is flattened to:

$member$(X,Y) ← $cons_p$(X,Z,Y), a_p(X), nil_p(Z).
$cons_p$(X,Y,cons(X,Y)) ←
a_p(a) ←
nil_p(nil) ←

In the unflattening procedure, for each flattened predicate $f_p(t_1, \ldots, t_n, X)$ in the body of clause C, replace all the occurrences of X by the functional term $f(t_1, \ldots, t_n)$.

To improve efficiency in Itou, all the occurrences of the same term are replaced throughout the clause by the same variable. This means that the system is very sensitive to coincidences of terms in the examples, as the same variable can be assigned to terms which refer to different objects. In Chess, several repeated pieces can be at the same rank or file and distinction between them is clearly desirable during the generalisation process. Itou performs saturations between literals of the example clause and the background knowledge. Once a literal is added to the body, it can be used to generate new literals. With many background knowledge definitions it is difficult to provide an initial clause that will neither initiate a large production of literals, nor fail to incorporate the literals required for the concept definition. This makes Itou very dependent on the examples provided by the user and the “complexity” of the background knowledge.

Even after applying constraints, the size of the saturated (and flattened) clauses can be very big (specially with several function symbols), thus the possible truncations (i.e., requests for the user’s confirmation) can be huge. The system also relies heavily on the oracle to validate each proposed generalisation, thus guiding the system towards the target concept.

The almost unrestricted generation of literals from the background knowledge has made Itou (like most ILP systems) applicable only to very restricted domain theories from which the induction process can be more easily controlled.

4.5.2 Ground theory systems

Recently, systems have used a finite set of ground unit clauses as background knowledge to improve the use and selection of the background knowledge. Within this approach, probably the best known systems are Golem [MF90] and Foil [Qui90]. Both systems are reviewed below.

4.5.2.1 GOLEM

Golem [MF90] follows the *rlgg* generalisation method suggested by Buntine (see section 4.3) with ground theories and functional and variable connection restrictions (see section 4.4.4) to limit the size of the hypotheses. The examples are processed in batch and the termination criterion is based on completeness and correctness. Golem incrementally constructs clauses which cover as many positive examples as possible without covering any negative until all the positive examples are covered. The algorithm is summarised in Table 4.5.

Once a generalisation is found, the clause is reduced by removing literals from the body of the clause until no more removals can be performed without covering a negative example.

In domains like Chess a very large number of facts can be deduced from some background concepts (e.g., a definition of threat between two pieces, can imply over 20,000 facts). Background knowledge in Golem is replaced by ground facts and although Golem has been able to run with up to 20,000 background facts, it starts to slow down after storing 10,000 facts. Generating all the background facts can be a time-consuming and tedious process, and it is not always easy to know what to generate unless the training examples are known in advance.

Although Golem has been applied to some real world problems [Mug91b], it is unable to learn non-determinate clauses, that is clauses whose literals are not completely determined by instantiations of the head. Because of this, a large number of Chess concepts that are inherently non-determinate, like attacks by discovery, one-ply threats, etc., which can be described in terms of a non-deterministic concept, like piece movement, are not learnable by Golem. In general, Golem's restriction can require to redefine the background knowledge in terms of deterministic concepts only. This is not always easy/possible to do, specially in domains like Chess, where by nature legal moves (as well as many other concepts) are non-deterministic. The non-deterministic nature

- Given a set of ground unit clauses consisting of: positive and negative examples, and background knowledge.
- **repeat**
  - Take a random selection of positive examples and perform *rlgg* between pairs of examples using the background facts (i.e., take the *lgg* between two clauses, the head being the examples and the body the background facts).
  - Select the *lgg* with the greatest cover over the positive examples consistent with the negative examples (GC).
  - **while** increasing cover
    - Perform *rlgg*, between GC and a positive example, which is consistent with the negative examples.
    - remove the positive examples covered by the hypothesis
- **until** all the positive and no negative examples are covered

Table 4.5: Golem algorithm

of the game is part of the reason why Chess is an interesting and challenging game. Bratko reports similar problems with Golem when learning qualitative models[4] (see also chapter 8).

4.5.2.2 FOIL

Foil [Qui90], is a top-down system that uses an information gain measure to support an specialisation on the basis of its ability to discriminate between positive and negative examples. Similarly to Golem, Foil starts with a set of ground unit "clauses", representing positive and negative examples, and background facts, and incrementally learn clauses until all the positive without any negative examples are covered by the clauses. To understand Foil more clearly, we need to introduce some definitions. A *tuple* is a finite sequence of constants. A tuple satisfies a clause if there is a mapping of the variables of the head of the clause onto the tuple and an extension of all variables in the body into constants satisfying the body. Foil starts with some positive and negative tuples satisfying a target concept, and a very general clause, which is gradually specialised by adding literals to its body. Foil must choose the predicate name and the variables (which we can called a variablisation of the predicate) to use in the literal to be added. If the added literal uses only existing variables of the current clause, then the new set of positive and negative tuples is a subset of those tuples which satisfy the additional predicate. If a new variable is introduced by the new literal, the tuples are extended to include values for that variable. This is done automatically in Foil. That is each tuple represents a value assignment to all bound variables in the clause. The value assignment (i.e., positive or negative) of each tuple is taken from the original tuple assignment.

Each literal in the body of the clause takes one of the four forms: $X_j = X_k$, $X_j \neq X_k$, $P(V_1, V_2, \ldots, V_n)$, or $\neg P(V_1, V_2, \ldots, V_n)$, where the X_i's are existing variables, the V_i's are existing or new variables, and P is some relation. New literals in Foil must contain at least an existing variable. Foil uses an information gain metric to decide which literal to add. This metric is defined as:

$$Gain(literal) = T^{++} * [\log_2 (\frac{P_1}{P_1 + N_1}) - \log_2 (\frac{P_0}{P_0 + N_0})] \qquad (4.4)$$

[4]Personal communication.

- Given a set of positive and negative tuples, and background knowledge tuples.
- **repeat until** all positive tuples are covered.
  - set current clause to be most general predicate head with an empty body.
  - **repeat until** no negative tuples are covered.
    * compute the information gain of all the possible single literals that can be added to the current clause
    * select that literal with the most information gain
    * add the literal to the body of the current clause
    * remove the positive tuples that are satisfied by the new clause.

Table 4.6: Foil algorithm

where P_0 and N_0 are the number of positive and negative tuples before adding the literal to the clause, P_1 and N_1 are the number of positive and negative tuples after adding the literal to the clause, and T^{++} is the number of positive tuples before adding the literal that satisfy the new literal. The algorithm is described in Table 4.6.

Foil's information gain heuristic does not guarantee to find a solution when there are several possible next literals that have approximately equal gain. Because of this, Foil is unable to learn the definition of *quicksort*[5], as the background facts of *partition*, which are required for the construction of the clause, are unable to discriminate between positive and negative examples (i.e., provide a positive value for the information gain heuristic). With its information gain heuristic, Foil can make locally optimal decisions but globally undesirable choices. This problem recurs throughout a large class of concepts. Foil has no function symbols and its performance is affected by the number of arguments in the target concept[6]. With its dependency on the

[5] A new implementation of Foil, which uses a primitive back-up facility (called Foil2), has been able to learn quicksort, but only after a careful selection of the background facts and examples. Foil2 will be further discussed in chapter 6.

[6] A. Srinivasan reports running Foil for two days with a predicate of nine arguments

number of examples, Foil can change its hypothesis if it is provided with the same positive examples twice. Similar problems are reported by I. Bratko when using Foil for qualitative reasoning (personal communication). As with Golem, a large time needs to be devoted to the definition of background facts.

The longer the clause and the larger number of variables required to define a concept, the more difficult it is for a top-down system like Foil to learn it. Depending on the background knowledge and example representation, concepts in Chess which are powerful enough to be used in a playing strategy may well be several literals long involving a large number of arguments. Concrete examples of Chess concepts will be given in chapter 6.

4.5.3 Summary

Chess has been an interesting and challenging domain for concept learning. Evidence suggests that high performance of concept learning systems for this domain depends on the ability to represent relational concepts. ILP is a fast growing research area which provides the mechanism and language which should be able to represent and learn patterns necessary for play simple endgames. We have seen that the applicability of ILP systems has been mainly restricted to simple concepts over limited background knowledge. Concepts in Chess, which are powerful enough to be useful when playing, may require a more realistic background knowledge, involving definitions for several concepts like legal moves, check mates, etc. This rules out the applicability of most ILP systems. In particular, systems which use non-ground theories are lost in the complexities of the search. Their almost unrestricted derivation of facts from the background knowledge makes them inadequate in the presence of several background knowledge definitions. Their advantages include a compact background knowledge representation and their ability to use newly learned concepts in the next inductive cycle. Ground theory systems have widened the ILP applicability by improving the selection and use of background knowledge. However, they depend on a careful definition of background facts and examples. These systems, however, have made a significant performance improvement over non-ground theory systems, and a more detailed analysis involving Golem and Foil will be made in chapter 6 with a simple concept in Chess. First, an ILP system using non-ground theories

---

(personal communication). We have noticed similar problems with predicates of six arguments.

and capable of learning concepts in Chess is described in the next chapter.

Chapter 5

PAL

This chapter describes a first order learning system, called PAL[1], capable of learning patterns in Chess. PAL uses a constrained relative least general generalisation algorithm (*rlgg*) as a generalisation method over non-ground theories, avoiding the process of carefully pre-defining a set of background facts. A non-recursive representation language is used to derive a limited number of relevant facts from the background knowledge. The generalisation method is coupled with an automatic example generator, thus reducing the user's intervention and guiding the learning process. The automatic selection of examples is guided by the current concept definition in a structured example space. Examples are characterised as descriptions of Chess positions, and unlike other systems the exact arguments of the target concept are not specified in advance. Chapter 6 shows how PAL is able to learn several Chess concepts. Chapter 7 describes a strategy, constructed with patterns learned by PAL, that plays correctly a simple endgame. Chapter 8 shows how PAL can be used in other domains by learning a qualitative model of a simple dynamic system.

Section 5.1 describes the characteristics of PAL and illustrates its basic behaviour with a simple example. Section 5.2 describes in detail the generalisation method and the heuristics used for an effective use of *rlgg*. Section 5.3 discusses the characteristics of the automatic example generator and how it is used to guide the learning process. Finally, the learning algorithm is summarised in section 5.4.

[1] Patterns and Learning. Previous descriptions of PAL are given in [Mor91b, Mor91a].

5.1 Introduction

This research is oriented towards an effective automatic acquisition of Chess which can be used for playing from simple example descriptions together with the rules of the game. PAL is first characterised in terms of its example presentation, representation language, and generalisation method. An outline of the learning algorithm is given and a simple example is provided to illustrate the system's behaviour.

5.1.1 Set of examples and presentation

Examples in PAL consist of a set of ground unit clauses describing each piece on the board. The arguments used in the target concept definition and the pieces involved in it 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 the board described by position 1. Other pieces in a board position can be described in the same way. In general, other descriptions can be used as well, some of which are given in chapter 6. Descriptions of board positions in terms of the pieces involved allows a simple and natural way in which to represent examples which does not depend on the user's knowledge of the final form of the target definition. This contrasts with other learning systems, and in particular with propositional learning algorithms where most of the programming time has been devoted to a careful selection and definition of a complete set of "relevant" attributes in which to describe the examples (e.g., [Qui83, Sha87]). For example, Shapiro defined 31 attributes to describe examples for the induction of a decision tree to classify win/draw positions for the King and Pawn against King (KPK) endgame [Sha87]. Defining such attributes is not an easy task. Shapiro comments:

> "It took about six man-weeks to complete this portion of the work described [define the 31 attributes for the KPK endgame]. At least half of this time was spent considering special cases for the two subproblems *rookpawn* and *get-to-mainpatt*. This was due to the considerable difficulty found with the iterative refinement of the attributes. When there were only one or two classes of clash the choice of refinement was critical. Changing the definition of

> an attribute to eliminate one clash caused other clashes[2] with different attributes."

A larger and often obscure set of attributes is required for more difficult endgames (e.g., Shapiro's attributes for the KPa7KR endgame [Sha87] and Quinlan's attributes for the KRKN endgame [Qui83]) in which clashes of classes are more likely to occur.

Many learning systems depend on an appropriate selection of examples by the user. A "good" selection of examples is not always easy to provide and it is difficult to assess the robustness of systems which depend on it. In PAL, an automatic example generator is used to select the examples and guide the learning process. The example space is structured into classes of examples and the selection is guided by the current concept definition. Details of the example generator are given in section 5.3.

5.1.2 Hypothesis language and background knowledge

The hypothesis language and background knowledge representation must be able to express patterns (relations between pieces and places) and 'recognise' instances of those patterns from descriptions of board positions. A pattern definition is defined as a non-recursive Horn clause with the following format:

$$Head \leftarrow D_1, D_2, \ldots, D_n, F_1, F_2, \ldots, F_m. \tag{5.1}$$

where,

- *Head* is the head of the pattern definition. Instantiations of the head are regarded as the patterns recognised by the system.
- The D_is are "input" predicates used to describe positions (e.g., *contents*/4) and represent the pieces which are involved in the pattern.
- The F_js are instances of definitions which are either provided as background knowledge or learned by PAL, and represent the conditions (relations between pieces and places) to be satisfied by the pattern.

[2]A *clash* is said to occur if two or more examples are described with the same attribute values but with different class values.

PAL starts with some pattern definitions and incrementally learns new patterns from descriptions of Chess positions. Details of how a particular pattern is constructed from these descriptions are given in section 5.2.

A uniform representation language is used for the hypotheses and the background knowledge, thus previously learned concepts are immediately accessible to the system without requiring any transformation process.

5.1.3 Generalisation method

PAL is a bottom-up learning system which starts with a very specialised concept definition and gradually generalises it with new examples. Its generalisation method is based on the *rlgg* algorithm suggested by Buntine [Bun88] and discussed in chapter 4. PAL uses a restricted background knowledge representation, such that given a finite number of pieces and pattern definitions, only a finite number of facts can be derived from them. Additional heuristics are used to limit the size of the hypotheses and increase the generalisation steps. Details are given in section 5.2.

5.1.4 Algorithm

An outline of the algorithm used by PAL can now be described. Given a set of background knowledge definitions (pattern definitions) and a description of a Chess position (a set of ground unit clauses), PAL constructs an initial clause with facts (instance of patterns) derived from the example description and the background knowledge definitions. An automatic example generator is used to provide new examples (validated by the user) until a termination criterion is met. Each new positive example is used to derive new facts from the background knowledge and construct a new clause. General descriptions of pattern definitions are constructed by following a generalisation process between clauses. The basic algorithm is outlined in Table 5.1 and illustrated in Figure 5.1. A more detailed description of PAL is given in the sections to follow. In particular, section 5.2 discusses how to *construct* a clause from an example description and the pattern definitions and how to *generalise* between two clauses, and section 5.3 describes how to automatically *generate* new examples.

given:
 	
• background knowledge definitions (*BK*)
 	
• a description of a Chess position (*EX1*)
 	
construct an initial definition (*Defn*) with facts derived from *BK* and *EX1*
 	
while a *stopping criterion* is not met
    generate a new example description (*NEx*)
    if the example is positive
      construct a new definition (*NDefn*) with facts derived from *BK* and *NEx*
      Set *Defn* = *generalisation* between *Defn* and *NDefn*
    else continue
output *Defn*

Table 5.1: PAL algorithm

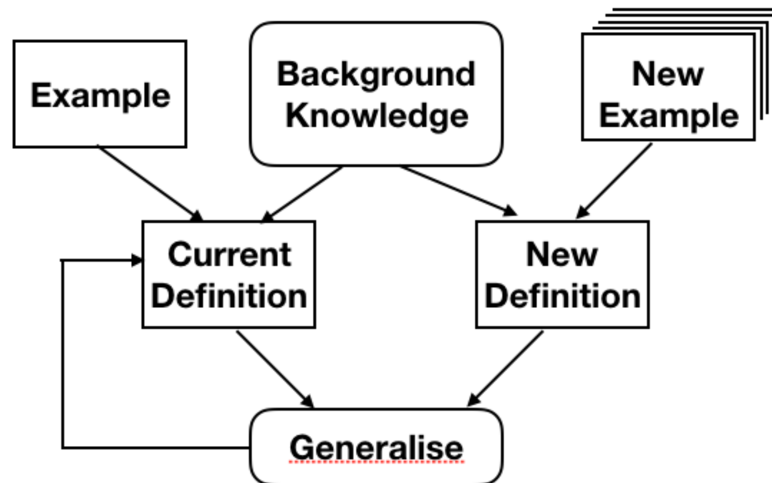


Figure 5.1: Outline of the algorithm

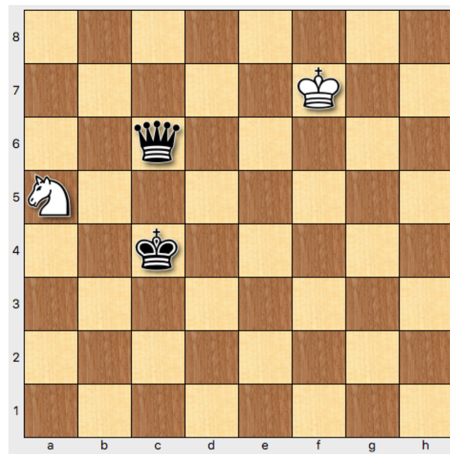


Figure 5.2: Example position

5.1.4.1 An example case

First, a realistic case, outside the scope of current ILP systems[3], is presented to illustrate the system's behaviour. PAL uses the following primitive representation. Positions are described by *contents*(*Side*,*Piece*,*square*(*File*,*Rank*),*Pos*) atoms, where *Side* is *white* or *black*, *Piece* is *pawn, knight, bishop, rook, queen*, or *king*, *square*/2 represents the position of each piece, where *File* and *Rank* are 1,2,..., or 8, and *Pos* is the example position[4]. Suppose that we want to learn the definition of a special kind of *fork* in Chess, where a piece is threatening another piece and at the same time checking a King. PAL is provided with a description of a position where this kind of pattern exists, but it is not told what the concept to learn is or which are the relevant arguments of the target definition. If the initial example is described as follows (this is illustrated in Figure 5.2):

contents(black,king,square(3,4),pos1).
contents(black,queen,square(3,6),pos1).
contents(white,king,square(6,7),pos1).
contents(white,knight,square(1,5),pos1).

and the background vocabulary consists of:

[3]The applicability of systems like Golem and Foil will be further discussed in chapter 6.
[4]Other representations can be used, some of which are given in chapter 6.

contents(*Side,Piece,Place,Pos*):
    Describes the position of each piece in a position.
sliding_piece(*Piece,Place,Pos*):
    Piece in Place is Queen, Rook or Bishop.
other_side(*Side1,Side2*):
    Side1 is the opponent side of Side2.
legal_move(*Side,Piece,Place,NPlace,Pos*):
    Piece in Place can move to NPlace.
in_check(*Side,Place,OPiece,OPlace,Pos*):
    King in Place is in check by OPiece in OPlace.

PAL uses the example description with the above background vocabulary to derive (recognise) a set of atoms (patterns) and construct a plausible definition. Background definitions, either provided by the user or learned by PAL, assume that there is an implicit example description. For instance, consider the following definition of *in_check*/5[5] (a complete description of all the background knowledge definitions is given in appendix G):

```
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)
```

This definition gets instantiated only if there is an example description with a King at *KPlace* and an opponent's piece *OPiece* at *OPlace* with a piece move to *KPlace*. This will be further discussed in section 5.2.2.

Given the above example description and background vocabulary, PAL constructs the following initial plausible definition[6] (for presentation purposes, we have adopted the following notation: *bl* = *black*, *wh* = *white*, and *square*(X,Y) = (X,Y)):

[5]Background knowledge definitions can include predefined predicates, such as *piece_move* in the definition of *in_check*. The user informs PAL which predicates are considered to be relevant with a list of predicate names, i.e., from which background knowledge definitions to derive a set of atoms.

[6]The complete clause consists of 22 literals in total, with 2 more *contents*/4 literals and 11 more *legal_move*/5 literals between the white Knight, white King and black King.

```
concept(bl,king,(3,4),bl,queen,(3,6),wh,king,(6,7),wh,knight,(1,5),pos1) ←
        contents(bl,king,(3,4),pos1),
        contents(wh,king,(6,7),pos1),
        ...
        other_side(bl,wh),
        other_side(wh,bl),
        sliding_piece(queen,(3,6),pos1),
        in_check(bl,(3,4),knight,(1,5),pos1),
        legal_move(bl,king,(3,4),(4,5),pos1),
        ...
        legal_move(wh,king,(6,7),(7,8),pos1),
        ...
        legal_move(wh,knight,(1,5),(2,7),pos1),
        ...
```

PAL then follows an experimentation process by automatically generating examples (validated by the user) from which other facts are deduced and similar definitions constructed. Following a generalisation process between definitions, eventually PAL recognises that one King is irrelevant to the concept and arrives at the definition given below after generating 22 positive and 67 negative examples.

```
concept(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(S1,S2),
        in_check(S1,(X1,Y1),P3,(X3,Y3),Pos),
        legal_move(S2,P3,(X3,Y3),(X2,Y2),Pos).
```

The above definition says that there is a *fork* if there is an example description (*Pos*) where a piece *P3*, in *square(X3,Y3)*, checks the opponent's King at *square(X1,Y1)* and threatens at the same time piece *P2* in *square(X2,Y2)*. The next section, describes how PAL constructs pattern definitions from example descriptions and how it generalises between definitions.

5.2 The generalisation method

Chapter 4 describes a model of generalisation, based on subsumption, between Horn clauses, and presents Buntine's method for constructing the relative least general generalisation or *rlgg* of two clauses. A plausible learning algorithm is first described following Buntine's method. Heuristics are then introduced to limit the size of the hypotheses and increase the generalisation steps followed by the algorithm. Finally the generalisation algorithm used by PAL is summarised with these heuristics.

5.2.1 Rlgg of clauses

Buntine suggested a method for constructing the relative least general generalisation (*rlgg*) of two clauses [Bun88] (see also section 4.3). His method can be extended to a set of clauses. Let C_1, C_2, $\ldots$, C_n be a set of clauses with disjoint variables and $\mathcal{K}$ a logic program. For $1 < i < n$, let $\theta_{i,1}$ be a substitution grounding the variables of the head of clause C_i to new constants and $\theta_{i,2}$ grounding the remaining variables in C_i to new constants. If $lgg_{\mathcal{K}}(C_1, C_2, \ldots, C_n)$ exists, it is equivalent w.r.t. $\mathcal{K}$ to the $lgg(C'_1, C'_2, \ldots, C'_n)$, where for $1 < i < n$,

$$C'_i \equiv C_i\theta_{i,1} \cup \{\neg A_{i,1}, \neg A_{i,2}, \ldots\} \tag{5.2}$$

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 5.2 can be rewritten as:

$$C'_i \equiv C_{ihead}\theta_{i,1} \leftarrow C_{ibody}\theta_{i,1}, A_{i,1}, A_{i,2}, \ldots \tag{5.3}$$

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

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

Since $lgg(C'_1, C'_2, \ldots, C'_n) \equiv lgg(C'_1, lgg(C'_2, \ldots, lgg(C'_{n-1}, C'_n)\ldots))$ [Plo71b], 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, this can form the basis of a learning algorithm whose target definition is the *rlgg* of these examples (see

Table 5.2). In particular, a learning algorithm can accept a new example, construct a clause with it and atoms derived from the background knowledge (logic program), and gradually generalise 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[7]. 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. However, the *rlgg* exists if a finite number of ground atoms are a logical consequence of the theory.

5.2.2 Knowledge representation: Producing only relevant facts

This section introduces constraints which produce a limited set of relevant facts from the background knowledge and the example description. An example in PAL, is specified by a set of ground unit clauses, representing the position of each piece in the board. PAL uses an example description to derive (recognise) a set of facts (instances of patterns) from its background knowledge (pattern definitions). In PAL, the background knowledge is encoded as pattern definitions, within a restricted form of non-recursive Horn clauses. Each time an example description is given to the system the following ground clause is constructed:

$$Head \leftarrow D_1, D_2, \ldots, D_n, F_1, F_2, \ldots, F_m{}^8. \tag{5.4}$$

where,

- *Head* is a ground atom constructed with the arguments used to describe the example (see below).
- The D_is are the "input" predicates used to describe the position.
- The F_js are ground instances of pattern definitions which are either provided as background knowledge or learned by PAL (which are derived from the background knowledge and the example description).

[7]The reader may wish to compare at this point Table 5.1 and Table 5.2.

[8]Roughly, the D_i's and F_j's of Equation 5.4 correspond to the C_{body}, and the A_k's of Equation 5.3. Note also that Equation 5.3 is simplified by considering the examples as ground clauses.

- **given:**
  - a logic program ($\mathcal{K}$)
  - a set of example clauses with disjoint variables (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
    $\mathcal{K} \wedge C_{kbody}\theta_{j,1}\theta_{j,2} \models A_{j,k}$ and $A_{j,k}$ is a ground atom
  - Set $NC = lgg(C'_j, NC)$
  - Set $SC = SC - \{C_j\}$
- **output** NC

Table 5.2: A plausible *rlgg* algorithm for a set of example clauses

Clause definitions assume that there is an implicit example description. This is more clearly explained below.

5.2.2.1 Determining the clause head

Unlike other systems, descriptions of board positions provided as ground unit clauses constitute valid example presentations of the target concept. Since they do not specify exactly which are the relevant arguments of the target definition, a "tentative" concept head is initially constructed with a new predicate name with all the arguments used in the example description. In the example of section 5.1 this corresponds to "*concept*" with all the arguments used in the *contents*/4 atoms (where, *arguments*([foo(X,b),foo(a,Y)]) = [X,b,a,Y]). The initial head, in conjunction with the facts derived from the background knowledge with the example description, constitutes an initial concept clause. This clause is gradually generalised by taking the *lgg* of it and clauses constructed from other example descriptions. Once a generalisation is produced, heuristics are used to reduce the size of the clause and the number of arguments used in the head. Descriptions of pieces which are removed from the concept definition (and therefore, from the concept's head) are considered as irrelevant to the definition. The heuristics used to determine which pieces are relevant and which irrelevant will be described in section 5.2.3.

New heads, compatible with the current concept head, are constructed from subsequent examples by considering only the arguments of the descriptions of the relevant pieces (see Table 5.3). Once a concept clause has been constructed, the concept head is augmented with new arguments which appear in the body of the clause. This will be explained in Table 5.4 where the learning algorithm is fully described[9].

The arguments which appear in the concept definition, other than the position (*Pos*), can be used to show exactly which pieces are involved in future instantiations of the pattern. For example, after learning the concept of *threat*, knowing that a white Knight at file 1 and rank 5 threatens a black Queen at file 3 and rank 6 in a particular position, can be used by a planning system to suggest a move. This is more informative to a planning system than knowing only that there is a threat in a position. By not using a predefined concept head, a more natural way in which to represent examples

[9]Appendix A shows a trace of the generalisation process involved when learning a Chess concept.

given:
an example description (D_1).
set *Current-Head* = $concept(A_1, A_2, \ldots, A_n)$, where
concept is a new predicate name and
$A_1, \ldots, A_n = arguments(D_1)$

while *termination criterion* is not met
accept a new example description (D_i)
set $D'_i = relevant\text{-}pieces(D_i)$
set $Head_i = concept(A_1, A_2, \ldots, A_m)$, where
$A_1, \ldots, A_m = arguments(D'_i)$
set *NHead* = *generalisation*($Head_i$,*Current-Head*)
set *Current-Head* = *reduce*(*NHead*)

Table 5.3: Determining the head of the clause

can be provided which does not depend on the user's knowledge of the final form of the target definition.

5.2.2.2 Determining the body of the clause

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 the pieces in Chess). We want to consider only those facts which are relevant to a particular position (e.g., only the legal moves of a particular Chess position). 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. Background definitions given as ground unit clauses are also considered as relevant. PAL only considers facts which are relevant to the example description.

The background knowledge definitions used by PAL assume that there is an implicit example description. The representation language used for the background definitions facilitates the derivation of relevant facts. For example, the pattern definition of *in_check*, given in section 5.1, gets instantiated only with descriptions of Chess positions where there is a King at *KPlace* and an opponent's piece *OPiece* at *OPlace* with a piece move to *KPlace*.

PAL uses the current example description (the D_is) in conjunction with

the current background knowledge to derive a set of relevant facts (the F_js). Both, the D_is and the F_js constitute the body of the clause for the current example description. The pattern definitions used in the example of section 5.1 are, *sliding_piece*/3, *other_side*/2, *legal_move*/5, and *in_check*/5 (some of their instances are shown in the example). Roughly, the D_is represent which pieces are involved in a particular pattern, while the F_js are the conditions that those pieces must satisfy for the pattern to be recognised. As non-ground background definitions follow the format of Equation 5.4, only a finite number of relevant facts can be deduced from the background definitions and a particular position.

All the pattern definitions which are available to the system are used to derive a set of relevant facts. Each new example description replaces any previous example descriptions. Depending on the pattern definitions and on the particular board description, PAL is able to derive more or fewer ground atoms. Once a pattern definition is learned, it is included into the background knowledge and can be used to learn subsequent concepts.

5.2.2.3 Dynamic patterns

Chase and Simon suggest that the correlation between perceptual skill and Chess skill can be accounted for by assuming that many Chess patterns are directly associated with appropriate plausible moves [CS88]. Similarly, PAL can learn pattern definitions 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. Instantiations of this predicate represent the different possible 1-ply movements. The *make_move* predicate has a similar format as the *legal_move*/5 predicate, except that it makes sure that the opponent's King is not in check before the actual movement is performed, it has an extra argument and it is used to change descriptions of positions (the background knowledge definitions of *legal_move* and *make_move* are given in appendix G). For instance,

$$make_move(white,king,square(1,2),square(2,3),pos1,pos2)$$

represents a movement of the white King from one place to another changing the current description of the board from position 1 (*pos1*) to position 2 (*pos2*) (i.e., creating a new state description with *contents*(*white,king,square*(2,3),*pos2*)

in it). To constrain the production of moves, only moves which involve at least one of the pieces in the example description and only 1-ply ahead are considered.

A pattern is *static* if it does not depend on *make_move*. PAL produces clauses as described in Equation 5.4 above for static patterns. A pattern is *dynamic* if it has at least one instance of the *make_move* predicate. To learn dynamic 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. The user can specify if the pattern to be learned is static or dynamic. The following clause is produced for dynamic patterns:

$$\begin{aligned} Head \leftarrow & \\ & D_1, D_2, \ldots, D_k, \\ & F_1, F_2, \ldots, F_m, \\ & MV_1, F_{1,1}, F_{1,2}, \ldots, F_{1,n}, \\ & MV_2, F_{2,1}, F_{2,2}, \ldots, F_{2,p}, \\ & \vdots \\ & MV_r, F_{r,1}, F_{r,2}, \ldots, F_{r,s}. \end{aligned} \tag{5.5}$$

where,

- MV_i is an instance of the *make_move* predicate representing a legal move 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.

If a new predicate name appears/disappears after a move, its complement is added before/after the move[10]. For example, the following definition tests for a possible capture of a piece in Chess. The definition of *threat*/7 given below, was learned by PAL after generating 23 positive and 8 negative examples.

```
threat(S1,P1,(X1,Y1),S2,P2,(X2,Y2),Pos1) ←
    contents(S1,P1,(X1,Y1),Pos1),
    contents(S2,P2,(X2,Y2),Pos1),
    other_side(S2,S1),
    make_move(S1,P1,(X1,Y1),(X2,Y2),Pos1,Pos2),
    ¬ contents(S2,P2,(X2,Y2),Pos2).
```

[10]The complement of P is $\neg P$ and vice versa.

Its interpretation is that a piece P2 is being threatened if it can be captured by another piece (P1). Similarly, the following definition tests for possible checks involving one move. The 1-ply check definition, given below, was learned by PAL using the definition of *in_check* given in section 5.1 after generating 20 positive and 18 negative examples. Its interpretation is that a piece *P1* (not checking a King) can check the opponent's King after moving to a new place (X3,Y3) (more examples are given in chapter 6).

```
can_check(S1,P1,(X1,Y1),S2,king,(X2,Y2),(X3,Y3),Pos1) ←
        contents(S1,P1,(X1,Y1),Pos1),
        contents(S2,king,(X2,Y2),Pos1),
        other_side(S1,S2),
        ¬ in_check(S2,(X2,Y2),P1,(X1,Y1),Pos1),
        make_move(S1,P1,(X1,Y1),(X3,Y3),Pos1,Pos2),
        in_check(S2,(X2,Y2),P1,(X3,Y3),Pos2).
```

5.2.3 Other constraints and heuristics

As seen in chapter 4, the *lgg* algorithm produces very small generalisation steps and can generate a large number of literals. PAL uses the following constraints and heuristics to limit the length of the clauses and remove redundant literals.

5.2.3.1 Variable connection

A variable is *connected* to a literal, if it is equal to a variable of that literal. A variable connection constraint is used to reduce the number of arguments in the head of the clause and the number of literals involved.

- Variable head arguments which are not *connected* to literals in the body of the clause (other than the input predicates or D_is) are removed from the definition.
- Body literals with variable arguments that are not *connected* to other variables in the body of the clause are eliminated from the definition[11].

[11] There is only one exception to this, when a move predicate is made to a place which is 'unconnected', but which is followed by new connected predicates.

Disallowing unconnected variables is a natural restriction which has been used by other ILP systems (e.g., [dRB88, MF90, Rou91]). Pattern definitions express different relations between pieces and places in the board. The variable connection constraint removes those literals which are not connected (related) to any other literal in the clause. Input predicates (D_is) with variable arguments in their position that are not connected to the body of the clause are considered as irrelevant and removed from the concept definition.

For instance, the following is an example of an *unconnected* clause:

married(X) $\leftarrow$ husband(X,Y).

Allowing unconnected variables in the body of the clause can produce a very large number of literals many of which may be redundant. Controlling such explosive generation of literals by allowing some unconnected variables is not easy. Without this constraint, several authors have reported clauses of thousands of literals long for very simple concepts (e.g., [Bun88, Fen90, MF90]). In particular, Rouveirol [Rou91] reports how a clause of 1,100 literals is reduced to 70 literals using this constraint for a simple concept.

5.2.3.2 A novel and powerful constraint

The variable connection constraint can make big reductions in the length of the clauses after the *lgg* has been computed, however, there can still remain many redundant literals. In particular, the *lgg* of two clauses can introduce a large number of redundant literals by taking the *lgg* of compatible literals that refer to different objects.

A Chess pattern consists of a set of relations between pieces. Given two positions which are instances of the same pattern we can identify a correspondence between pieces in the two positions. For example, when the pattern is *threat*, there is a piece A threatening a piece B in each position. The fact that we can form this correspondence provides a constraint for the generalisation mechanism.

Our solution for implementing this constraint is to label every constant occurring in the atoms used to describe positions and to keep those labels during the derivation process. By keeping track of which pieces are responsible for which literals, we can eliminate all the generalisations between literals involving different pieces. This is a novel and powerful constraint which can produce significant cuts in the production of irrelevant literals. A simple example is used to illustrate this.

The *lgg* algorithm methodically produces the *lgg* of compatible literals, starting with the heads of the clauses and progressively moving towards the end of the clauses. The constants occurring in the example description, from which all the literals are derived, are labeled with unique constant symbols. For instance, if we have an example position with two pieces:

contents(wh,bishop,(2,3),pos1) $\longrightarrow$ *contents*(wh$_\alpha$,bishop$_\beta$, (2$_\gamma$,3$_\delta$),pos1)
contents(bl,knight,(1,5),pos1) $\longrightarrow$ *contents*(bl$_\eta$,knight$_\lambda$, (1$_\mu$,5$_\sigma$),pos1)

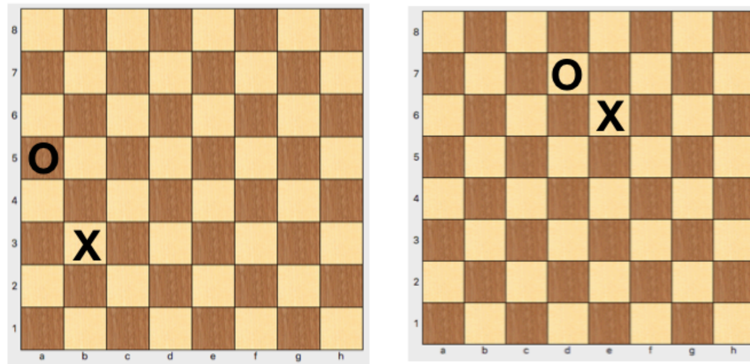
unique labels, i.e., α, β, γ, ... are associated with each constant of the example description. These labels are kept during the derivation of each fact from the background knowledge. For example, if we have the pattern definition of *legal_move*, we can have:

contents(wh$_\alpha$, bishop$_\beta$, $(2_\gamma, 3_\delta)$, pos1)
contents(bl$_\eta$, knight$_\lambda$, $(1_\mu, 5_\sigma)$, pos1)
legal_move(wh$_\alpha$, bishop$_\beta$, $(2_\gamma, 3_\delta)$, (6,7), pos1)
...
legal_move(bl$_\eta$, knight$_\lambda$, $(1_\mu, 5_\sigma)$, $(2_\gamma, 3_\delta)$, pos1)
legal_move(bl$_\eta$, knight$_\lambda$, $(1_\mu, 5_\sigma)$, (3,6), pos1)
...

where the labels associated with the arguments of the atom describing a particular piece are kept during the derivation proofs of relevant facts which involve those labelled constants. The derivation process might involve several atoms (descriptions of several pieces). For instance, the legal move of the black Knight into the place of the white Bishop in the example above. In this way the system can distinguish which piece(s) is(are) "responsible" for which facts by following the labels in the clause.

Significant cuts in the production of irrelevant literals can be made by using the labels to guide and constrain the *lgg* of compatible literals. If the example generator (described in section 5.3) changes the white Bishop at *square*(2,3) to a black Pawn at *square*(5,6), and the black Knight in *square*(1,5) to a white Bishop at *square*(4,7), then PAL produces a new clause but maintains the corresponding labels of the first position (see Figure 5.3).

contents(bl$_\alpha$, pawn$_\beta$, $(5_\gamma, 6_\delta)$, pos2)
contents(wh$_\eta$, bishop$_\lambda$, $(4_\mu, 7_\sigma)$, pos2)



Piece1 = white bishop at(2,3)/black pawn at(5,6)
Piece2 = black knight at(1,5)/white bishop at(4,7)

Figure 5.3: The 'O' and 'X' represent the associated labels that are used to distinguish each piece description (i.e., α, β, ...).

> *legal_move*(bl_α, pawn_β, $(5_\gamma, 6_\delta)$, (5,5), pos2)
> ...
> *legal_move*(wh_η, bishop_λ, $(4_\mu, 7_\sigma)$, $(5_\gamma, 6_\delta)$, pos2)
> *legal_move*(wh_η, bishop_λ, $(4_\mu, 7_\sigma)$, (3,6), pos2)
> ...

The labels used in the first position for the first piece are associated with the first piece of the second example. The example generator knows which pieces are changed and associates their corresponding labels[12]. 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. In the example above, the *lgg*s between legal moves of the white Bishop in the first position (9 literals) will be taken only with the legal moves of the black Pawn in the second position (1 literal). Similarly for the rest of the compatible literals of the clauses. A simple matching procedure is employed to do this; *lgg*s of compatible predicates with different labels are not considered. The resulting clause is:

> *contents*(S1_α, P1_β, (X1_γ,Y1_δ), Pos)
> *contents*(S2_η, P2_λ, (X2_μ,Y2_σ), Pos)
> *legal_move*(S1_α, P1_β, (X1_γ,Y1_δ), (X3,Y3), Pos)
> ...

[12]Examples which are manually provided or selected at random required that the pieces are presented in the same order. Examples are given in chapter 6.

legal_move($S2_\eta$, $P2_\lambda$, ($X2_\mu$,$Y2_\sigma$), ($X1_\gamma$,$Y1_\delta$), Pos)
legal_move($S2_\eta$, $P2_\lambda$, ($X2_\mu$,$Y2_\sigma$), (3,6), Pos)
...

If variable arguments X3 and Y3 in one of the above literals are not connected to any other variable in the clause, this literal is eliminated by the variable connection constraint.

The labels are also used to distinguish between relevant and irrelevant (i.e., unconnected) atoms to construct appropriate compatible heads (see section 5.2.2).

What are we leaving out?

Without this constraint the *lgg* algorithm considers generalisations between predicates that involve different pieces. This is equivalent to consider generalisations that result from switching the pieces in the board. On the other hand, this constraint can produce a large reduction in the length of the clauses. In the above example, Piece1 has nine legal moves in the first position and one in the second, while Piece2 has four legal moves in the first position and six in the second. The *lgg* algorithm without labels produces 91 literals for the legal move predicate (i.e., $(9 + 4) \times (6 + 1)$). With labels it produces 33 ($9 \times 1 + 6 \times 4$). In general, the *lgg* algorithm can produce clauses of length N^2 for two clauses with N compatible literals. Lets say we have N compatible literals (such as legal moves) from three different pieces $P1$, $P2$, and $P3$, that can be split as follows $N = N_1 + N_2 + N_3$ (where N_i is the number of literals of piece Pi for $i = 1, 2, 3$). If we have another example with the same number of compatible literals for the same pieces, the *lgg* algorithm produces a clause of length $(N_1 + N_2 + N_3)^2$, while *lgg* with labels produce a clause of length $N_1^2 + N_2^2 + N_3^2$.

5.2.3.3 Restricted moves

Considering all the possible facts that change after each possible move can produce very long clauses (e.g., consider all the possible new legal moves that can be generated after all the 1-ply moves of all the pieces). To constrain this, only those moves which introduce a new predicate name or remove an existing predicate name are added to the clause (see for example the definitions of *threat* and *can_check* given above). Concepts where this constraint is

partially relaxed and its consequences to the efficiency of PAL, are discussed in chapter 7.

5.2.4 The generalisation algorithm

Table 5.4 summarises the generalisation algorithm[13]. It has been able to learn concepts like *forks*, *threats*, *pins*, *skewers*, *discovered checks/attacks*, etc. (results are given in chapter 6). Examples are given as descriptions of Chess positions. The background knowledge is described as pattern-based definitions from which only a finite number of relevant facts can be derived given an example description. Unique labels associated with the description of each piece are used to guide the *lgg* between clauses, reduce the length of the clauses, and avoid the production of irrelevant literals which normally occur with an *lgg* algorithm. The user can inform the system whether the concept to learn is dynamic or static, although static patterns can be learned starting from dynamic definitions. In the next section an automatic example generator, which is used to guide the learning process, is described.

5.3 Automatic example generator: A learning mechanism

An important component in the characterisation of an inference method is its example presentation. It is generally believed that a "careful" experiment selection is more effective for concept formation than a random experiment selection. In many inductive systems, the examples are provided by the user. The examples can be given interactively or in a batch mode. In an interactive presentation, examples are often carefully selected by the user to achieve the desired results (e.g., [MB88, Rou91, SB86, Win85]). Unfortunately in some such cases, the system will fail to produce the required concept if it is provided with another selection of examples or even with the same examples but in a different sequence. For instance, a system like Cigol [MB88] will produce the following generalisations if presented with the following sequence of examples (where => means the generalisations produced by Cigol):

(1) member(a,[a]).

[13]A simplified Prolog implementation of the algorithm is given in appendix H.

given:

- a description of a Chess board position D_1
- a set of pattern definitions T (domain theory)

construct an initial clause $\mathit{Current\text{-}Clause} = \mathit{Head}_1 \leftarrow \mathit{Body}_1$, where
$\mathit{Head}_1 = \mathit{concept}(A_1, A_2, \ldots, A_n)$,
concept is a new predicate name, and
$A_1, \ldots, A_n = \mathit{arguments}(D_1)$
$\mathit{Body}_1 =$ all possible *relevant* facts derivable from $T \cup D_1$

while new example descriptions are provided
accept a new Chess board description D_i
set $D'_i = \mathit{relevant\text{-}pieces}(D_i)$
construct a new clause $\mathit{Clause}_i = \mathit{Head}_i \leftarrow \mathit{Body}_i$, where
$\mathit{Head}_i = \mathit{concept}(A_1, A_2, \ldots, A_m)$ and
$A_1, \ldots, A_m = \mathit{arguments}(D'_i)$
$\mathit{Body}_i =$ all possible *relevant* facts derivable
from $T \cup D_i$

set $\mathit{NClause} = \mathit{lgg}(\mathit{Current\text{-}Clause}, \mathit{Clause}_i)$ [considering *labels*]
set $\mathit{Current\text{-}Clause} = \mathit{Current\text{-}Head} \leftarrow \mathit{Current\text{-}Body} =$
resulting clause after applying *variable connection*
constraints to *NClause*
output $\mathit{Output\text{-}Head} \leftarrow \mathit{Current\text{-}Body}$, where
$\mathit{Output\text{-}Head} = \mathit{concept}(A_1, \ldots, A_j, A_{j+1}, \ldots, A_p)$,
$A_1, \ldots, A_j = \mathit{arguments}(\mathit{Current\text{-}Head})$, and
$A_{j+1}, \ldots, A_p = \mathit{arguments}(\mathit{Current\text{-}Body})$ not present in
Current-Head.

Table 5.4: Chess generalisation algorithm

(2) member(1,[1,2]).
(3) member(f,[f,g,h]).
(3') => member(A,[A|B]). (removes (1), (2), (3))
(4) member(1,[1]). (removed by (3')
(5) member(1,[2,1,3]).
(6) member(a,[b,a,c,d]).
(6') => member(A,[B,A,C|D]). (removes (5), (6))
(7) member(1,[3,2,1,4]).
(8) member(b,[c,d,b,e,f]).
(8') => member(A,[B,C,A,D|E]). (removes (7) (8))
(8") => member(A,[B|C]) :– member(A,C). (removes (6') (8'))

Which ends with the final definition of member:

member(A,[A|B]).
member(A,[B|C]) :– member(A,C).

However, if the fourth example is given at the beginning,

(1) member(a,[a]).
(4) member(1,[1]).

Cigol produces,

(1') => member(A,[A]).

which, unless it is rejected by the user, will produce with the rest of the examples:

member(1,[1,2]).
member(f,[f,g,h]).
member(A,[A]).
member(A,[B|C]) :– member(A,C).
member(A,[B,A,C|D]).

The "closest" we can get to the intended definition, once the first clause is accepted, is the following program (again by choosing an adequate sequence of examples):

```
member(A,[A]).
member(A,[A,B]).
member(A,[A,B,C|D]).
member(A,[B|C]) :– member(A,C).
member(A,[B,A,C|D]).
```

In general, it may be difficult to choose the right examples in the right order and it is difficult to determine the robustness of the system with user-selected examples.

Providing examples in batch mode usually requires the user to collect a fixed (preferably large) set of examples. Providing such a set can be a difficult task and it is not easy to know in advance if the system will learn the concept with such a set. Failing to provide an adequate example set results in incomplete and/or incorrect definitions (for which the user has to provide additional examples). Usually, a long process of analaying why the system failed with the provided examples and the provision of new examples needs to be followed in order to correct the system's behaviour.

An automatic example generator can reduce the requirement on the user to collect examples and at the same time guide the learning process. Section 4.4.2 reviewed some automatic example generators. In this section, a general framework for describing the example space is first discussed. Then, the strategy to generate examples based on the concept definition is described. Finally, the particular strategy followed by PAL to traverse the example space based on "perturbations" is given.

5.3.1 A framework for describing the example space

Just as structuring the hypothesis space can be crucial for concept learning to take place efficiently, so does the example space need to be structured for designing an efficient example generator. In a concept learning algorithm, the example space 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 a particular number of arguments, the example space is defined by all the examples that can be generated by changing or 'perturbing' the values of such arguments.

A *perturbation class* is defined as the set of instances that can be generated by changing only the values of some specified arguments. Similarly, a *perturbation level* represents all the instances that can be generated by

```
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 5.4: Perturbation space

changing the values of a certain number of arguments. A *generator* is defined as a unique constant symbol used to distinguish each argument of the definition and which provides a link between an argument and its domain.

New examples are generated by "perturbing" (changing) the values of the arguments that the generators refer to. If an example can be described by instantiating N arguments, the example space can be distributed among N perturbation levels with $2^N - 1$ different perturbation classes. For example, if an instance of the concept of *threat* between two pieces is described with four arguments, with `P1`, `L1`, `P2`, and `L2` as generators (meaning that piece `P1` in place `L1` threatens piece `P2` in position `L2`), the perturbation space can be structured in four levels (see Figure 5.4).

For example, the second perturbation level of Figure 5.4 represents all the classes of examples that can be generated by changing any two arguments at the same time (i.e., `{P1,L1}`, `{P1,P2}`, ...). At each perturbation class, $D_i \times D_j \times \ldots \times D_n$ examples can be generated, where each D_k is a particular argument domain at that class. For instance, the perturbation class `{L1,P2}` represents the class of examples that can be generated by changing the position of the attacking piece `L1` and the piece which is being threatened `P2`. If the following domain information is given to the system:

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

then the place of the attacking piece can be changed to 63 different positions and the piece being attacked can be changed for 5 new different pieces. Clearly the example space grows exponentially with the number of arguments involved.

Several domain constraints can be applied to reduce this space. In particular, not all the perturbations generate legal examples. For instance, the first and last ranks can be eliminated from the domain of the positions of the Pawns. Knowing that any legal position must contain two and only two Kings (one on each side) can be used to constrain the domains of the possible values for the pieces and avoid changing sides of one King without changing the other, etc. Despite these constraints, the example space can still be huge (e.g., two Kings alone can be in 3612 different legal positions, which corresponds to a perturbation class at level 2).

Given the above framework for organising the example space, the generation of examples is affected by the definition of two main strategies. One is concerned with how to select examples from a particular perturbation class. The adopted strategy is based on the current concept definition and is described in the next section. The other strategy defines the way in which to traverse the example space (i.e., which perturbation levels and classes to consider first). This is discussed in section 5.3.3.

5.3.2 An example selection strategy

Structuring the example space can help us to design a particular strategy in which to traverse the space, however, we would like to "intelligently" select values from each perturbation class. PAL's example selection strategy is guided by the current concept definition. Given a concept definition and a perturbation class, new examples are generated by selecting values from the domains of the arguments involved in the class.

A *failure set* for an example description is defined as the literals in the current clause that are not provable given the background knowledge and the current example description. Similarly, the *success set* for an example description is defined as the literals in the current clause that are provable from the background knowledge and the current example description. The goal is to generate an example that will fail on at least one of the literals of the current concept clause of the definition (i.e., produce a non-empty failure set). The reason being that it may be a positive example (all the examples are validated by the user) that should be covered by the target clause (i.e., the strategy is looking for plausible generalisations of the current clause). If the example is positive, it is given to the generalisation algorithm described in section 5.2 and a new generalised definition is produced.

If the example is classified (by the user) as negative, the example selection

strategy takes the failure set and tries to construct a new example that will succeed on at least one of the literals of the failure set (i.e., tries to generate a non-empty success set for the failure set). This is because the failure set must have at least one literal relevant to the target clause. Removing the failure set from the definition would produce an overgeneralisation, so the strategy is looking for a plausible specialisation of that over-general clause.

Domain knowledge is used to generate only legal positions. The strategy maintains a list of all the generated examples to avoid generating duplicates. It also keeps a list of the failure/success sets for each definition, to avoid generating different examples that will fail/succeed on the same sets. If no new examples can be generated to fail/succeed on different sets of the same concept definition (i.e., it has finished exploring a perturbation class), the strategy waits for the next perturbation class (see Table 5.5).

For instance, following the example of *fork* described in section 5.1 (black King at *square*(3,4), black Queen at *square*(3,6), white King at *square*(6,7), and white Knight at *square*(1,5)), the position of the white King can be changed from *square*(6,7) to *square*(1,7), causing the following literal to fail (among others);

legal_move(white,king,square(1,7),square(7,8),pos1).

This produces a new positive example and eventually a new generalisation. However, if the position of the Knight is changed from *square*(1,5) to *square*(2,5), although it fails on several literals (i.e., has a non-empty failure set), it produces a negative example. The strategy checks which literals failed on that example and tries to find a place that will succeed on at least one of them (e.g., *square*(5,5)). Similarly, replacing the Knight for another piece fails on several literals and produces a negative example. However, changing the Knight to any other piece produces the same failures. The strategy realises that it cannot produce a positive example by changing exclusively the Knight with another piece (i.e., it cannot produce a non-empty success set) and continues with the next perturbation class.

The strategy in which the perturbation classes are given to the example selection strategy is described in the next section. The complete algorithm involving both strategies is summarised in Table 5.7.

given:

- domain values for the arguments used in the example description
- a perturbation class
- a concept clause (*Defn*)

while making sure that duplicate examples, or examples with the same failure set are not generated,
generate an example such that:
failure-set(*Defn*) $\neq \{\emptyset\}$ and show it to the user
if the example is accepted by the user (positive), exit
else if the example is rejected by the user (negative),
set *NegDefn* = *failure-set*(*Defn*)
while making sure that duplicate examples or examples with the same success set are not generated,
generate an example such that:
success-set(*NegDefn*) $\neq \{\emptyset\}$ and show it to the user
if the example is accepted by the user (positive), exit
else if the example is rejected by the user (negative), continue

Table 5.5: Example selection strategy

5.3.3 An experimentation strategy

So far, we have seen how to structure the example space into perturbation classes and how to generate new examples from each class guided by the concept definition. This section describes the strategy which has been followed to traverse the example space. In PAL, experiments involving the minimum number of "changes" are performed first. That is, perturbation classes with the minimum number of generators are explored first. The perturbation classes are generated dynamically. The experimentation strategy takes the arguments used to describe the initial example and constructs only the lowest perturbation level with them. After exploring a perturbation class (looking at all its possible values), only its immediate perturbation classes above are generated. Arguments with empty domains are not considered in the example space. Similarly, if an argument is eliminated from the head of the concept definition (as described in section 5.2), all the perturbation classes where its generator 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 all the possible places of the attacking piece `L1` have been tried, the new perturbation space becomes:

```
{P1,L1}    {L1,P2}    {L1,L2}
     {P1}   {P2}   {L2}
```

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

```
{P1,L1}  {P1,P2}  {P1,L2}  {L1,P2}  {L1,L2}
                {P2}   {L2}
```

If the attacked piece is removed from the head of the concept (i.e., `P2` is no longer relevant), then the new perturbation space becomes:

```
{P1,L1} {P1,L2} {L1,L2}
        {L2}
```

Recognising irrelevant arguments is important since they represent significant cuts in the search space[14]. All the perturbation classes of a lower level

[14]An argument which is not in the perturbation space is not necessarily removed from the definition (i.e., arguments with empty domains).

are explored before classes of upper levels. The perturbation process ends when there are no more perturbation classes left or stopped by the user (see Table 5.6).

Although the experimentation strategy proceeds by levels, the perturbation classes of one level can be sorted to prefer particular perturbation classes first. Similarly, heuristics can be incorporated to prefer certain perturbations first. Unless indicated, PAL changes first the side of all the pieces (i.e., all the white pieces are changed to black and vice versa) and removes all the generators corresponding to the sides from the perturbation space.

An evaluation function is used to sort classes from the same perturbation level. It gives preference to perturbation classes whose arguments are constants, have domains with smaller number of elements, and appear in a larger number of literals in the current hypothesis. Each argument of a perturbation level is assigned with the following value:

$$\text{value}(\text{Arg}) = \text{varcte}(\text{Arg}) + \text{domsize}(\text{Arg}) - \text{nliterals}(\text{Arg}).$$

where *varcte* value is 20 for variable arguments and 0 for constants[15], *domsize* is the size of the domain of the argument, and *nliterals* is the number of literals where the argument appears in the definition[16]. The value of perturbation classes with several arguments is defined as the sum of the values of their arguments. Experiments with different evaluation functions are given in chapter 6.

In section 5.2.3, the arguments used to describe example positions are labelled with unique constant symbols to guide the generalisation algorithm. The example generator uses the same labels (i.e., the generators) to distinguish the perturbation classes, keep track of the changes, and inform the generalisation method of them.

The example space depends on the descriptions of the pieces involved in the initial example and only examples with at most the same number of pieces can be generated. It is assumed that the example space, determined by the domains of the arguments involved in the initial example, contains at least the examples required by the generalisation algorithm to derive the target definition.

[15]This arbitrary assignment is used to express a preference for ‘perturbing’ constant arguments first. Other values will be considered in chapter 6.

[16]This follows the labels of the definition, as described in section 5.2.3, to distinguish between different arguments with the same values.

To create an initial perturbation space

given:

- an example description (EX)
- domain values for the arguments used in the example description

set $[A_1, \ldots, A_n] = arguments(EX)$

begin
for $1 < i < n$
set gen_i = unique constant symbol for A_i, which links A_i with $domain(A_i)$ (create generators)
set $class_i = [gen_i]$
end

set *current level* = 1
set $pert\text{-}level(1) = [class_1,\ class_2,\ \ldots,\ class_n]$
set $pert\text{-}level(2) = [\,]$

To change levels

given:

- an explored perturbation class ($class_i$)
- the current perturbation level $pert\text{-}level(\mathrm{N}) = [class_i,\ class_j,\ \ldots,\ class_m]$

set $pert\text{-}level(\mathrm{N}) = pert\text{-}level(\mathrm{N}) - \{\ class_i\ \}$

if $pert\text{-}level(\mathrm{N}) = [\,]$
set *current level* = N + 1
set $pert\text{-}level(\mathrm{N} + 2) = [\,]$
else begin
for $j < k < m$ (all the perturbation classes of $pert\text{-}level(\mathrm{N})$)
set *new-sets* = all possible sets of length $N + 1$ form by adding new elements of $class_k$ to $class_i$
set $pert\text{-}level(\mathrm{M} + 1) = new\text{-}sets \cup pert\text{-}level(\mathrm{M} + 1)$
end

Table 5.6: Algorithms to create an initial perturbation space and to change perturbation levels

given:

- an example description
- domain values for the arguments used in the example description
- a concept clause definition

construct the lower level of the perturbation space (see Table 5.6)

while there are perturbation classes left
  (1) sort the perturbation classes
  (2) take the first perturbation class and
    call the *experimentation strategy* (see Table 5.5)
      if a positive example is generated, exit
      else change-levels (see Table 5.6) and goto 2.

Table 5.7: The perturbation algorithm

The perturbation method (summarised in Table 5.7) has been used to guide the learning process of the generalisation algorithm described in Table 5.4.

5.3.4 PAL without the example generator

The assignment of unique labels to the description of the initial example provides a linkage between the example generator and the generalisation method. The perturbation method uses the labels to organise its perturbation strategy, while the generalisation method uses them to constrain the *lgg* algorithm, handle properly coincidences in the examples, and distinguish between different pieces. PAL's generalisation algorithm can be used with examples selected from an external agent (e.g., by the user, or randomly selected from an example set). In general, the generalisation process requires less examples when a 'trained' user carefully selects the examples. Comparisons of PAL's performance with/without the example generator are made in chapters 6 and 7, when PAL is used to learn different Chess concepts. PAL still assigns unique labels when the examples are selected externally, however,

the descriptions of the pieces must be given in the same order to maintain consistency. For instance if the first example is described as follows:

contents(white,pawn,square(2,5),pos1).
contents(black,rook,square(1,8),pos1).
...

PAL will assign unique labels to the arguments used in the example description. If another example has the following description:

contents(black,queen,square(2,5),pos2).
contents(white,bishop,square(1,8),pos2).
...

PAL will assume that the white Pawn has been changed for a black Queen and the corresponding labels of the white Pawn are assigned to the black Queen. Similarly, it will assume that the black Rook has been changed for a white Bishop.

PAL takes the description of the pieces to build a concept head. By keeping the same order in which the pieces are given to the system compatible heads can be constructed. Otherwise, PAL would be considering examples where the pieces have been switched.

5.3.5 Related work

Section 4.4.2.1 provides a general overview of different example generators. In this section we compare the automatic example generator used by PAL with that of other systems.

One approach followed by some ILP systems with an automatic example generator (e.g., Clint [dRB88] or Marvin [SB86]), is to select a subset of literals from the current hypothesis and search for an example that will succeed with that subset but fail with the current hypothesis. This can be computed by taking the current hypothesis (C) a subset of it (C'), standarising them apart, and finding an example ($C'_{head}\theta$) which satisfies the following query: $\leftarrow C' \wedge \neg C$. This however assumes a perfect query answering mechanism. For first-order logic this is only semi-decidable (it may never stop) and systems which followed this approach have restricted their hypothesis language to DATALOG programs (i.e., programs without function symbols), where queries are decidable. Rather than searching for an example, PAL suggests

examples directly trying to fail the current hypothesis. A similar strategy is followed by systems like AM [Len76] and LEX [MUB83] where examples are proposed by the system (e.g., extreme cases in AM or near misses in LEX) and their usefulness are tested later. PAL, however, does not use a hierarchy of concepts (as LEX or AM) from which to suggest examples.

Subramanian and Feigenbaum [SF86] showed that a large reduction of examples can be obtained in factorable concepts by considering each component independently (see also section 4.4.2.1). In general, information about the factorability of the concepts is not available/possible. We can say that PAL explores the different possible independent factors of a concept by considering the different perturbation classes. PAL explores first one plausible independent factor by changing the value of one argument while holding others constant. This process continues with other arguments and with combinations of arguments until no more perturbation classes can be explored. PAL can tell exactly which arguments affect which literals by following the unique label symbols associated with each definition and could tell in principle which are the independent factors.

PAL's example generator provides a way in which to structure the example space, the examples space can be reduced during the learning process, performs small perturbations first, and has a clear termination criterion.

5.4 The learning algorithm

The learning algorithm can be summarised using the descriptions of the previous two sections. Initially, PAL 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. PAL first constructs an initial concept definition and an initial perturbation level. It then calls the example generator to create new examples (see section 5.3). Each time a positive example is created, the system uses the constrained *rlgg* algorithm (see section 5.2) to construct 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 expands the perturbation space and continues with the next perturbation class. The process ends when there are no more perturbation levels left, or when the user decides to terminate the process.

5.4.1 Disjunctive definitions

As mentioned in chapter 4, the *lgg* algorithm constructs a single clause (the least general generalised clause) from a set of clauses. The *lgg* algorithm fails with disjunctive definitions (i.e., definitions which required more than one clause). Each time an *lgg* is computed between two example clauses of different disjuncts, an overgeneralised clause is produced. In order to learn disjunctive definitions, systems which use *lgg* must rely on the negative examples provided to the system and/or 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. Since PAL generates its own examples, there is no guarantee that sufficient negative examples will be produced to reject possible overgeneralisations. PAL asks the user for confirmation before accepting any definition. In practice, very few definitions were rejected by the user due to the example generator strategy followed by PAL. When the perturbation process finishes, 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. In this way, PAL learns disjunctive concepts by learning each clause separately (see Table 5.8). 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 all the disjuncts to be learned.

5.4.2 Distinction between components

PAL is applicable to domains where examples are given as descriptions of states in terms of components. In Chess the components are the pieces in

the board. In a simulation, the components can be the state variables (an example will be given in chapter 8). Associating descriptions of states with unique variables serve two purposes in PAL. On one hand, it guides and constraints the generalisations between clauses. On the other, it is used to construct the example space and guide the automatic example generator.

The background knowledge in PAL, which is used to construct the body of the hypotheses, describes relations between the different components. The labels are used to consider only compatible literals which refer to the same components (i.e., compatible literals which express relations involving different components are not considered). This removes a large number of irrelevant literals that would be otherwise generated.

The arguments of the atoms used to describe an instance of the target concept are associated with unique labels. They are used to construct the example space as new instances are constructed by changing the values of such arguments. The hypotheses are associated with these labels so that the automatic example generator can know exactly which literals are affected by which labels (i.e., which literals will be affected by changing particular arguments).

5.4.3 Additional clause reduction

Redundant literals in the final concept definition are removed by expanding the definitions of the literals involved in the body and matching the expansions against the rest of the literals. Literals which are matched by an expansion are removed from the definition. For example, using the definition of *in_check* and *legal_move*, the definition of *fork* given in section 5.1 is reduced to:

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

given:

- an initial example description
- domain values for the arguments
- background knowledge definitions

construct an initial clause (as described in section 5.2)
construct an initial perturbation level (as described in section 5.3)

do until no more perturbation levels or stopped by the user
  call the *perturbation algorithm* to generate a new positive example
  call the *generalisation algorithm* to generate a new definition
    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 against the stored examples
if some examples are not covered,
then pick the first uncovered example and start again
else reduce the new definition and add it to the background
  knowledge

Table 5.8: Learning algorithm

5.5 Summary

An ILP system capable of learning patterns in Chess expressed in a subset of first order Horn clauses has been described. PAL uses a constrained relative least general generalisation algorithm of clauses (*rlgg*) as generalisation method, and an automatic example generator to reduce the user's intervention and guide the learning process. Examples are given as atoms describing the position of each piece in the board. The goal predicate or relevant arguments for the concept are not specified in advance. PAL uses a non-recursive pattern-based knowledge representation, which considers the current example description and from which only a finite set of relevant facts can be derived. Each description of the board is associated with unique labels to guide the generalisation algorithm and reduce the length of the hypotheses, providing an effective implementation of *rlgg*. The background knowledge definitions are restricted to general predicates stating the rules of the game, which in conjunction with a simple example presentation, provides an adequate framework in which to learn patterns in Chess. Examples of concepts learned by PAL will be given in the next chapter.

The example space, from which the automatic example generator selects its examples, is generated dynamically and can be reduced during the learning process. It has a clear termination criterion and it is guided by the current concept definition. It avoids the production of duplicates or spurious examples that do not create any new failures/successes in the literals of the concept definition. Disjunctive definitions can be learned by PAL, although the user might intervene to avoid overgeneralisations and an instance of each disjunct must be provided in advance.

The user informs the system whether the concept to learn is static or dynamic (i.e., involving a 1-ply move), selects the patterns to be used as background knowledge, provides the particular domains of the arguments used to describe the examples, and chooses an initial example. A simplified Prolog implementation of PAL[17] is given in appendix H with enough comments to allow an interested reader to produce a complete reconstruction of the algorithm. Chapter 6 describes the conditions on which several Chess concepts were learned by PAL, compares its performance with other ILP systems, and discusses PAL's limitations. Chapter 7 shows that the patterns learned by PAL can be used to construct a correct playing strategy for a

[17]PAL is written in Quintus Prolog.

simple endgame. Chapter 8 shows how PAL can be used to other domains by learning a qualitative models of a dynamic system from descriptions of qualitative states.

Chapter 6

Learning patterns in Chess

Although Chess is considered as the game par excellence in AI, very little progress has been made within the machine learning community to learn concepts in Chess. Chapter 5 describes PAL, a first order learning system capable of learning patterns in Chess from simple example descriptions together with the rules of the game. This chapter shows how PAL is used to learn several concepts in Chess, compares its performance with other ILP systems, and discusses its main limitations. In particular, section 6.1 shows how PAL is used to learn the concept of illegal white-to-move positions for the King and Rook against King endgame, which has been recently used to evaluate some ILP systems. Section 6.2 analyses the applicability of Foil [Qui90] and Golem [MF90] in Chess by looking at their performance on a different concept in Chess. Section 6.3 describes how PAL is used to learn other more useful Chess concepts which involves a larger background vocabulary. Finally, section 6.4 analyses PAL's performance and discusses its limitations.

6.1 Illegal WTM KRK positions

A comparative study between Duce [Mug87], Cigol [MB88], Assistant-86 [CKB87], and C4 [Qui87], suggests that the ability to produce high performance in a domain like Chess, relies almost exclusively on the ability to express first order predicate relations [MBHMM89]. The study involved learning the concept of illegal white-to-move (WTM) positions for the King and Rook against King (KRK) endgame. It consisted of 2 main experiments (with 100 and 1000 examples each).

- Experiment 1: 5 random training sets of 100 and 1000 examples. The hypothesis vocabulary is limited to express the file and rank of each piece.
- Experiment 2: 5 random training sets of 100 and 1000 examples with additional hypothesis vocabulary to express equality between files/ranks, less than (one file/rank is less than other file/rank), and adjacent (one file/rank is adjacent to another file/rank).

The resulting outputs of each system were tested against 5 random sets of 1,000 positions. Although efficiency problems constrained Cigol to be used only with training sets of 100 examples, its average performance in the second experiment (77.2 %) surpassed that of the other systems.

Since then, several ILP systems have used the same problem as a test-bench (namely, Foil [Qui90], Golem [MF90], and Linus [LDG91]). All of them have arrived at the following (or equivalent) definition with training sets of 1,000 examples[1]:

illegal(A,B,C,D,C,E).
illegal(A,B,C,D,E,D).
illegal(A,B,A,B,C,D).
illegal(A,B,C,D,A,B).
illegal(A,B,C,D,A,E) ← adjacent(B,E).
illegal(A,B,C,D,E,B) ← adjacent(A,E).
illegal(A,B,C,D,E,F) ← adjacent(A,E), adjacent(B,F).

Where, *illegal(A,B,C,D,E,F)*, means that white King in place $\langle A, B\rangle$, white Rook in $\langle C, D\rangle$, and black King in $\langle E, F\rangle$, is an illegal WTM position. Thus, *illegal* positions are those where the black King is in check, the Kings are adjacent to each other or when two pieces occupy the same place. This definition is not correct, as it ignores examples where the white King is in between the black King and the white Rook (all on the same rank/file), which should be considered as legal. However, it covers more than 98% of the cases.

PAL was tested with both the automatic example generator and with randomly generated training sets on the same concept. The same background

[1] Golem uses *adjacent or equal* as background knowledge (i.e., a file/rank is adjacent or equal to another file/rank), with which the number of clauses is reduced to 4.

knowledge was added to PAL. A simplification in the way to describe examples was used to learn this concept, which is closer to that used by the other systems. Each example was described as a set of *piece*(*SP,F,R,Pos*) atoms, where *SP* represents the name and side of the piece, in this case [*wk,wr,bk*] for white King, white Rook, and black King respectively. *F* represents the files [*a,b,. . .,h*] and *R* the ranks [1,2,. . .,8], and *Pos* is the example description. For instance,

piece(wk,e,4,pos4).
piece(wr,c,2,pos4).
piece(bk,e,2,pos4).

represents a position (*pos4*) where the black King is in check by the Rook and in opposition with the white King. The example space was constrained by declaring the domain of side-piece as empty (i.e., only examples where the files and/or ranks are changed can be generated). The definitions were not reduced to leave them with the same number of arguments, although all the clauses could be expressed with only 2 pieces involved. The following definition (equivalent to the definition given above), was obtained after generating 58 + and 21 − examples in total[2]:

(1) illegal(wk,A,B,wr,C,D,bk,C,E,Pos) ←
    piece(wk,A,B,Pos), piece(wr,C,D,Pos), piece(bk,C,E,Pos).
(2) illegal(wk,A,B,wr,C,D,bk,E,D,Pos) ←
    piece(wk,A,B,Pos), piece(wr,C,D,Pos), piece(bk,E,D,Pos).
(3) illegal(wk,A,B,wr,A,B,bk,C,D,Pos) ←
    piece(wk,A,B,Pos), piece(wr,A,B,Pos), piece(bk,C,D,Pos).
(4) illegal(wk,A,B,wr,C,D,bk,A,B,Pos) ←
    piece(wk,A,B,Pos), piece(wr,C,D,Pos), piece(bk,A,B,Pos).
(5) illegal(wk,A,B,wr,C,D,bk,A,E,Pos) ←
    piece(wk,A,B,Pos), piece(wr,C,D,Pos), piece(bk,A,E,Pos),
    adjacent(B,E), adjacent(E,B).
(6) illegal(wk,A,B,wr,C,D,bk,E,B,Pos) ←
    piece(wk,A,B,Pos), piece(wr,C,D,Pos), piece(bk,E,B,Pos),
    adjacent(A,E), adjacent(E,A).
(7) illegal(wk,A,B,wr,C,D,bk,E,F,Pos) ←
    piece(wk,A,B,Pos), piece(wr,C,D,Pos), piece(bk,E,F,Pos),

[2]The same definition can be obtained with 14 carefully selected examples.

adjacent(A,E), adjacent(E,A),
adjacent(B,F), adjacent(F,B).

PAL was modified to accept examples that were randomly generated. Each file/rank of each piece was generated by a random integer generator between 1 and 8, and all the pieces in the example description were given in the same order to maintain consistency in the assignments of labels (see chapter 5). After N randomly generated examples, PAL considers all the positive examples to generate a definition. It takes the first positive example, creates an initial clause and considers the remaining positive examples, one by one, from which similar clauses are constructed and new generalisations constructed. If with a new positive example, a generalisation is created which covers a negative example, then that generalisation is ignored and the example is stored. If a positive example creates a generalisation which does not cover any negative example, then this new definition is kept and used with the rest of the positive examples. After all the positive examples have been tried, the final definition is saved and checked against the list of stored examples. Those examples which are covered by the definition are removed and the process continues with the rest of the (uncovered) examples until all the positive, without any negative examples have been covered (see Table 6.1). The general principle of generating definitions to cover positive examples without covering any negatives until all the positives are covered, is similar to the strategy used by other ILP systems like Golem or Foil. PAL, however, does not allow definitions to cover a small percentage of negative examples, as the other systems do. A more detailed comparison between Golem and PAL is given in section 6.2.

As with other ILP systems, PAL was tested over 5 training sets of 100 randomly generated examples. The following clauses were obtained for each training set:

- Test1: clauses (1), (2), (4), (5), (6), and a special case for (7).
- Test2: clauses (1), (2), (5), and (6).
- Test3: clauses (1), (2), (3), a special case for (5), a special case for (6), and (7).
- Test4: clauses (1), (2), a special case for (4), a special case for (5), a special case for (6), and (7).

- Generate N random examples.
  Set P = list of positive examples.
  Set $S = [\,]$ (list of uncovered examples).
- **while** $P \neq [\,]$
  1. take the first example from P (e_1) and create an initial clause definition (C_1-*Defn*). Set $P = rest(P)$.
  2. **if** $P = [\,]$, output C_1-*Defn*. Set $P = S$, set $S = [\,]$ and continue.
  3. **else**, take the first positive example from P (e_i), create a clause with it (C_i-*Defn*), and
     set *NewC-Defn* = *generalisation*(C_1-*Defn*, C_2-*Defn*)
     - **if** *NewC-Defn* covers a negative example,
       then ignore it, store e_2 in S, set $P = rest(P)$, and goto 2.
     - **else** make C_1-*Defn* = *NewC-Defn*, set $P = rest(P)$, and goto 2.

Table 6.1: PAL's algorithm for randomly generated examples

- Test5: clauses (1), (2), (3), a special case of (4), (5), and (7).

Similar 'incomplete' clauses are obtained when any of the other ILP systems is used with training sets of 100 examples. The problem is that in most cases, while several examples for clauses (1) and (2) can be randomly generated, almost none are generated for the rest of the clauses. PAL needs at least two examples of each disjunct to produce the appropriate generalisation (this is also true for a system like Golem). There is about 1/8 chance of generating one example for clauses (1) or (2), 1/16 for clause (7), 1/32 for clauses (5) or (6), and 1/64 chances of generating one for clauses (3) or (4). Thus an average of 16 examples are required to generate one of the clauses (1) or (2), 32 examples to generate clause (7), 64 for clauses (5) or (6), and 128 examples are required for clauses (3) or (4). Sometimes both examples might lie on the same rank or file, producing an over-specialised clause. An estimate of between 448 examples and 544 (considering an extra example for clauses (1), (2), (5), (6), and (7) in case an example lies on the same rank or file), is in theory required to generated all the clauses. With this in mind, an additional test was made with a training set of 500 examples. This was enough to produce all the clauses of the above definition. PAL does not allow hypotheses to cover any negative example. Clauses (1) and (2):

illegal(wk,A,B,wr,C,D,bk,C,E,Pos) ←
  piece(wk,A,B,Pos), piece(wr,C,D,Pos), piece(bk,C,E,Pos).
illegal(wk,A,B,wr,C,D,bk,E,D,Pos) ←
  piece(wk,A,B,Pos), piece(wr,C,D,Pos), piece(bk,E,D,Pos).

are overgeneral as they cover examples where the white King is in between the white Rook and the black King, and the three pieces are on the same file or rank. During the above trials, those special examples were considered as positive (thus, as illegal positions). In general, it is desirable to allow some inconsistency of the hypotheses with the data. An inconsistency can be produced in the presence of noise and/or when the available background knowledge is not enough to produce a correct definition. Although not implemented, PAL could be modified to allow a small percentage of negative examples to be covered by a clause, as some ILP systems do, producing an equivalent performance. This is left for further research.

Bain [Bai91], has been working to learn a correct definition of illegal positions using non-monotonic learning with extensions over Cigol and Golem.

The idea is that after testing the hypothesis over a large set of examples, the clauses that cover some negative examples are further specialised. This is done by taking all the negative examples that are covered by a clause, generalising them (but now taking the positive examples as negatives), and adding the complement of the head of the resulting clause, to the overgeneral clause. Bain reports the following complete definition for this domain:

```
illegal(A,B,C,D,C,E) ←
        not(legal_special1(A,B,C,D,E)).
illegal(A,B,C,D,E,D) ←
        not(legal_special2(A,B,C,D,E)).
illegal(A,B,A,B,C,D).
illegal(A,B,C,D,E,F) ←
        adj_or_eq(A,E),
        adj_or_eq(B,F).

legal_special1(A,B,A,C,D) ←
        less_than(B,C),
        less_than(D,B).
legal_special1(A,B,A,C,D) ←
        less_than(B,D),
        less_than(C,B).

legal_special2(A,B,C,B,D) ←
        less_than(A,C),
        less_than(D,A).
legal_special2(A,B,C,B,D) ←
        less_than(A,D),
        less_than(C,A).
```

Where *legal_special* covers those exceptions where the white King is in between the Rook and the black King, and *adj_or_eq*(*A,B*) means that file/rank *A* is adjacent or equal to file/rank *B*.

Although not implemented, the same method could be used by PAL. Providing one instance for each disjunct and *less_than*/2 as background knowledge, PAL produces the following clauses for the definition of *legal_special*:

```
legal_special(wk,A,B,wr,C,B,bk,D,B,Pos) ←
```

```
        piece(wk,A,B,Pos), piece(wr,C,B,Pos), piece(bk,D,B,Pos),
        less_than(C,A),
        less_than(C,D),
        less_than(A,D).
legal_special(wk,A,B,wr,C,B,bk,D,B,Pos) ←
        piece(wk,A,B,Pos), piece(wr,C,B,Pos), piece(bk,D,B,Pos),
        less_than(D,A),
        less_than(D,C),
        less_than(A,C).
legal_special(wk,A,B,wr,A,C,bk,A,D,Pos) ←
        piece(wk,A,B,Pos), piece(wr,A,C,Pos), piece(bk,A,D,Pos),
        less_than(C,B),
        less_than(C,D),
        less_than(B,D).
legal_special(wk,A,B,wr,A,C,bk,A,D,Pos) ←
        piece(wk,A,B,Pos), piece(wr,A,C,Pos), piece(bk,A,D,Pos),
        less_than(D,B),
        less_than(D,C),
        less_than(B,C).
```

6.2 PAL vs Foil vs Golem

The main purpose of the concept of illegality described above, was to illustrate the inadequacies of propositional systems. In this section, the applicability of Golem [MF90] and Foil [Qui90] in Chess is tested with a more interesting concept[3]. Learning more useful concepts in Chess can involve concepts like legal moves, checks, etc. as background knowledge. For ground-theory systems, providing background facts for such concepts can be a difficult and time-consuming process. In order to simplify the test, an idea similar to the "exceptions" for the concept of illegal positions was extended with additional background knowledge. The target concept, called *diagonal*, represents those positions where three pieces are in a diagonal line. The basic idea behind this concept can be used to learn concepts such as *pin*, *skewer* or *discovered checks*. In addition to *less_than*/2, the definition of *line*/4 was provided as background knowledge to represent any two positions in a diagonal, vertical,

[3]The latest public versions of Golem (Golem1.0alpha, 1991) and Foil (Foil.2, 1991) were used in the test.

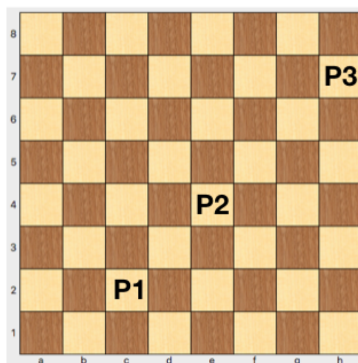


Figure 6.1: Three pieces in diagonal (i.e., *diagonal*(3,2,5,4,8,7).)

or horizontal line. That is, *line*(*X1,Y1,X2,Y2*) means that position $\langle X1, Y1\rangle$ and position $\langle X2, Y2\rangle$ are in a straight line. For ground-theory systems, this represents 1,456 facts for *line*/4 and 28 facts for *less_than*/2. To simplify the concept, only positions where the first piece $\langle X1, Y1\rangle$ was in the lower left corner, the second piece $\langle X2, Y2\rangle$ in the middle, and the third piece $\langle X3, Y3\rangle$ in the upper right corner, were considered as positive. This makes a total of 196 possible positive examples out of 64^3 (262,144) examples in the example space. With this background knowledge, the target definition can be defined as follows (see Figure 6.1):

diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  less_than(X1,X2), less_than(X2,X3),
  less_than(Y1,Y2), less_than(Y2,Y3),
  line(X1,Y1,X2,Y2), line(X2,Y2,X3,Y3).

6.2.1 Learning diagonal with PAL

PAL was tried on this concept. The example positions were described with *piece*(*P,F,R*), where *F* and *R* are the file and rank of a piece, and *P* represents the particular piece. Since the piece and side are not important to this test, the domain for *P* was [p1, p2, p3]. The initial example given to PAL is shown

in Figure 6.1 (i.e., *piece*(p1,3,2,pos1), *piece*(p2,5,4,pos1), *piece*(p3,8,7,pos1)). PAL was provided with background definitions for *line* and *less_than*. The example space was again constrained only to changes in the positions of the 3 pieces. PAL arrived at the following correct definition after generating 3 + and 135 − examples.

diagonal(p1,X1,Y1,p2,X2,Y2,p3,X3,Y3,Pos) ←
    piece(p1,X1,Y1,Pos), piece(p2,X2,Y2,Pos), piece(p3,X3,Y3,Pos),
    less_than(X1,X2), less_than(X1,X3), less_than(X2,X3),
    less_than(Y1,Y2), less_than(Y1,Y3), less_than(Y2,Y3),
    line(X1,Y1,X2,Y2,Pos), line(X1,Y1,X3,Y3,Pos),
    line(X2,Y2,X3,Y3,Pos).

6.2.2 Learning diagonal with Foil

Foil was tested under the same conditions. The complete background facts for *line*/4 and *less_than*/2 were provided as background knowledge. Different positive and negative examples were given to Foil to try to learn this concept (the outputs produced by Foil are also included). Since no negative literals are expected in the final definition, and in order to simplify Foil's search, they were not considered in the test.

- Test_1: With the same examples used by PAL (i.e., 4 + and 135 −), Foil is not able to produce any definition. The reason being the relatively few number of positive examples in comparison with the number of negative examples.

- Test_2: Foil was tested with all the possible positive examples of the target concept (196 +) and the same negatives examples used by PAL (135 −). With them, Foil produced the following incorrect definition:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
      line(X1,Y1,X2,Y2), line(X1,Y1,X3,Y3), line(X2,Y2,X3,Y3),
      line(X1,X3,Y1,Y3), less_than(X1,X3).
  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
      =(X1,Y1), =(X2,Y2).

  E.g., counter example: *diagonal*(3,5,4,4,5,3).

- Test_3: Negative examples were incrementally added to Foil to try to learn the correct definition. Each time an incorrect hypothesis was produced, new negative examples were given to try to correct it. Foil changed its hypothesis several times (none of which was correct). For instance, the following definition was produced by Foil with all the positive examples and the same negative examples required by Golem (159 −) to produce a correct definition (see below):

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  line(X1,Y1,X2,Y2), line(X1,Y1,X3,Y3), line(X2,Y2,X3,Y3),
  less_than(X1,X3), less_than(Y1,Y3).

  E.g., counter example: *diagonal*(1,1,3,1,2,2).

- Final Test: Foil eventually learned the target concept when all the positive examples were given and with 179 carefully selected negative examples. Foil's definition is as follows:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  less_than(X1,X2), less_than(X1,X3),
  less_than(Y1,Y2), less_than(Y2,Y3),
  line(X1,Y1,X2,Y2), line(X1,Y1,X3,Y3).

Although Foil is able to learn the above concept, it was only through a long process of several sessions of analysing why Foil failed and providing new examples to constrain Foil's hypotheses. This example illustrates some of the problems mentioned in chapter 4. Namely, problems of preparing the *right* data (background facts as well as examples) and problems with its information gain heuristic.

6.2.3 Learning diagonal with Golem

Similarly tests were run with Golem. However, the background definition of *line*/4 is non-deterministic (i.e., the same inputs can produce different outputs). Although the definition of *diagonal* is determinate (i.e., all the arguments in the literals are determine given an instantiation of the head), Golem cannot learn *diagonal* with *line*/4). The way to avoid this problem

in Golem, is to design some deterministic background knowledge that could be used to define an "equivalent" definition. Instead of *line*/4, the complete background facts for *abs_diff*/3 (absolute difference between two numbers) was provided as background knowledge. That is, *abs_diff*(*N1,N2,Diff*) means that the absolute difference between *N1* and *N2* is *Diff*. This represents a total of 64 background facts. With this new background knowledge, we expect Golem to arrive to the following equivalent definition:

diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
  abs_diff(X1,X3,D2), abs_diff(Y1,Y3,D2),
  abs_diff(X2,X3,D3), abs_diff(Y2,Y3,D3),
  less_than(X1,X2),less_than(X1,X3), less_than(X2,X3),
  less_than(Y1,Y2),less_than(Y1,Y3), less_than(Y2,Y3).

Similar to Foil, Golem was tested with a different number of examples.

- Test_1: With the same examples required by PAL, Golem produces the following incorrect definition:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
    abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
    abs_diff(X1,X3,D2), abs_diff(Y1,Y3,D2),
    less_than(X1,X2), less_than(X1,Y3).

  E.g., counter example: *diagonal*(1,3,2,2,3,1).

- Test_2: Golem was then given all the possible positive examples (196 +) and the same negative examples used by PAL (135 −). Golem produces the following (still incorrect) definition:[4]

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
    abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
    abs_diff(X1,X3,D2), abs_diff(Y1,Y3,D2),
    less_than(X1,X2), less_than(X2,X3).

[4] Golem's strategy to reduce the size of the *rlgg* clauses is based on the negative examples (see chapter 4). Since Golem was given the same negative examples as Test_1, it should have produced the same definition, however, Golem can sometimes change slightly its definition if it is run twice. The same counter example in Test_1 applies here.

- Test$_3$: Negative examples were incrementally given to try to obtain the solution. Some of its intermediate incorrect hypotheses include (with 196 +, 150 −):

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
  abs_diff(X2,Y3,D2), abs_diff(Y2,Y3,D2),
  abs_diff(X1,Y1,D3), abs_diff(X2,Y2,D3), abs_diff(X3,Y3,D3),
  less_than(X1,X2), less_than(X2,X3).

  E.g., counter example: *diagonal*(4,3,5,6,6,5).

- Final Test: Golem was able to produce a correct definition with all the positive example (196 +) and with 159 negative examples. Golem's definition is as follows:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
  abs_diff(X1,X3,D2), abs_diff(Y1,Y3,D2),
  abs_diff(X2,Y3,D3), abs_diff(Y2,Y3,D3),
  abs_diff(X1,Y1,D4), abs_diff(X2,Y2,D4), abs_diff(X3,Y3,D4),
  less_than(X1,X2), less_than(X2,X3).

Golem is able to learn at least an equivalent definition of the target concept. However, in some cases the background knowledge needs to be defined in terms of deterministic concepts and the resulting definition can become obscure to the user. As with Foil, Golem retains the burden of preparing the background facts.

6.2.4 Golem and PAL

At this point it is convenient to establish the main differences between Golem and PAL, since in both systems the generalisation algorithm is based on *rlgg*.

Example presentation : Examples in Golem are given as ground instances of the goal predicate. In PAL a set of ground unit clauses are given to describe example states where the exact arguments used in the goal predicate are not specified in advance. PAL also uses an automatic example generator to guide its learning process.

Background knowledge : Golem uses ground unit clauses as background knowledge, while PAL uses a restricted form of non-recursive Horn clause from which a finite set of relevant ground atoms are derived for each example description.

Generalisation method : Both systems are based on *rlgg* and both used a variable connection constraint. In addition, Golem uses a functional constraint to limit the size of the clauses, and reduces a clause until no more reductions are possible without covering a negative example. PAL uses an identification of internal structures within a state and exploits this via a labelling mechanism to reduce the length of the clauses and guide the generalisation process. Negative examples in PAL are only used to avoid the production of over-generalisations.

Clause restrictions : Golem is restricted to learn determinate clauses. PAL can learn a limited class of non-determinate, although non-recursive, clauses.

Applicability : Golem is meant to be a general purpose learning system. PAL is only applicable to domains which can be represented by *states* which have an internal structure with well defined components, and with background knowledge definitions which express relations between the components. The general applicability of PAL will be further discussed in chapter 8, where PAL is applied to another domain.

6.2.5 Summary

In this section, Foil and Golem were tried on a particular Chess concept which involves a larger, although still simple, background knowledge. "In principle", both systems could be used in Chess, however the example illustrates some of their main problems. Both systems suffer from the problem of preparing the background facts. In Chess, some concept definitions can involve concepts of legal moves, threats, checks, etc. Defining background facts for such 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. In such cases, appropriate subsets need to be selected to maintain efficiency, which requires a prior knowledge of the training example set. Neither system is used in an incremental way, which

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

In addition to this, Foil's construction of hypotheses is heuristically guided by its information gain measure. This measure is affected by the number of positive and negative examples in the training set (e.g., see Test_1 in section 6.2.2). Like any greedy search algorithm, Foil is prone to make locally optimal but globally undesirable choices. A new implementation of Foil (Foil2, which was used in the test) incorporates *checkpoints*, that is, points where two or more alternatives appear to be roughly equal. If the greedy search fails, the system reverts to the most recent checkpoint and continues with the next alternative. This however, does not eliminate the need to carefully choose the training set to ensure that the desired literal is included in the definition. In domains where the example space can be very large, as with many Chess concepts, trying to select an adequate subset for the required generalisation, is not an easy task and often requires an interactive process of analysing the system failures and adding new examples to try to correct them. Foil, as any other top-down heuristically guided algorithm, has problems with concepts several literals long. We have seen some of them when learning the definition of *diagonal*. Depending on the background knowledge, concepts in Chess which are powerful enough for play may require long clauses. Some of these concepts are given in the next section and in chapter 7.

In addition to the preparation of the background facts, Golem is limited to learning determinate clauses. The non-deterministic nature of obvious background knowledge in Chess, such as legal moves, makes a large number of Chess concepts inherently non-determinate. Finding a determinate counterpart for such concepts is not always easy/possible to do, and sometimes can only be made in terms of such "opaque", although deterministic concepts, that the solution is not longer transparent to the user.

6.3 Learning concepts in Chess

In this section, PAL is used to induce a more useful and interesting set of concepts in Chess. PAL was provided with background knowledge definitions to express the basic rules of the game, like legal moves, checks and check-mates, as well as ways to recognise illegal positions for reducing the number of examples produced by the example generator. The domains of the arguments used to describe Chess positions and a representative example for

each concept were also given.

The information about the domains is as follows:

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

The concept of *illegal* described above contain practically no Chess knowledge and could be equally applied to a different domain. The following background vocabulary was provided to the system (all the background knowledge definitions used by PAL to learn the Chess concepts in this chapter are given in appendix G):

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.

Although this is a richer background vocabulary, it reflects some of the basic concepts that a new Chess player is expected to have, and will be used to learn several Chess concepts.

The number of examples produced by the example generator, as described in chapter 5, is compared with those produced by PAL when additional knowledge about symmetries is taken into account and with those produced by a user. 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 generalisations 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. Even with concepts which are not specific to Pawns and which have 3 axes of symmetry, an over-generalisation can be produced with instances of such concepts involving Pawns. In such cases, the over-generalisation will be rejected by the system (or the user) and the process will continue with other examples (hopefully without Pawns). Since the final concept does not depend specifically on Pawns, the previously rejected examples will be covered by the final definition. The minimum number of examples that a trained user (the author) needed to produce the same definitions was also recorded, as well as the cpu times for these experiments[5]. Results are summarised in Tables 6.2 and 6.3. The contents of these tables will be discussed in section 6.4.

6.3.1 Chess concepts

PAL is able to learn a wide range of Chess concepts, with the above conditions, some of which are described below. Throughout the definitions, *Si* denotes *Sidei* for $i = \{1,2\}$, *Pj* denotes *Piecej* for $j = \{1,2,3\}$, (X,Y) denotes $square(X,Y)$, *Pos* denotes a board position, and ¬ denotes negation. The number of examples generated by the system with/without symmetries and those produced by the user are given in parenthesis after each concept name and arity (i.e., ConceptName/Arity (PAL, PAL w/symmetry, user):). Appendix A, shows a trace of PAL when learning the concept of *can_threat* (described below).

- Threat/7 (23 + 8 −, 10 + 7 −, 2 +): A piece (P1) threatens an opponent's piece (P2) if P1 can capture P2. The system is told that it is a dynamic pattern. *Threat*/6 only applies when the opponent's side is not in check (this last condition is imposed by the definition of *make_move*/5. See chapter 5 and appendix G).

  threat(S1,P1,(X1,Y1),S2,P2,(X2,Y2),Pos1) ←

[5] All the experiments were run in a SUN Sparc-Station 1+.

```
        contents(S2,P2,(X2,Y2),Pos1),
        other_side(S2,S1),
        make_move(S1,P1,(X1,Y1),(X2,Y2),Pos1,Pos2),
        ¬ contents(S2,P2,(X2,Y2),Pos2).
```

- Threat1/7 (18 + 8 −, 8 + 7 −, 2 +): The static counterpart of the above definition can be learned as well. One piece (P1) threatens an opponent's piece (P2) if there is a legal move of P1 to the place of P2. The difference between *threat* and *threat1* is that the former is applicable only when the opponent's side is not in check.

```
threat1(S1,P1,(X1,Y1),S2,P2,(X2,Y2),Pos) ←
        contents(S2,P2,(X2,Y2),Pos),
        other_side(S2,S1),
        legal_move(S1,P1,(X1,Y1),(X2,Y2),Pos).
```

- Fork/10 (22 + 67 −, 9 + 16 −, 3 +): There is a "special" fork if a piece (P3) threatens another piece (P2) and checks the King at the same time (this is the definition given in the example of chapter 5).

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

The definition of *threat1/7*, given above, can be added to the background knowledge. With it, PAL produces the following definition:

```
fork(S1,king,(X1,X2),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos) ←
        in_check(S1,(X1,Y1),P3,(X3,Y3),Pos),
        threat1(S2,P3,(X3,Y3),S1,P2,(X2,Y2),Pos).
```

Both definitions are learned from almost the same number of examples. The only difference is the "length" of the definition, as PAL can use *threat1*'s definition to produce a further "reduction" in the final definition.

- Can_threat/8 (23 + 7 −, 6 + 9 −, 3 +): A piece (P1) can threaten another piece (P2) after making a move to (X3,Y3). This is another dynamic pattern, which is learned after the concept of *threat* has been learned. Appendix A shows the generalisation process followed by PAL in order to learn this concept.

  can_threat(S1,P1,(X1,Y1),S2,P2,(X2,Y2),(X3,Y3),Pos1) ←
  contents(S1,P1,(X1,Y1),Pos1),
  ¬ threat(S1,P1,(X1,Y1),S2,P2,(X2,Y2),Pos1),
  make_move(S1,P1,(X1,Y1),(X3,Y3),Pos1,Pos2),
  threat(S1,P1,(X3,Y3),S2,P2,(X2,Y2),Pos2).

- Can_fork/11 (34 + 29 −, 3 + 0 −, 3 +): A piece (P1) can produce a fork to the opponent's King and piece (P3) after making a move to (X4,Y4). This concept is learned after learning the concept of *fork*.

  can_fork(S1,P1,(X1,Y1),S2,king,(X2,Y2),
  S2,P3,(X3,Y3),(X4,Y4),Pos1) ←
  contents(S1,P1,(X1,Y1),Pos),
  ¬ fork(S2,king,(X2,Y2),S2,P3,(X3,Y3),S1,P1,(X1,Y1),Pos1),
  make_move(S1,P1,(X1,Y1),(X4,Y4),Pos1,Pos2),
  fork(S2,king,(X2,Y2),S2,P3,(X3,Y3),S1,P1,(X4,Y4),Pos2).

- Can_check/8 (20 + 18 −, 2 + 1 −, 2 +): A piece (P1) can check the opponent's King after a moving to (X3,Y3).

  can_check(S1,P1,(X1,Y1),S2,king,(X2,Y2),(X3,Y3),Pos1) ←
  contents(S1,P1,(X1,Y1),Pos1),
  other_side(S1,S2),
  ¬ in_check(S2,(X2,Y2),P1,(X1,Y1),Pos1),
  make_move(S1,P1,(X1,Y1),(X3,Y3),Pos1,Pos2),
  in_check(S2,(X2,Y2),P2,(X3,Y3),Pos2).

- Discovered_check/11 (17 + 44 −, 5 + 6 −, 4 +): 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),
     other_side(S1,S2),
     sliding_piece(P1,(X1,Y1),Pos1),
     ¬ 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).
```

- Discovered_threat/11 (26 + 48 −, 4 + 2 −, 3 +): A piece (P1) can threat an opponent's piece (P3) after moving another piece (P2) to (X4,Y4). It is learned after learning the concept of *threat*.

```
disc_threat(S1,P1,(X1,Y1),S1,P2,(X2,Y2),
             S2,P3,(X3,Y3),(X4,Y4),Pos1) ←
     sliding_piece(P1,(X1,Y1),Pos1),
     ¬ threat(S1,P1,(X1,Y1),S2,P3,(X3,Y3),Pos1),
     make_move(S1,P2,(X2,Y2),(X4,Y4),Pos1,Pos2),
     threat(S1,P1,(X1,Y1),S2,P3,(X3,Y3),Pos2).
```

- Pin/10 (22 + 42 −, 4 + 3 −, 3 +) and (22 + 60 −, 4 + 3 −, 3 +): A piece (P3) cannot move because it will produce a check on its own side by piece (P1). Since PAL is provided with the definition of legal moves, it cannot move a piece to find out that it produces a check (it is an illegal move). Rather, the definition of *pin* is learned from the opponent's perspective. That is, if an opponent's piece (P3) cannot move, it is threatened by P1, and by capturing P3, P1 checks the opponent's King. This is learned after the learning the concept of *threat*.

```
pin1(S1,P1,(X1,Y1),S2,king,(X2,Y2),S2,P3,(X3,Y3),Pos1) ←
        sliding_piece(P1,(X1,Y1),Pos1),
        stale(S2,P3,(X3,Y3),Pos1),
        threat(S1,P1,(X1,Y1),S2,P3,(X3,Y3),Pos1),
        ¬ in_check(S2,(X2,Y2),P1,(X1,Y1),Pos1),
        make_move(S1,P1,(X1,Y1),(X3,Y3),Pos1,Pos2),
        ¬ contents(S2,P3,(X3,Y3),Pos2),
```

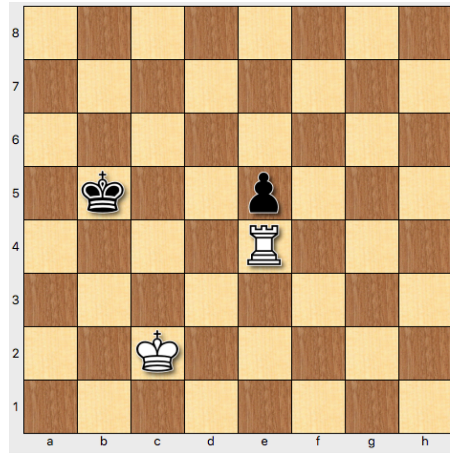


Figure 6.2: A position accepted by Pin1

¬ stale(S2,P3,(X3,Y3),Pos2),
¬ threat(S1,P1,(X1,Y1),S2,P3,(X3,Y3),Pos2),
in_check(S2,(X2,Y2),P1,(X3,Y3),Pos2).

This definition, however, is incorrect as it accepts positions where a Pawn cannot move (i.e., *stale*) because it is "blocked" by another piece rather than being unable to move because it creates a check on its own King (e.g., see Figure 6.2). To correct this, the system must know that the 3 pieces involved in the concept must be in an horizontal, vertical or diagonal line. By adding to the background knowledge the definition of *in_line* to express this, and after learning the concept of *threat*, PAL learns a correct definition. Note that with this new background knowledge, PAL can learn a static pattern definition for *pin*. The concept of *in_line* used in this definition was learned by PAL, after adding a concept in the background knowledge that recognises whenever two pieces are in a vertical, horizontal or diagonal line (*line*), and by introducing a comparison between numbers (*less_than*). Appendix B describes the learning process. The new definition is as follows:

pin2(S1,P1,(X1,Y1),S2,king,(X2,Y2),S2,P3,(X3,Y3),Pos) ←
sliding_piece(P1,(X1,Y1),Pos),
stale(S2,P3,(X3,Y3),Pos),
threat(S1,P1,(X1,Y1),S2,P3,(X3,Y3),Pos),

in_line(S2,king,(X2,Y2),S2,P3,(X3,Y3),S1,P1,(X1,Y1),Pos).

- Skewer/11 $(19 + 55 -, 4 + 22 -, 3 -)$: A King in check by a piece (P1) "exposes" another piece (P3) when it is moved out of check to (X4,Y4). This definition is learned after learning the concept of *threat1*.

  skewer(S1,P1,(X1,Y1),S2,king,(X2,Y2),
  S2,P3,(X3,Y3),(X4,Y4),Pos1) ←
  sliding_piece(P1,(X1,Y1),Pos1),
  stale(S2,P3,(X3,Y3),Pos1),
  in_check(S2,(X2,Y2),P1,(X1,Y1),Pos1),
  ¬ threat1(S1,P1,(X1,Y1),S2,P3,(X3,Y3),Pos1),
  make_move(S2,king,(X2,Y2),(X4,Y4),Pos1,Pos2),
  ¬ stale(S2,P3,(X3,Y3),Pos2),
  ¬ in_check(S2,(X2,Y2),P1,(X1,Y1),Pos2),
  threat1(S1,P1,(X1,Y1),S2,P3,(X3,Y3),Pos2).

6.4 Discussion

6.4.1 Hypothesis language limitations

The hypothesis language is limited to non-recursive concepts (chapter 8 discusses the applicability of PAL and shows how it can be used to learn concepts in a different domain). The background definitions and the concepts learned by PAL assume that there is an implicit example position description. This allows concepts to be represented in a compact and understandable way and provides a mechanism where a selective deduction of relevant facts can be made. PAL cannot include negation, unless a predicate name appears or disappears after a move, it cannot introduce new predicate symbols, and cannot learn recursive definitions. The latter is partly due to the example representation, although in principle, previously generated heads could be included into the list of relevant atoms to allow it to learn recursive concepts. As the target concept head is not specified in advance, previous heads will have to be updated with changes in the arguments used by the current head. Allowing recursive definitions, will required to limit the depth of the resolution steps that are allowed, to limit the number of deductions from the background knowledge. Non-recursive definitions has not been a limitation in Chess.

Concept	Generated Examples	G.E. Symm.	G.E. User	Add. Back. Knowledge
threat/7	23 + 8 −	10 + 7 −	2 +	—
threat1/7	18 + 8 −	8 + 7 −	2 +	—
fork/10	22 + 67 −	9 + 16 −	3 +	threat1 (optional)
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
pin1/10	22 + 42 −	4 + 3 −	3 +	threat
pin2/10	22 + 60 −	4 + 3 −	3 +	threat, in_line
skewer/11	19 + 55 −	4 + 22 −	3 +	threat1
Average (Tot)	57	12.27	2.81	

Table 6.2: Table of results for Chess concepts

Despite its limitations, PAL has been able to learn several Chess concepts in a compact and understandable way from simple example descriptions, which is outside the scope of current machine learning systems. Further discussion on PAL's learning capabilities will be given in chapter 7.

6.4.2 Learning rate

The learning rate is determined by the number of examples produced by the system over particular concepts and the time required to learn them. Table 6.2 gives the number of positive and negative examples produced by PAL, with and without symmetries, the number of examples produced by the user, and the additional background knowledge that is used to learn the concepts. Table 6.3 shows the cpu times require to induce the concepts in a SUN Sparc-Station 1+[6].

The number of examples generated by PAL compares very favourably 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

[6]PAL is written in Quintus Prolog.

Concept	Gen. Exam.	G.E. Symm.	G.E. User
threat/7	3:42	4:44	0:14
threat1/7	1:05	1:38	0:09
fork/10	4:28	3:46	0:11
can_check/8	3:18	1:28	0:11
can_threat/8	8:16	9:02	0:37
can_fork/11	16:11	4:01	0:53
disc_check/11	18:47	11:53	0:36
disc_threat/11	29:00	14:53	1:32
pin1/10	21:00	16:08	0:35
pin2/10	6:14	1:54	0:12
skewer/11	9:18	5:14	0:29
Total	121:19	74:36	5:39

Table 6.3: Table of cpu times (minutes:seconds)

user to almost one fifth in average. Each concept can be learned in a few minutes (considering the extra time required by the user to analyse the examples and definitions produced by PAL). The cpu times of PAL with symmetries, is not so much reduced as the number of examples (only to about half of the time). The reason being that although fewer examples are shown to the user, additional work is done for each one of them (i.e., 7 generalisations as opposed to 1). It is important to note that the cpu times and the number of examples produced by PAL, when the user knows the target concept and provides adequate examples, are sufficiently small to consider using PAL to define a known concept rather than hand-coding it. This is especially significant for the construction of Chess systems which follow a pattern-based approach, as a substantial programming effort is devoted to the definition of patterns. Furthermore, example presentations are accepted by PAL as descriptions of Chess positions which are easier to provide and understand by a human player.

The learning rate is affected by the initial example, the available background knowledge, and the strategy to select examples. All of these factors are further discussed below.

6.4.2.1 Selecting the initial example

Depending on the background knowledge, a greater or lesser number of facts can be derived from particular examples, and some care must be taken to select the starting examples. In general, the initial examples were chosen with the minimum number of pieces required to learn the concept, and in such a way, that only a small number of legal moves could be made. Some fortuitous circumstances in the examples can make PAL fail to produce a desired generalisation at a particular perturbation class, and sometimes this will not be achieved until the next perturbation level. Once the examples were chosen, they were very rarely changed to try to improve the system's performance, and in some cases some features, unexpected by the user, emerged from the examples (see for example appendix A). None of the initial examples involve more than 5 pieces and none of the concepts involve more than 3 pieces. PAL's example space is generated dynamically and most of the time irrelevant pieces were identified early in the learning process. No systematic study was made on the effect of adding irrelevant pieces when learning concepts, but it is expected to decrement PAL's performance. Similarly, concepts which involve a larger number of pieces in their definitions will clearly affect PAL's performance, as the example space grows exponentially with the number of pieces.

6.4.2.2 Available background knowledge

As seen with the definition of *pin* the amount of background knowledge available to the system can affect the quality of the concepts. In some cases, the available background knowledge is not enough to produce a correct definition, within the hypothesis language, and additional knowledge must be included. A starting core of background knowledge definitions can be used to learn intermediate concepts, which in turn can be used to learn further definitions (see appendix B and F). One important question is how much starting knowledge to include in order to learn a wide range of concepts. This will be further discussed in chapter 7, where the hypothesis language limitations of PAL are further discussed. It is important to note that, with reasonable domain knowledge, PAL could learn, in principle, only static patterns. This however, can become cumbersome when trying to define appropriate background definitions for learning one-ply concepts like, *can_threat*, *can_check* or *can_fork* as described above. As the background knowledge increases, PAL's

performance will tend to decrease, despite its pattern-based approach.

6.4.2.3 Selection of examples

The number of examples produced and explored by the system depends on the strategy by which the example space is traversed. Section 5.3 describes an evaluation function which is used to sort perturbation classes of the same perturbation level. Different changes in the evaluation function were tried to test their impact in the learning rate. The changes included:

- Ignore the evaluation function.
- Reduce the importance of constants (i.e., change $varcte(Arg)$ value from 20 to 10 when Arg is a variable).
- Prefer changes where an argument appears in a small number of literals (i.e., change the evaluation function from, $value(Arg) = varcte(Arg) + domsize(Arg) - nliterals(Arg)$, to $value(Arg) = varcte(Arg) + domsize(Arg) + nliterals(Arg)$).

Since the examples produced by PAL depend on the particular concept definition, different sorting algorithms can improve or worsen the system's performance. As long as PAL continues to proceed by levels, all of the above changes make little impact in the number of examples that are generated by the system, however, they can change the number of explored examples, and subsequently the time required to arrive at the intended definition. In particular, if the only perturbation class that can produce an example for the required generalisation is last in the current perturbation level, then the system can spend a long time trying all the possible examples of the preceding classes (although only very few might be shown to the user). For the above Chess concepts, the sorting strategy described in chapter 5, produces a slightly smaller number of examples and faster times. This suggests that the example structure is fairly robust in the sense that the sorting of levels did not change greatly PAL's performance, although further tests are needed.

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. Taking advantage of the symmetric properties of some Chess concepts significantly reduces the number of examples presented to the user and increases the generalisation

steps. Another possibility is to include knowledge about translations of positions (i.e., shift the positions of all the pieces one or more squares in the board). Knowing that a concept is invariant with respect to shifted positions can be used to increase the generalisation steps by generalising between several shifted positions of a positive example at the same time.

Although not implemented, the invariance of a clause with respect to an axis of symmetry can be deduced from the examples and the concept definitions. For instance, the example generator can produce an example which is symmetric along a particular axis of symmetry with respect to the last positive example. If this symmetric example is accepted as positive and does not create an over-generalisation, then the axis of symmetry could be stored and used in the future. For each new example all its symmetric examples (along the stored axes of symmetry) can be used in the generalisation process to increase the generalisation steps. A similar process could be used to 'discover' invariance of a concept with respect to shifted positions.

6.4.3 Learning concepts in other domains

PAL is applicable to pattern-based domains where examples are given by a set of components describing a particular state of a system and the background knowledge use the description of the states (examples) to express relations between the components. PAL has been applied to the 8-puzzle, where the position of each tile in the board is used to describe a current state, background knowledge has relations between the positions of the tiles (e.g., adjacent) which are used to induce new concepts, and the example generator produces new examples by changing the positions of the tiles. Similarly, PAL has been used to learn simple concepts in card games. A concept is described by a set of cards, the background knowledge has relations between cards (e.g., less-than, same-suit, same-colour, etc.), and the example generator changes the possible values and suits of the cards. In chapter 8, PAL is used to learn qualitative models of dynamic systems, which shows that PAL can be successfully used to other domains besides Chess.

6.5 Summary and conclusions

Domains like Chess, where a relatively large amount of background knowledge can be used to induce inherently non-determinate concepts, are still

inadequate for other ILP systems. PAL is flexible enough to allow the adding/removing of different background knowledge definitions and learn concepts like *illegality*, with almost no background knowledge and with a simpler Chess board representation, and concepts like *pin* or *discovered_threat*, with a much richer background vocabulary.

The efficiency of PAL can be affected by the number of pieces in the target concept, the example generation process, and the current background knowledge. A target concept with a large number of pieces can take a long time to learn, as the example space grows exponentially with the number of pieces. The learning rate of PAL can be improved by adding extra domain knowledge to increase the generalisation steps (e.g., by considering geometrical properties of the board). The available background knowledge plays a central role in the induction process. One interesting question is how much starting knowledge is required to learn a wide range of Chess concepts. This is further discussed in chapter 7.

This chapter shows that a good sample of Chess concepts, outside the scope of current ILP systems, can be learned by PAL. It also shows that they can be learned in a relatively short time from board descriptions which are easy for a human player to understand and provide. An obvious question is whether the concepts learned by the system can be used for playing Chess. To answer this question, a correct playing strategy for a simple endgame was designed with concepts learned by PAL. This is the topic of the next chapter.

Chapter 7

KRK: a case study

Chapter 2 shows how most Chess systems which follow a knowledge intensive approach have designed their playing strategies around the recognition of patterns. A substantial programming effort is spent in the selection of the appropriate patterns. In the previous chapter, PAL was used to learn several Chess patterns. This chapter tests whether the patterns learned by the system can be used to construct a correct playing strategy[1]. Due to the complexity of Chess, the correctness of playing strategies can only be tested over 'simple' endgames. King and Rook against a sole opponent King (KRK) was chosen, since a simple playing strategy based on the recognition of patterns can be easily constructed and, more importantly, verified.

7.1 Design of a KRK playing strategy

Mechanisation of the King and Rook against the sole opponent King has a long standing history. Torres y Quevedo in the last decade of the 19th. century constructed an electro-mechanical device to play KRK, which checkmates from any starting position subject to certain qualifications. The initial positions must have the Rook dividing both Kings and its file must be either ≤ 3 or ≥ 6. Michie describes a reconstruction with 6 decision rules and shows how in worst cases the strategy checkmates in 62 moves [Mic77]. Chapter 2 describes Bramer's [Bra77], Huberman's [Hub68], and Michie and Bratko's [Bra82, Mic76] approaches to Chess, all of which were applied to KRK, and

[1] Correct in the sense that it selects moves that will ultimately lead to victory from a winning position against any sequence of play by the opponent.

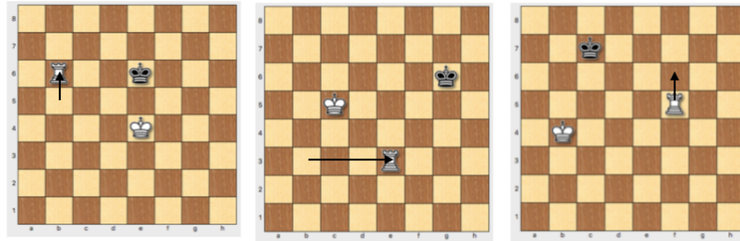


Figure 7.1: Three strategy rules for the KRK endgame

used pattern recognition as the driving force for their playing strategies.

Designing a pattern-based playing strategy involves the following steps:

(**1**) Design the general strategy of the game,

(**2**) Learn the patterns involved in it,

(**3**) Check the strategy and return to (**1**) if necessary.

There is a trade-off between the number of patterns used by the playing strategy and the amount of search that the strategy performs. A larger number of patterns tends to compensate for smaller searches. A simple strategy which involves 1-ply search was chosen for this endgame. It consists of ordered rules with the following format:

> **if** pattern(s) in current position,
> **move** piece to form a new position,
> **ensure** pattern(s) in new position.

For instance, a rule for a KRK playing strategy can be as follows:

> **if** both Kings are in opposition, (Patt_1)
> **move** Rook to new position,
> **ensure** the opponent's King is in check, (Patt_2)
> and that the Rook is not threatened. (Patt_3)

Where Patt_1, Patt_2 and Patt_3 represent particular patterns of the game.

A broad winning strategy in the KRK endgame is to force the opponent's King to the edge, or into a corner if necessary, and then deliver mate in

while making sure that stalemate is never created or the Rook left undefended under attack
repeat until mate

- **if** the Kings are in opposition, try to force the opponent's King to move backwards
- **else** try to move the King to one square away from opposition.
- **else** try to move the King closer to the opponent's King, while maintaining the Rook dividing both Kings.
- **else** try to move the Rook closer to the opponent's King, while maintaining the Rook dividing both Kings.
- **else** try to move the Rook to divide both Kings either vertically or horizontally.

Table 7.1: KRK strategy

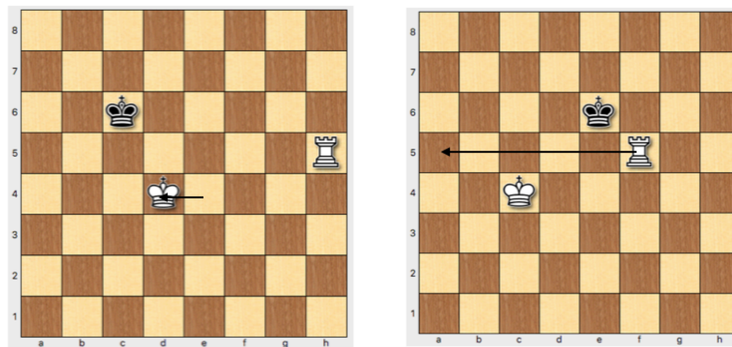


Figure 7.2: Two strategy rules for the KRK endgame

a few moves. The basic principles and the main concepts learned by the system for this strategy are outlined in Table 7.1. Figure 7.1 illustrates three of the rules used in the playing strategy for the KRK endgame: if both Kings are in opposition, move the Rook to form an L shaped pattern (left); if both Kings are on the same side, move the Rook to divide both Kings (center); if the Kings are already divided, move the Rook closer to the opponent's King (right). Similarly, Figure 7.2 ilustrates another two rules: with the Rook dividing both Kings, move the King to be "almost" in opposition with the opponent's King (left); if the Rook is threatened (right), in a wr-bk-wk pattern, move the Rook to the edge away of the threat into a wr-wk-bk pattern. The complete strategy, comprising 18 rules, is detailed in appendix D and the concepts learned by the system are fully described in appendix C.

7.1.1 Background knowledge

Chase and Simon [CS88] suggest that some patterns are built around visual features, such as spatial proximity, as well as Chess functions. This is more noticeable in Chess endgames, where some concepts require geometrical properties of the board, e.g., if a piece is closer or moves closer to another piece or even to a particular place, like the queening square[2]. The concept of *distance* between pieces was defined and included as background knowledge for this purpose. As described in chapter 5, concept definitions are constructed from facts derived from background knowledge which include at least one of the atoms of the example description. In general, other background knowledge definitions can be included which do not involve *directly* any of the atoms from the example description. This is the case when we want to produce comparisons between distances of pieces. The definition of *less_than* was included in the background knowledge but used only after deriving all the pattern-based facts[3].

So far, only moves that introduce a new predicate name or remove an existing one have been included in the definition of dynamic patterns. This condition is relaxed for geometrical knowledge, such as *distance*, which can change as a consequence of a move, but which do not necessarily disappear. A direct consequence of this relaxation is that, in some cases, very long

[2]The square where a Pawn is promoted.

[3]In practice, it takes compatible facts (facts with the same arity and functor symbol) and compares their corresponding arguments (arguments which appear at the same place).

clauses can be produced, affecting the system's performance. For instance, in a KRK endgame, there can be 30 plausible moves in a single position (14 Rook moves and 8 moves for each King), each associated with 2 new distances (to the other two pieces) and 7 new comparisons (less_than(A,B) or less_than(B,A), between the 2 new distances and between them and the original 3 distances between pieces). This can produce clauses with more than 200 literals, many of which have the same name and predicate symbol. One way of reducing an explosive generation of facts is by constraining the background knowledge definitions. For instance, instead of defining *distance* between any two pieces in the board, *distance* is defined only between certain pieces (e.g., *restr_distance* and *restr_manh_dist* defined below). This was used to learn some of the concepts involved in the playing strategy.

In addition to the background knowledge described in section 6.3, the following "geometrical" background vocabulary was provided to the system. Table 7.2 specifies which of this background knowledge was available to the system in conjunction with the rest of the background knowledge to learn particular concepts.

less_than(*Num1,Num2*):
    Num1 is less than Num2.
distance(*Place1,Place2,Dist,Pos*):
    Distance (Dist) between a piece in Place1 and a piece in Place2.
restr_distance(*Place1,Place2,Dist,Pos*):
    Distance (Dist) between a King in Place1 and the Rook in Place2.
manh_dist(*Place1,Place2,Dist,Pos*):
    Manhattan distance between two pieces.
restr_manh_dist(*Place1,Place2,Dist,Pos*):
    Manhattan distance between a King and the Rook.
coordx(*Place,X,Pos*):
    The file of a piece in Place.
coordy(*Place,Y,Pos*):
    The rank of a piece in Place.

7.2 KRK concepts

As in chapter 6, the examples generated by PAL were compared with those shown to the user when symmetric properties of the concepts are considered,

Concept	Generated Examples	Gen. Exs. Symmetry	G.E. Man.	No. Cls.	Additional Back. Knowledge
threatkR/7	13 + 9 −	2 + 1−	2 +	1	—
rook_divs/10	72 + 15 −	46 + 16 −	28 +	4	coordx, coordy
opposition/8	11 + 43 −	2 + 0 −	2 +	2	distance
alm_oppos/9	16 + 15 −	1 + 0 −	2 +	1	distance, manh_distance
r_edge/4	36 + 8 −	29 + 6 −	8 +	4	—
ks_sside/10	95 + 5 −	60 + 10 −	28 +	4	coordx, coordy
l_patt/12	69 + 148 −	15 + 2 −	14 +	2	opposition
rKk/10	70 + 18 −	55 + 19 −	28 +	4	coordx, coordy
rkK/10	64 + 9 −	57 + 20 −	28 +	4	coordx, coordy
closerKk/10	15 + 1 −	2 + 0 −	4 +	1	distance
rcloserKk/10	18 + 3 −	2 + 0 −	4 +	1	manh_distance
awayRk/10	27 + 4 −	8 + 3 −	7 +	1	restrict_distance
rawayRk/10	26 + 16 −	7 + 0 −	7 +	1	restr_manh_dist
rcloserRk/10	48 + 15 −	18 + 4 −	10 +	2	restr_manh_dist
in_lineRk/7	32 + 10 −	9 + 2 −	6 +	2	coordx, coordy
closeRK2/8	8 + 3 −	5 + 2 −	2 +	1	distance
distkR6/8	13 + 10 −	7 + 2 −	2 +	1	distance
Average per clause (Tot)	26.8	11.4	5.1	—	—

Table 7.2: Results for the KRK

and with those produced by a trained user (the author) to obtain the same definitions. As some of the concepts involve disjunctive definitions, although the concept as such is completely symmetric along the 3 axes of symmetry, each disjunct can have only a single axis of symmetry. To learn such concepts, each disjunct was learned separately and the user was prompted for the particular symmetry axes. Table 7.2 shows the main results. The first column has the name of the concept to learn, together with its arity. The next three columns have the number of examples produced by the system without/with symmetries and by the user. The fifth column has the number of clauses required to defined the concept and the last column has the additional background knowledge added to the system. The last row shows the average number of examples generated per clause.

As in chapter 6, the average number of examples presented to the user is greatly reduced when knowledge about symmetries is included and in some cases, it is smaller than the number of examples that could be manually provided to achieve the same results. Not all the concepts can take advantage of all the symmetries. For this domain, concepts described with 4 disjunctive clauses have only one axis of symmetry on each disjunct. Similarly, the disjuncts of two clause concepts are symmetric along two axes and one clause concepts are symmetric along 3 axes. In general, the less orientation dependent the disjuncts of a concept are, the greater generalisation steps can be achieved.

7.3 Checking the strategy

The strategy outlined in Table 7.1 and fully described in appendix D can checkmate the opponent's King for all possible starting positions where the winning side makes the first move (roughly 40,000, considering reflections and rotations), regardless of the opponent's moves. The correctness of the strategy was tested by following every possible path of a starting position. A simple depth-first program using bit sets, to mark positions from which check mates were forced by the strategy for all the possible paths, was used to verify the correctness of the strategy. Each path was followed and checked for loops, captures or stalemates, until a check mate was delivered or a marked position encountered. The algorithm to check the correctness of the strategy is described in Table 7.3.

Checking the correctness of a playing strategy for Chess can only be made for a limited number of Chess endgames due to the size of the search space[4]. Finding the correct strategy took a couple of weeks, although all the patterns can be learned in one day. The original strategy consisted of 11 rules with 12 concepts and changes were made by the author to avoid loops, captures or stalemates. "Tuning" the strategy involved changing the conditions of rules, changing the order of the rules, creating a new rule with the available patterns, or learning a new pattern to include it as an extra condition to a rule or to create a new rule. In total 5 new concepts were created, there was one rule reordering and 6 new rules were added. Some errors in the original strategy included not checking for stalemates or possible captures of the rook.

[4]The most recent database generated for an endgame has been made for the KRBKNN endgame involving a search space of 6×10^9 positions.

- Given the set of rules for the KRK endgame.
- **while** there are unmarked winning-side-to-move (WSM) positions
  - set *NewPos* = an unmarked WSM position
  - mark *NewPos*
  - set *Path* = [*NewPos*]
  - **call** *win-move*

win-move

make a winning-side move using the KRK-rules
if the new position is check mate
then stop
else call *lose-move*

lose-move

if the losing-side King is stalemated **or**
  can safely captured the Rook **or**
  can move to a position which is in *Path*
then return failure
else while a losing-side move can move into an
      unmarked position
  make a losing side move to an unmarked
      position $NewPos_i$
  mark $NewPos_i$
  set *Path* = [$NewPos_i$ | *Path*]
  call *win-move*

Table 7.3: Algorithm for checking the correctness of the KRK-rules

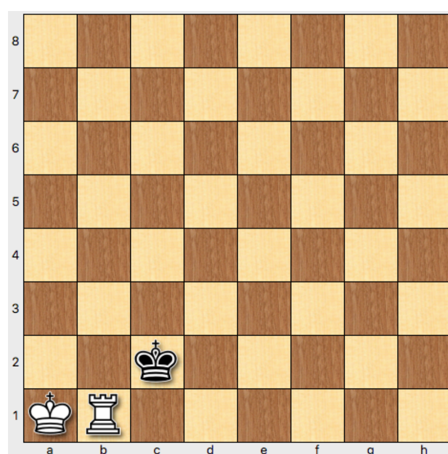


Figure 7.3: A worst case example for the KRK strategy

New rules were added to avoid loops and contemplate special border cases. In general, it was relatively easy to identify a flaw in the strategy and correct it (sometimes learning a new concept). A larger amount of time was spent in checking the correctness of the strategy each time a new change was made.

7.3.1 Improving the strategy

Although the strategy will checkmate the opponent's King for every starting position with the winning side to move, the longest "path" involves 57 moves[5] (see Figure 7.3).

The design of the above strategy follows a very naive approach. It can be improved if the concept of the area on which the opponent's King is "confined" by our Rook is included in the background knowledge. This can be used to learn how to "squeeze" the opponent's King's area until mate can be delivered in a few moves. The design of such strategy was developed in 5 days, it uses 7 concepts of the previous strategy with 8 new concepts in 19 rules. Full details of the concepts and the rules of the strategy, are given in appendix E.

---

[5]The worst case is when the opponent's King is in file/rank 3 and the winning-side King is in file/rank 1. The strategy takes at most 9 moves to force the opponent's King to move "backwards". Thus, it takes 54 moves to mate plus 3 initial moves in the worst case to get the rook in the right position.

while making sure that stalemate is never created or the Rook left undefended under attack,
repeat until mate.

- **move** the Rook to reduce the area of the opponent's King until it is trapped in a corner.
- **else**, move the King to keep the current area, closer to the Rook and/or closer to the opponent's King.
- **if** the opponent's King is trapped in a corner, deliver mate in a few moves.

Table 7.4: New KRK strategy

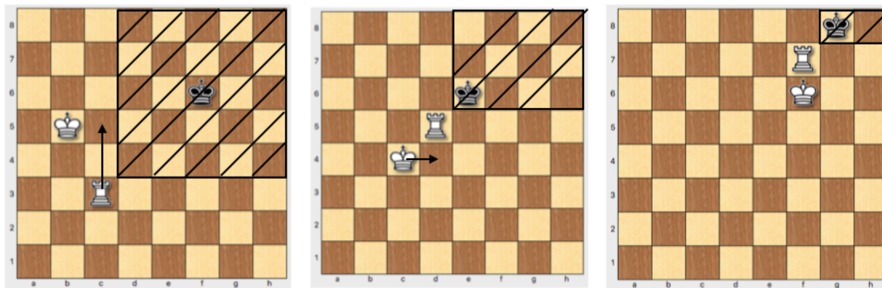


Figure 7.4: Squeeze the area of the opponent's King (left) and keep the area (center) until it is trapped into a corner (right)

The new strategy is outlined in Table 7.4 and illustrated in Figure 7.4. It does not use Manhattan distance as background knowledge to learn the new concepts, requires a smaller number of patterns, and was developed in less time. Although it is far from being optimal, the new strategy safely mates the opponent's King from every starting position of the winning side in less than 50 moves. Trying to optimise the strategy would involve a much bigger effort that requires the detection of many exceptions. Zuidema reports how trying to improve the strategy of a program for the same endgame, resulted in an overburdened program, with smaller improvements produced only after a great deal of programming effort [Zui74]. Bramer also reports the nuisance of special cases and how the number of patterns was almost doubled when constructing a sub-optimal strategy for this endgame [Bra75]. In general, refining the algorithm and exceptions of rules can only be made at a high price in terms of time to do it and computation resources. A tool that allows a fast induction of patterns from examples using a high level language can facilitate this task.

Bain and Muggleton are trying to use Golem to learn an optimal strategy for this endgame from a complete database where positions are classified by their optimal N-depth-ply to check mate. The idea is to learn rules to distinguish different N-depth-ply positions. All the positions at depth N are given as positive examples and the rest of positions as negative examples. These can be used to induced rules that could play optimally by moving from positions classified by the rules of depth N to positions classified of depth $N-1$. So far, they have succeeded for depths 0 and 1. However, the deterministic nature of the background knowledge used by Golem, might be inappropriate to distinguish between positions of different depths and the resulting rules may be difficult to understand.

7.4 Discussion

This chapter has shown how PAL can be used to learn useful patterns. Its pattern-based knowledge representation allows it to derive only a finite set of "relevant" atoms from the background knowledge. The "complexity" of the background knowledge which affects all the *rlgg*-based algorithms, becomes relevant when learning dynamic patterns with background knowledge definitions which do not follow the pattern-based approach (e.g., *less_than*).

Additional background knowledge can be used, not only to correct con-

cept definitions, but also to learn more "powerful" concepts that can be used in the design of better playing strategies. Chapter 6 and appendix B show how adding to the background knowledge the concept of vertical, horizontal or diagonal line between two pieces (*line*) and a way to compare numbers (*less_than*), can be used to learn the concept of 3 pieces in a vertical, horizontal or diagonal line, and amend the definition of *pin*. In this chapter, the KRK playing strategy was improved by adding the concept of *confined_area*, which is used to learn the concept of *squeeze*, to reduce the area on which the opponent's King is confined. One of the initial aims of this research was to learn Chess concepts using just the rules of the game. In practice, it is clear that additional "geometrical" knowledge is required for some concepts. An important question is how much knowledge is needed?

Background knowledge definitions which express the rules of the game with some "primitive" concepts like *distances* and *less_than* seems to be adequate for learning a wide range of concepts. In particular, they were used to learn patterns to decide whether a Pawn could safely promote, in a King and Pawn against King (KPK) endgame, by solely moving the Pawn (results are summarised in appendix F). The background knowledge included distance and Manhattan distance between 2 pieces, distance to the queening square (the square where the Pawn is promoted) and *less_than*. Several sub-patterns were learned first, as constructing blocks for more complicated patterns (see also the definition of *in_line* in appendix B), which shows how PAL could be used to learn more complicated concepts starting from 'simple' ones. The real question then is, how simple can we go?

There is a bottom line to the nature of the concepts that PAL can learn. In particular, PAL is unable to learn the concept of *confined_area* with say, background definitions of multiplication between numbers and distances to the borders of the Chess board. PAL cannot distinguish between all the different multiplications between numbers to produce only the desired area. However, giving a weaker version of area (i.e., all the possible areas that a piece can form in a board, which gives a restriction on the possible multiplications) and *less_than*, PAL can learn the following version of *confined_area*[6]:

```
confined(S1,rook,square(X1,Y1),S2,king,square(X2,Y2),
        A1,A2,A3,bx1,bx8,by1,by8,Pos) ←
    contents(S1,rook,square(X1,Y1),Pos),
```

[6]The complete definition involves four disjuncts, one for each corner.

```
contents(S2,king,square(X2,Y2),Pos),
sliding_piece(rook,square(X1,Y1),Pos),
other_side(S1,S2),
area(square(X1,Y1),bx1,by1,A1,Pos),
area(square(X1,Y1),bx8,by8,A2,Pos),
area(square(X2,Y2),bx1,by1,A3,Pos),
area(square(X2,Y2),bx8,by8,A4,Pos),
less_than(A4,A2), less_than(A1,A3).
```

where *area(Place,Bord1,Bord2,Area,Pos)*, means that a piece in *Place*, forms an area of size *Area*, with borders *Bord1* and *Bord2* in example position *Pos*. In the definition, *bx1*, *bx8*, *by1*, and *by8* are the left, right, bottom, and top borders, respectively. That is, the Rook's lower left hand side area is smaller than that of the opponent's King, while its upper right hand side area is bigger.

7.5 Conclusions

There have been some attempts to use a pattern-based approach in the design of playing strategies in Chess [Bra77, Bra82, Hub68, Wil79]. The emphasis on these systems has been on ways to use and combine patterns, while a substantial programming effort is actually employed in the definition of the *right* patterns. In this chapter, it is shown how PAL can learn patterns that are useful for the design of a playing strategy. PAL provides a useful tool for the development of playing strategies as concepts are relatively easy to learn from descriptions of Chess positions, from which a pattern can be more easily recognised by the user.

The background knowledge used in chapter 6 was used to learn several interesting concepts, like *discovered_attacks*, *forks*, *pins*, etc. From this chapter, it is clear that the background vocabulary needs to be extended to include geometrical definitions, as some concepts depend on spatial relations as well.

The pattern-based knowledge representation allows PAL to learn several concepts which are difficult to learn by other ILP systems, however, it also imposes a constraint of what is learnable by PAL. The characteristics of the background knowledge determines what can be learned by the system. Some background knowledge might be insufficient to learn particular concepts (e.g., learn *confined_area* with *multiplication*), while other background definitions

(e.g., *confined_area*), might allow the system to learn very powerful concepts (e.g., *squeeze*).

Chapter 8

Learning qualitative models

Although this research is focussed on learning Chess concepts, this chapter shows how PAL can be applied to a different domain by changing its background knowledge. PAL is applicable to structural domains where examples are given by a set of components (Chess pieces) describing a particular state of a system (a Chess board), and the background knowledge uses the description of the states (examples) to express relations between the components. Qualitative simulation domains are like Chess in the sense that examples can be given by a set of components (i.e., qualitative variables) describing a particular state of a system, and the background knowledge can express relations between those components (e.g., QualVar1 + QualVar2 = QualVar3).

In a recent paper, Bratko et al. [BMV92] report how Golem was used to learn a qualitative model of a simple dynamic system (the U-tube). The hypothesis is that qualitative models are easier to learn than differential equations (attempts to learn linear "quantitative" equations have been made in systems like Bacon [LBS83]). In this chapter PAL is used to learn a qualitative model for the U-tube. A comparison between PAL and Golem is given. In particular, it is shown that some of the problems encountered by Golem in this domain (i.e., non-determinate target concept, the need for very large number of background facts and the use of generalisation against background examples) can be overcome by PAL.

Section 8.1 describes the QSIM formalism first introduced by Kuipers [Kui86]. Section 8.2 explains the characteristics of the U-tube, a simple dynamic system where Golem was tested. Section 8.3 reviews the experiments with Golem and discusses its results. Section 8.4 shows how PAL is applied to this domain, explains its results, and compares its results with Golem. Fi-

nally, in section 8.5 conclusions and future research directions are indicated.

8.1 Qualitative simulation

Qualitative models have been shown to be better suited for some tasks than traditional numerical models (see for example [BML89, Pea88]). QSIM [Kui86] is a qualitative formalism used for simulating dynamic models and was taken as the basis for the experiments with Golem. In QSIM, a qualitative simulation of a system starts with a description of the known structure of the system and an initial state, and produces possible future states of the system. The structure of the system is described by a set of physical parameters (or qualitative variables) and a set of constraints describing how those parameters are related to each other. 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 et al. [BMV92], a QSIM qualitative simulation algorithm can be sketched in Prolog as follows:

```
simulate(State) :- % Start with State
     transition(State, NextState), % Move to next state
     simulate(NextState).

% State = state( Variable1, Variable2, ... )

transition( state(V1, V2, ... ), state(NewV1, NewV2, ... )) :-
     trans(V1, NewV1), % Model independent
     trans(V2, NewV2),
     ... ,
     legalstate(NewV1, NewV2, ... ). % Model-dependent
```

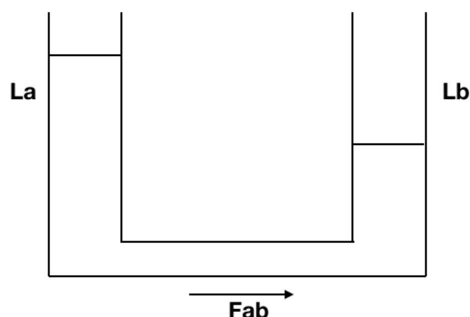


Figure 8.1: The U-tube

Trans is a non-deterministic relation that generates possible transitions of the variables and is defined as part of the QSIM theory [Kui86]. The model of a particular system is defined by *legalstate* which imposes constraints on the values of the variables.

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:

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

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

```
legalstate(... ) :–
     constraint1(... ),
     constraint1(... ),
     ...
```

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

8.2 The U-tube

The experiment consisted of learning a qualitative model for the U-tube. The U-tube (illustrated in Figure 8.1) consists of two containers, A and B,

connected with a pipe and filled with water to their corresponding levels La and Lb. If the flow from A to B is Fab (and from B to A, Fba), the difference in the level of the containers A and B is *Diff*, and the pressure along the pipe is *Press*, then a qualitative model for the U-tube, defined by a set of constraints on the physical parameters of the system, can be formulated as follows:

$$\frac{d}{dt}La = Fba$$

$$\frac{d}{dt}Lb = Fab$$

$$Fab = -Fba$$

$$Diff = La - Lb$$

$$Press = M_o^+(Diff)$$

$$Fab = M_o^+(Press)$$

If we are not explicitly interested in the pressure, the two equations involving M_o^+ can be simplified into one:

$$Fab = M_o^+(Diff)$$

Appropriate landmark values for all the variables of this model for the U-tube are:

- La: *minf, 0, la0, inf*
- Lb: *minf, 0, lb0, inf*
- Fab: *minf, 0, fab0, inf*
- Fba: *minf, fba0, 0, inf*
- Diff: *minf, 0, diff0, inf*

where *inf* and *minf* are infinite and minus infinite respectively, and *la0*, *lb0*, etc., are the initial values of the variables. These symbolic values are ordered from left-to-right with a *less-than* relation.

The qualitative variables can be represented as follows: *Name:Value/Deriv*. Where *Name* identifies each variable, *Value* can take a landmark or an interval of landmark values, and *Deriv* can be *inc*, *std*, or *dec*. For example,

Time	La	Lb	Fab	Fba	Diff
t0	la0/dec	lb0/inc	fab0/dec	fba0/inc	diff0/dec
(t0,t1)	0..la0/dec	lb0..inf/inc	0..fab0/dec	fba0..0/inc	0..diff0/dec
t1	0..la0/std	lb0..inf/std	0/std	0/std	0..diff0/std
(t1,inf)	0..la0/std	lb0..inf/std	0/std	0/std	0..diff0/std

Table 8.1: Behaviour for the U-tube

if in the initial state the value of level La is equal to $la0$ and it is decreasing (dec). This can be represented as follows:

$$La = la0/dec$$

In the time interval that follows, La is between 0 and $la0$ and decreasing. This is described as follows:

$$La = 0..la0/dec$$

The qualitative simulation of the U-tube, if in the initial state there is more water in container A, is given in Table 8.1.

The standard qualitative model used in [BMV92] 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).                                    (8.1)
```

where [c(...), c(...), ...] represent a list of corresponding values. In the *add* constraint they say that whenever $Lb = lb0$ and $Diff = diff0$, $La = la0$ (i.e., $lb0 + diff0 = la0$).

8.3 Experiments with Golem

In Golem the definitions of the constraint predicates were tabulated into tables of ground facts. The following simplifications were made:

- Golem did not consider corresponding values. This makes the repertoire of constraints available to Golem weaker (i.e., knowing $lb0 + \mathit{diff0} = la0$, can be used to constrain the possible values that the variables in the *add* constraint can assume). On the other hand a smaller number of background facts needs to be generated (i.e., Golem would otherwise need to generate all the possible *add* predicates with the possible corresponding values as background facts).

- The *mult* constraint was not tabulated. A very large number of facts must be tabulated for this constraint, which makes it problematic for a system like Golem. Although not needed on this domain, it might be required in another domain.

- The *add* constraint requires a very large number of ground facts and was not tabulated explicitly. Instead it was replaced by three more economical relations: *norm_mag* (normalises a given qualitative value with respect to a landmark value), *lookup_consist_table* (lookup table for adding signs), and *verify_add_deriv* (lookup table for adding derivatives). This means that the *add* constraint cannot be included in the rules generated by Golem.

- Consistency of infinite values in the *add* constraint were ignored. The *add* constraint checks for consistency of additions involving infinite values. That is, if we add two infinite values, the result must be infinite as well (i.e., add(inf,inf,inf)). Ignoring this constraint does not affect the results for this domain.

The following constraint predicates were thus tabulated in Golem:

range(F, Range).
deriv(F1, F2).
m_plus(F1,F2,[ ]).
m_minus(F1,F2,[ ]).
norm_mag(FunName, QValue, 0, NormalisedQValue).
lookup_consist_table(NormV1, NormV2, NormV3).
verify_add_deriv(Dir1, Dir2, Dir3).

```
range(la:0/std,0..la0/std).
range(la:0/std,0..la0/dec..std).
range(la:0/std,0..la0/dec..inc).
range(la:0/std,0..la0/std...std).
...
m_plus(fab:0/dec,fba:0..inf/dec,[ ]).
m_plus(fab:0/dec,fba:0..inf/std,[ ]).
m_plus(fab:0/dec,fba:0..inf/inc,[ ]).
m_plus(fab:0/dec,fba:fba0..0/dec,[ ]).
...
deriv(la:la0..inf/dec,fba:fba0..0/std).
deriv(la:la0..inf/dec,fba:fba0..0/inc).
deriv(la:la0..inf/inc,fba:fba0..0/dec).
deriv(la:la0..inf/inc,fba:fba0..0/std).
...
norm_mag(la,la0,0,pos).
norm_mag(la,inf,0,pos).
norm_mag(la,minf..0,0,neg).
norm_mag(la,0..la0,0,pos).
...
```

Table 8.2: Examples of background facts for Golem

Some of the background facts, taken from [BMV92] are given in Table 8.2 to illustrate their meaning. In total 5,408 background facts from the above predicates were provided to Golem.

Golem used four positive examples, the first three from Table 8.1, plus an additional example state described as follows:

La:0/std, Lb:0/std, Fab:0/std, Fba:0/std, Diff:0/std.

In addition, six negative examples were manually provided by the authors to Golem. Their first attempt, with only five negative examples, produced an over-general model. Golem (see page 48) uses the negative examples to reduce the clause found by the *rlgg* algorithm until no more removals can be performed without covering a negative example. This means that Golem can over-generalise if a small set of negative examples are initially chosen.

In the experiments with Golem the four variables (i.e., *La, Lb, Fab, Fba*) were considered, although in principle, only *La, Lb* and *Fab* should be required. Results of Golem running with three variables are described later. The model induced by Golem with 4 + and 6 − examples considering four variables is as follows:

```
legalstate(la:A/B, lb:C/D, fab:E/B, fba:F/D) :-
    deriv(la:A/B, fba:F/D),
    deriv(lb:C/D, fab:E/B),
    minus(la:A/B, lb:G/D, [ ]),
    minus(la:G/B, lb:C/D, [ ]).
```

Golem constructed this model using 10 examples and 5,408 background facts. The result of the *rlgg* algorithm was a clause with 26 body literals, reduced to 4 literals by generalising against the 6 negative examples.

To show that this model is complete and correct with respect to the standard model, Bratko et al. provided some analytic justification and tested the model on certain behaviour sequences.

From [BMV92] it is easy to identify three problems with Golem. The first is its inability to learn non-determinate concepts. Because of this, Golem cannot learn the definition of the model given in Equation 8.1 since it is non-determinate. That is, the existential variable *Diff* is not uniquely determined in the clause given *La* and *Lb*.

The second problem is the large number of possible facts that needs to be generated for some background definitions, like *add* or *mult*. In particular, *add* was simplified into three more "economical" predicates to reduce the number of tabulated facts. The number of facts increases with the number of variables and landmark values. For this domain, the *add* constraint requires several thousand facts for each triple of variables[1].

The third problem is the use of negative examples to reduce the clause as much as possible. Although this is justified by the need to reduce the length of the clauses produced by the *rlgg* algorithm, in many cases negative examples need to be incrementally provided by the user until a correct definition is found (see section 6.2).

---

[1] Considering four landmark values for each variable and three intervals between the landmarks, each variable can assume seven possible values and three possible directions of change. Thus each variable can be in 21 possible qualitative states. Thus the combination of three variables in a three place predicate gives $3! \times 21^3 = 55,566$.

If Golem is run under the same conditions with only the first three variables in the head of the clause, it produces the following result:

```
legalstate(la:A/B, lb:C/D, fab:E/B) :–
    deriv(lb:C/D, lb:C/D).
```

which is clearly an overgeneral and therefore incorrect model. Before generalising against the 6 negative examples the clause has 18 literals. To obtain better results additional negative examples need to be provided to Golem. No tests were made with additional negative examples, so conclusions cannot be drawn about whether a correct model can be induced.

8.4 Experiments with PAL

In qualitative simulation, each state of a system is given by the values and directions of change of each qualitative variable. In PAL, the qualitative states (examples) are described by two-place atoms: *qvar(Name:Value/Deriv,State)*. For instance, the first qualitative behaviour of Table 8.1 (time = t0) can be described as follows:

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

From Equation 8.1 it can be seen that not all the qualitative variables are required in principle to induce a model. First, we will show how PAL induces a model for the U-tube considering the same variables as Golem. We will then show the results of PAL with fewer variables.

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[2]. As in Golem, the corresponding values for the constraints were ignored. In PAL, background knowledge definitions use the description of the current state of the system to derive only a finite number of relevant facts. A pattern in PAL is defined as follows (see also chapter 5):

[2]I am grateful to Saso Dzeroski for providing me with the code.

$$Head \leftarrow D_1, D_2, \ldots, D_n, F_1, F_2, \ldots, F_m.$$

where the D_is are the "input" predicates used to describe a position (state). 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").

```
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).
```

This representation constrains the generation of facts from the background definitions as only those facts which apply to a particular state can be derived. The same change was made to other background predicates. For instance,

```
minus(F1:M1/D1, F2:M2/D2,State) :-
    qvar(F1:M1/D1,State), % New
    qvar(F2:M2/D2,State), % New
    verify_minus_zeroinf_consistence(M1, M2),
    verify_minus_mag(F1, F2, M1, M2),
    verify_minus_der(D1, D2).
...
```

The rest of the code remained unchanged,

```
verify_add_mag(F1, F2, F3, M1, M2, M3) :-
    norm_mag(F1, M1, 0, A1),
    norm_mag(F2, M2, 0, A2),
    norm_mag(F3, M3, 0, A3),
    lookup_consist_table(A1, A2, A3).

lookup_consist_table(neg, neg, neg ).
lookup_consist_table(neg, zero, neg ).
lookup_consist_table(neg, pos, neg ).
lookup_consist_table(neg, pos, zero).
```

```
lookup_consist_table(neg, pos, pos ).
...
```

The same positive examples that were provided to Golem were manually given to PAL. With them, PAL obtains the following definition[3]:

```
legalstate(la:A/B,lb:C/D,fab:E/B,fba:F/D,S) :–
    qvar(lb:C/D,S),
    qvar(la:A/B,S),
    qvar(fba:F/D,S),
    qvar(fab:E/B,S),
    deriv(lb:C/D,fab:E/B,S),
    deriv(la:A/B,fba:F/D,S),
    deriv(fba:F/D,fab:E/B,S),
    deriv(fab:E/B,fba:F/D,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 $Fab \propto (La - Lb)$ and that $dLa/dt = -Fab$ or that $dLb/dt = Fab$. The first condition is shown by the last two *add* constraints. The second condition is given by the first two *deriv* constraints. 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 modelled system.

PAL produces a longer clause than Golem, because Golem uses the negative examples to reduce the clause as much as possible. Unlike Golem, PAL can use the *add* constraint as background knowledge.

[3]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.

Running PAL with only 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.

Since Golem cannot learn the standard model because it is non-determinate, it is of interest to note that PAL can learn a specialisation (w.r.t. θ-subsumption) of this clause if the qualitative variable *diff* is provided.

In the next section, the models induced by PAL are compared in qualitative terms, together with the model induced by Golem, against the standard model of Equation 8.1 without considering corresponding values.

8.4.1 Comparison of Golem and PAL

The models induced by Golem and PAL were compared against the standard model by evaluating the models on possible states of the U-tube. This provides a measure of the quantitative performance of the outputs of the two algorithms. Some possible states were eliminated on the grounds that they are physically impossible. No values were considered where the qualitative variables could assume an infinite or minus infinite value, and levels *La* and *Lb* were non-negative (e.g., the landmark values for *La* in the tests were as follows: *La* : *0, 0..la0, la0, la0..inf*). With these constraints, the total number of possible states is 2,160. That is, *La* can be in 12 states, *Lb* in 12, and *Fab* in 15 (in the generated states, *Fba* was determined from *Fab*).

Table 8.3 gives the total number of states that are accepted (+) and rejected (−) for each model. *Pal*1 refers to the first model induced by PAL (with 4 variables), *Pal*2 is the model with three variables, and *Golem* is the model induced by Golem and reported in [BMV92]. The standard model accepts 80 examples when no infinite values are considered and the levels

Example Space	Standard	Pal1	Pal2	Golem
2,160	80 + 2,080 −	58 + 2,102 −	58 + 2,102 −	50 + 2,110 −
Legal States	Standard	Pal1	Pal2	Golem
32	32 + 0 −	32 + 0 −	32 + 0 −	24 + 8 −

Table 8.3: Results for the U-tube

on the containers are considered positive. In [BMV92] only 32 are of these states are considered to be possible legal states for the U-tube[4]. This further reduction in the number of examples was obtained by adopting further constraints. All the models were tested on these states.

From Table 8.3 it can be seen that *Pal*1 and *Pal*2 accept the 32 legal states. In fact, they accept the same states for this example space. Golem's model is equivalent to the standard model only in a dynamic sense. From some initial states, the Golem model produces the same behaviour as the standard model. The model induced by Golem fails in a state where all the water is in container B:

La:0/inc, Lb:lb0..inf/dec, Fab:minf..0/inc, Fba:0..inf/dec

or a state where all the water is in container A:

La:la0..inf/dec, Lb:0/inc, Fab: fab0..inf/dec, Fba:minf..fba0/inc

Both of these states are considered to be legal. It should be noted that the examples accepted by *Pal*1 and *Pal*2 (i.e., 58), and Golem (i.e., 50), are subsets of those accepted by the standard model (i.e., 80).

8.4.2 Other related work

Following [BMV92], other authors have worked on this problem. Dzeroski [DB92] experimented with mFoil, a top-down ILP system, and the U-tube. mFoil is similar to Foil except that it uses a different heuristic function,

[4]Again, I am grateful to Saso Deroski for providing me with the examples.

beam search instead of hill climbing, and the stopping criterion is based on statistical significance, similar to the one used in CN2 [CN89] (see [DB92] for mode details).

Using a Laplacean heuristic function, a significance level of (99%), and with a beam size of 20 (its default value is 5), mFoil obtained the following clause from the same four positive examples and 543 randomly generated negative examples:

```
legalstate(La,Lb,Fab,Fba) :–
     minus(Fab,Fba),
     add(Lb,Fab,La),
     m_minus(La,Fba),
     deriv(Fab,Fba).
```

Although this model is not equivalent to the standard model, Dzeroski reports that among the beam, there are 19 other clauses (models) which can distinguish between the given positive and negative examples, two of which are reported correct with respect to the standard model in a dynamic sense. One of them is:

```
legalstate(La,Lb,Fab,Fba) :–
     minus(Fab,Fba),
     add(La,Fba,Lb),
     deriv(Lb,Fab),
     deriv(La,Fba).
```

This model accepts the same examples as $Pal1/2$ (i.e., in an example space of 2,160, it accepts the same 58 states as $Pal1/2$, and rejects the same 2,102 states. It obviously accepts all of the 32 legal states).

Varsek [Var91] applies a genetic algorithm to the same problem. The representation is based on binary trees, where the leaves corresponds to the QSIM constraints. A *crossover* operation corresponds to changing subtrees between trees and a *mutation* operation generates random subtrees on single trees. The fitness value was determined by different parameters that were determined experimentally. The setting of the experiments was as follows: 17 positive and 78 negative examples, an initial population of 200 individuals with an average size of 7.8 constraints, and 35 generations (a total of 3,400 candidate models were created and evaluated). Varsek obtained, among others, the following model:

```
legalstate(La,Lb,Fab,Fba) :–
    add(Lb,Fab,La),
    minus(Fab,Fba),
    deriv(La,Fba),
    deriv(Lb,Fab).
```

which again accepts the same examples as $Pal1/2$.

PAL has advantages over both these approaches. It learns models of equivalent power without the requirement for large numbers of negative examples, and it does not require an adequate tuning of parameters to obtain the target concept.

Perhaps the best results for learning qualitative model have been obtained by Coiera with a system called Genmodel [Coi89]. Genmodel is specifically designed for learning qualitative models and has learned, among others, a model for the U-tube. Genmodel starts with a sequence of example behaviours of a dynamic system, and constructs a list of landmark values and a set of corresponding values for the qualitative variables. It then derives the set of all the constraints that can be applied to the first state. In this sense, Genmodel derives the same constraints from an initial example description as PAL[5]. Genmodel then uses the corresponding values and the rest of the example behaviours to filter out the constraints, removing those which are inapplicable. Genmodel does not use function symbols, but has the advantage of using the full power of QSIM by considering the corresponding values to filter out a larger number of constraints. With 4 variables and the same examples, Genmodel obtains the following model:

```
legalstate(La,Lb,Fab,Fba) :–
    m_minus(Fab,Fba),
    minus(Fab,Fba),
    deriv(La,Fba),
    deriv(Lb,Fab).
```

which accepts the same examples as the standard model (without considering corresponding values). With information about the corresponding values, PAL obtains the same results as Genmodel. Although PAL was not originately designed to this domain, it is interesting to note that it can achieved an equivalent performance to Genmodel.

[5] We compare the results produced by Genmodel reported in [Coi89] for the bath-tube with an experiment with PAL on the same problem and the same conditions.

8.5 Discussion

Although the U-tube looks relatively simple, Bratko et al. [BMV92] and Dzeroski [DB92] report how ILP systems like Foil [Qui90] and Linus [LDG91] 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 $21^3 = 9{,}261$.

We have demonstrated that the learning mechanism employed by PAL generalises 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.

1. PAL is not restricted to learning determinate clauses. Such clauses occur naturally in this and other structural domains (such as Chess). This is demonstrated by the fact that the standard model is non-determinate. PAL has learned a specialisation of this model.

2. The non-deterministic nature of the background knowledge means that a large number of ground facts are required by Golem, which restrict its application in larger domains. The implicit relevance restrictions imposed by PAL means that more background knowledge can be used more economically.

3. The restrictions on *rlgg* used by PAL do not restrict the quality of the solution and produce a more general result than Golem.

Chapter 9

Conclusions and future work

Pattern-based reasoning has been used by several computer systems to guide their reasoning strategies[1]. For Chess, in particular, a pattern-based approach has been used with relative success in simple end-games [Bra77, Bra82, Hub68] and tactically sharp middle games [Ber77, Pit77, Wil79]. However, a substantial programming effort needs to be devoted to the definition and implementation of the *right* patterns for the task.

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. 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 (background knowledge 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 'recognise' 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

[1]In particular, the conditions used in the production rules used in most expert systems can be regarded as particular patterns to match.

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 and in qualitative reasoning, and it is suspected to be useful to other domains where a pattern-based approach can be applied. Providing examples as descriptions of states means that the exact arguments of the target concept are not specified in advance. PAL uses an automatic example generator to guide the learning process and to recognise which arguments are relevant for a concept. The generation of examples is based on the current concept definition and it is organised in a structured example space.

A novel mechanism for labelling 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 generalisation 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.

PAL is applicable to domains where examples are given be a set of components describing particular states of a system, and the background knowledge uses the description of the states to express relations between the components. The main emphasis of the thesis has been to apply PAL to Chess, however, it can be used to other domains where examples can be given as descriptions of states.

9.1 Main contributions of the Thesis

This thesis has made a contribution both to computer Chess, and to Inductive Logic Programming (ILP).

In Chess we have shown that:

- A wide range of symbolic representations of patterns in Chess, expressed in a compact and understandable way, can be learned from examples represented in a simple and natural way. This is outside the scope of both current ILP systems, and other learning mechanisms.
- The concepts learned by PAL are powerful enough to be used in the design of playing strategies. This is particularly relevant to planning systems which rely on a pattern-based approach to produce plans in

reactive environments like Chess, as a substantial programming effort is devoted to the definition of adequate patterns.

The thesis has also made a contribution to ILP. In Chapter 1 (page 5) we defined four issues which should be addressed within any ILP-based learning mechanism for Chess patterns, and for pattern-based concepts more generally. The issues are: the volume of background knowledge, the non-determinacy of pattern-based concepts, incremental concept acquisition, and robustness of learning with respect to example presentation.

These issues are largely resolved in PAL. The specific contributions which resolve the issues are:

- PAL uses an implicit notion of relevance within a pattern-based knowledge representation to produce only a limited number of relevant facts from the current state description. This mechanism worked adequately in both Chess and the qualitative modelling domain despite its simplicity. This mechanism not only reduces the time complexity of the algorithm it also provides a finer tuning of background knowledge than existing algorithms and eliminates the requirement that users identify the relevant ground facts to be derived from a background theory.

- The identification of internal structures within a state, pieces in Chess and qualitative variables in the U-tube model, and the exploitation of this identification via a labelling mechanism considerably reduces the complexity of the basic *lgg* algorithm. As with the notion of relevance this constraint did not restrict the concepts discovered.

- The use of the automatic example generator to ensure that the user did not select the correct sequence of examples demonstrates that, in pattern-based domains at least, the *rlgg* mechanism can be made sufficiently robust for use by domain experts without knowledge of ILP technology.

9.2 Future work

As with most research work, there is the potential for further progress. We will look at three areas. Improvements in the generalisation method, improvements in the example generator, and we will explore the possibilities of learning new concepts from traces of games.

9.2.1 The generalisation method

PAL's generalisation method is based on transforming the *rlgg* of clauses into a more manageable and constrained *lgg* of "saturated" clauses. To avoid a possible combinatorial generation of facts during this transformation process, recursive definitions are not considered and some constraints, discussed in chapter 5, are introduced to limit the size of the clauses. In general, the *lgg* of clauses is limited to single clauses (i.e., it cannot learn disjunctive definitions), cannot include negation of literals, and cannot introduce new terms. PAL only tackles partially the first limitation by considering the negative examples generated so far by PAL, however it still relies on the user to avoid the creation of over-generalised clauses when learning disjunctive definitions. If there is insufficient knowledge in PAL to derive a definition that will distinguish between positive and negative examples, the hypothesis fails. This can be, in some cases, corrected by introducing specialisations and/or allowing some inconsistencies in the hypotheses. Bain's closed-world specialisation method [Bai91], mentioned in chapter 6 for learning a correct definition of *illegality* in the KRK endgame, could be incorporated into PAL for this purpose, either by introducing a new predicate or by introducing a new component.

9.2.2 The example generator

The number of examples presented to the user by the example generator compares very favourably with the size of the example space. However, it should be clear that a very long experimentation process may result when learning concepts involving a large number of pieces. Knowing that a concept does not depend on a particular orientation of the board was used to improve the learning rate by making generalisations between all the symmetric examples of the current positive example before presenting to the user a new example. Instead of specifying the particular symmetric characteristics of the concept at hand, the axes of symmetry can be deduced from the examples and the hypotheses produced by the system.

The example generator can produce examples which are symmetric with respect to the last positive example along a particular axis of symmetry. If this symmetric example is accepted as positive by the user and if it does not produce an over-generalisations, then the particular axis of symmetry used to generate the example, can be stored and used in the future. This process

can continue until all the symmetric properties of the concept are known. With them further generalisations steps can be produced by considering all the symmetric examples. This is particularly relevant for learning disjunctive concepts as each disjunct is in general no longer symmetric along the 3 axes.

A more immediate and general extension is to used the labels to split the example space into independent factors. The labels associated with the hypotheses can distinguish exactly which literals are affected by which arguments. Although not implemented, this information can be used to identify directly which are the independent factors of the concept and reduce even more the example space by considering each factor separately. The main advantage of this approach over Subramanian et al. [SF86] is that the factors are deduced directly from the concept definition, rather than given by the user or deduce after an experimentation process. Similarly the independent factors could be changed dynamically during the learning process. For instance, the positions of two pieces can be considered as an independent factor when they occur together in some literals in the current concept definition. If during the learning process a new definition removes all the literals affecting both positions, then each position can be considered, from then on, independently.

9.2.3 Learning from traces of games

In chapter 7 a correct strategy for playing a simple endgame was designed with patterns learned by PAL. A system on top of PAL could be used to learn complete game-strategies from traces of games. Subsequent positions of traces of games can be compared to see what patterns changed as a consequence of a move. Each move can then be associated with a set of patterns before and after the move, which are considered as relevant to the move. This is already done by PAL when learning dynamic patterns, however, while in PAL several moves are considered to identify patterns that change for learning a single concept, in traces of games the actual moves are given but they can correspond to different concepts. The real problem is to identify which of the moves in different traces of games, correspond to the same general rule in the strategy. This process could be guided in principle by matching the moves and the corresponding patterns associated with them.

Learning from traces of games is a natural way in which to learn to play Chess. This is already done, to a certain extent, by Morph [LS91] (see also chapter 4). In Morph, the descriptions of positions are associated with a

fixed and large set of relations between pieces, defining all the possible direct and indirect threats of the pieces involved, and only patterns associated with these relations can be learned. In PAL, Chess board positions are described in a much simpler way, and a set of background knowledge definitions, used to identify relations between the pieces and places in the board, can be incrementally augmented with new relations learned from a set of example descriptions.

A powerful learning system that could be constructed by combining PAL's learning capabilities to learn appropriate relations, with a learning system that could take these relations with traces of games, to learn playing strategies.

Bibliography

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319 – 342, 1988.

[AS83] D. Angluin and C. H. Smith. Inductive inference: theory and methods. *Computing surveys*, 15(3):237–269, 1983.

[Bai91] M. Bain. Experiments in non-monotonic first-order learning. In S. Muggleton, editor, *Proceedings of the International Workshop of Inductive Logic Programming*, pages 195 – 206, Viana de Castelo, Portugal, 1991.

[BE90] H. J. Berliner and C. Ebeling. Hitech. In T.A. Marsland and J. Schaeffer, editors, *Computers, chess and cognition*, pages 79 – 109. Springer-Verlag, New York, 1990.

[Ben89] S. W. Bennett. Learning approximate plans for use in the real world. In B. Spatz, editor, *Proceedings of the Sixth International Workshop on Wachine Learning*, pages 224–228, San Mateo, CA, 1989. Morgan Kaufmann.

[Ber77] H.J. Berliner. A representation and some mechanisms for a problem-solving chess program. In M.R.B. Clarke, editor, *Advances in Computer Chess 1*, pages 7–29. Edinburgh University Press, Edinburgh, 1977.

[BML89] I. Bratko, I. Mozetic, and N. Lavrac. *Kardio: a study in deep and qualitative knowledge for expert systems.* MIT Press, Cambridge, 1989.

[BMV92] I. Bratko, S. Muggleton, and A. Varsek. Learning qualitative models of dynamic systems. In S. Muggleton, editor, *Inductive*

Logic Programming, pages 437–452. Academic Press, London, 1992.

[Bra75] M. A. Bramer. Representation of knowledge for chess endgames. Technical report, Open University, Faculty of Mathematics, Milton Keynes, 1975.

[Bra77] M. A. Bramer. *Representation of knowledge for chess endgames: Towards a self-improving system.* PhD thesis, Open University, Milton Keynes, 1977.

[Bra82] I. Bratko. Knowledge-based problem-solving in AL3. In Hayes J E Michie D Pao Y H, editor, *Machine intelligence 10*, pages 73–100. Horwood, 1982.

[Bun88] W. Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial intelligence*, 36(2):149–176, 1988.

[Cam66] D. T. Campbell. *Pattern Matching as an Essential in Distal Knowing.* Holt, Rinehart and Winston, New York, 1966.

[Cam88] M. S. Campbell. Chunking as an abstraction mechanism. Technical Report *CMU-CS-88-116*, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, 1988.

[CG87] J. G. Carbonell and Y. Gil. Learning by experimentation. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 256–266, Los Altos, CA, 1987. Morgan Kaufmann.

[Cha77] N. Charness. Human chess skill. In P.W. Frey, editor, *Chess skill in man and machine*, pages 34–53. Springer-Verlag, 1977.

[CKB87] B. Cestnik, I. Kononenko, and I. Bratko. ASSISTANT 86: a knowledge-elicitation tool for sophisticated users. In I. Bratko and N. Lavrac, editors, *Proceedings of the 2nd European Working Session on Learning*, pages 31–45, Wilmslow, 1987. Sigma Press.

[CN89] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine learning*, 3 (4):261–283, 1989.

[Coi89] E. Coiera. Generating qualitative models from example behaviors. Technical Report *DCS Report No. 8901*, School of Electrical Engineering and Computer Science, University of New South Wales, Sydney, Australia, 1989.

[CS88] W. G. Chase and H. A. Simon. The mind's eye in Chess. In A. Collins and E.E. Smith, editors, *Readings of cognitive science: a perspective from psychology and artificial intelligence*, pages 461–494. Morgan Kaufmann, San Mateo, CA, 1988.

[DB83] T. G. Dietterich and B. G. Buchanan. The role of experimentation in theory formation. In *Proceedings of the International Machine Learning Workshop*, pages 147–155, Urbana, IL, 1983. University of Illinois, Department of Computer Science.

[DB92] S. Dzeroski and I. Bratko. Handling noise in inductive logic programming. Technical Report Proceedings ILP92: International Workshop on Inductive Logic Progamming, *ICOT TM-1181*, Institute for New Generation Computer Technology, Tokyo, Japan, 1992.

[dG65] A. de Groot. *Thought and Choice in Chess*. Mouton, The Hague, 1965.

[dG86] A. D. de Groot. Intuition in chess. *ICCA Journal*, 9(2):67–75, 1986.

[dJM86] G. de Jong and R. Mooney. Explanation-based learning: an alternative view. *Machine Learning*, 1((2)):145–176, 1986.

[dRB88] L. de Raedt and M. Bruynooghe. On interactive concept-learning and assimilation. In D. Sleeman and J. Richmond, editors, *Proceedings of the Third European Working Session on Learning*, pages 167–176, London, 1988. Pitman.

[dRB90] L. de Raedt and M. Bruynoghe. Indirect relevance and bias in inductive concept-learning. *Knowledge Acquisition*, 2(4):365–390, 1990.

[dRB92] L. de Raedt and M. Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8(2):107 – 150, 1992.

[dRBM91] L. de Raedt, M. Bruynooghe, and B. Martens. Integrity constraints and interactive concept-learning. In L.A. Birnbaum and G.C. Collins, editors, *Proceedings ML91: Eighth International Conference on Machine Learning*, pages 394 – 398, San Mateo, CA., 1991. Morgan Kaufmann.

[FD89] N. S. Flann and T. G. Dietterich. A study of explanation-based methods for inductive learning. *Machine learning*, 4(2):187–226, 1989.

[Fen90] C. Feng. *Learning by Experimentation*. PhD thesis, The Turing Institute - University of Strathclyde, 1990.

[FHN72] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Some new directions in robot problem solving. In B. Melzer and D. Michie, editors, *Machine intelligence 7*, pages 405–430. Edinburgh University Press, Edinburgh, 1972.

[FM83] E. Feigenbaum and P. McCorduck. *The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge to the World*. Addison-Wesley, Reading, MA, 1983.

[GGM91] D. Gadwal, J.E. Greer, and G.I. McCalla. UMRAO: a chess endgame tutor. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th. International Joint Conference on Artificial Intelligence*, pages 1081 – 1086, San Mateo, CA, 1991. Morgan Kaufmann.

[Gol67] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[HACN90] F. H. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. A grandmaster chess machine. *Scientific american*, 263(4):18–24, 1990.

[HR85] B. Hayes-Roth. A blackboard architecture for control. *Artificial intelligence*, 26(3):251–321, 1985.

[Hub68] B. J. Huberman. A program to play chess end games. Technical Report *CS-106*, Stanford University, Computer Science Department, Stanford, CA, 1968.

[Iba89] G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine learning*, 3(4):285–317, 1989.

[Kor85] R. E. Korf. Macro-operators: a weak method for learning. *Artificial intelligence*, 26(1):35–77, 1985.

[Kui86] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29 (3):289–338, 1986.

[LBS83] P. Langley, G.L. Bradshae, and H.A. Simon. Rediscovering chemistry with the Bacon system. In J.G. Carbonell R.S. Michalski and T.M. Mitchell, editors, *Machine learning*, pages 307 – 330. Tioga, Palo Alto, CA., 1983.

[LDG91] N. Lavrac, S. Dzeroski, and M. Grobelnik. Learning nonrecursive definitions of relations with linus. In Y. Kodratoff, editor, *Proceedings of the European Working Session on Learning*, pages 265–281, Berlin, 1991. Springer-Verlag.

[Len76] D. B. Lenat. AM: an artificial intelligence approach to discovery in mathematics as heuristic search. Technical Report *AIM-286 ; STAN-CS-76-570*, Stanford University, Artificial Intelligence Laboratory, Stanford, CA, 1976.

[LHS+91] R. Levinson, F.H. Hsu, J. Schaeffer, T.A. Marsland, and D.E. Wilkins. Panel: the role of chess in artificial intelligence research. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th. International Joint Conference on Artificial Intelligence*, pages 547 – 552, San Mateo, CA, 1991. Morgan Kaufmann.

[Lin91] C. Ling. Logic Program Synthesis from Good Examples. In S. Muggleton, editor, *Proceedings of the International Workshop of Inductive Logic Programming*, pages 41 – 57, Viana de Castelo, Portugal, 1991.

[LNR87] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: an architecture for general intelligence. *Artificial intelligence*, 33(1):1–64, 1987.

[Lor73] K. Lorenz. *Behind the Mirror*. Harcourt Brace Jovanovich, New York, 1973.

[LS91] R. Levinson and R. Snyder. Adaptive Pattern-Oriented Chess. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 601–606, Boston, 1991. AAAI Press - The MIT Press.

[MB88] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In J. Laird, editor, *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352, San Mateo, CA, 1988. Morgan Kaufmann.

[MBHMM89] S. Muggleton, M. Bain, J. Hayes-Michie, and D. Michie. An experimental comparison of human and machine learning formalisms. In B. Spatz, editor, *Proceedings of the Sixth International Workshop on Machine Learning*, pages 113–118, San Mateo, CA, 1989. Morgan Kaufmann.

[MF90] S. Muggleton and C. Feng. Efficient Induction of Logic Programs. In S. Arikaxa, S. Goto, S. Ohsuya, and T. Yokomari, editors, *Proceedings of the Conference of Algorithmic Learning Theory*, pages 368–381, Tokyo, Japan, 1990. Ohmsha.

[Mic76] D. Michie. AL1: a package for generating strategies from tables. *SIGART Newsletter*, (59):12–14, 1976.

[Mic77] D. Michie. King and rook against king: historical background and a problem on the infinite board. In M.R.B. Clarke, editor, *Advances on Computer Chess 1*, pages 30–59. Edinburgh University Press, Edinburgh, 1977.

[Mic82] D. Michie. Computer chess and the humanisation of technology. *Nature*, (299):391–394, 1982.

[Mic89] D. Michie. Brute force in chess and science. *ICCA journal*, 12(3):127–143, 1989.

[Min84] S. Minton. Constraint-based generalization: learning game-playing plans from single examples. In *Proceedings of the National Conference on Artificial Intelligence*, pages 251–254, Menlo Park, CA, 1984. Kaufmann.

[Mit82] T. M. Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[MKKC86] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[Moo90] R. J. Mooney. Learning plan schemata from observation: explanation-based learning for plan recognition. *Cognitive science*, 14(4):483–509, 1990.

[Mor89] K. Morik. Sloppy modeling. In K. Morik, editor, *Knowledge Representation and Organization in Machine Learning*, pages 107–134. Springer-Verlag, Berlin, 1989.

[Mor90] E. Morales. Some experiments with macro-operators in the 8-puzzle. Technical Report *TIRM-90-042*, The Turing Institute, Glasgow, 1990.

[Mor91a] E. Morales. Learning chess patterns. In S. Muggleton, editor, *Proceedings of the International Workshop of Inductive Logic Programming*, pages 291–307, Viana de Castelo, Portugal, 1991.

[Mor91b] E. Morales. Learning features by experimentation in chess. In Y. Kodratoff, editor, *Proceedings of the European Working Session on Learning*, pages 494–551, Berlin, 1991. Springer-Verlag.

[MSB91] S. Muggleton, A. Srinivasan, and M. Bain. MDL codes for non-monotonic learning. Technical Report *TIRM-91-049*, The Turing Institue, Glasgow, U.K., 1991.

[MUB83] T.M. Mitchell, P.E. Utgoff, and R. Banerji. Learning by experimentation: acquiring and refining problem-solving heuristics. In J.G. Carbonell R.S. Michalski and T.M. Mitchell, editors, *Machine learning*, pages 163 – 1930. Tioga, Palo Alto, CA., 1983.

[Mug87] S. Muggleton. Duce, an oracle based approach to constructive induction. In J. McDermott, editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 287–292, Los Altos, CA, 1987. Morgan Kaufmann.

[Mug91a] S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8:295–318, 1991.

[Mug91b] S. Muggleton. *Proceedings of the International Workshop of Inductive Logic Programming*. Viana de Castelo, Portugal, 1991.

[New88] M. M. Newborn. Recent progress in computer chess. In D.N.L. Levy, editor, *Computer Games I*, pages 154–205. Springer-Verlag, New York, 1988.

[Nib88] T. Niblett. A study of generalisation in logic programs. In D. Sleeman and J. Richmond, editors, *Proceedings of the Third European Working Session on Learning*, pages 131–138, London, 1988. Pitman.

[Nie91] J. Nievergelt. Information content of chess positions: implications for game-specific knowledge of chess players. In J.E. Hayes, D. Michie, and E. Tyugu, editors, *Machine intelligence 12: towards an automated logic of human thought*, pages 283–289. Clarendon Press, Oxford, 1991.

[Pea88] D. A. Pearce. The induction of fault diagnosis systems from qualitative models. In *Proceedings AAAI-88: Seventh National Conference on Artificial Intelligence*, pages 353 – 357, San Mateo, CA., 1988. Morgan Kaufmann.

[Pit77] J. Pitrat. A chess combination program which uses plans. *Artificial Intelligence*, 8:275–321, 1977.

[PK86] B. W. Porter and D. F. Kibler. Experimental goal regression: A method for learning problem-solving heuristics. *Machine Learning*, (1):249–286, 1986.

[PK90] M. Pazzani and D. Kibler. The Utility of Knowledge in Inductive Learning. Technical Report *90-18*, University of California, Department of Information and Computing Science, Irvine, CA, 1990.

[Plo69] G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh, 1969.

[Plo71a] G. D. Plotkin. A further note of inductive generalization. In B. Meltzer and D. Michie, editors, *Machine intelligence 6*, pages 101–124. Edinburgh University Press, Edinburgh, 1971.

[Plo71b] G. D. Plotkin. *Automatic methods of inductive inference*. PhD thesis, Edinburgh, 1971.

[Pop59] K. R. Popper. *The Logic of Scientific Discovery*. Basic Books, New York, 1959.

[Qui83] J. R. Quinlan. Learning efficient classification procedures and their application to chess end games. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, pages 463–482. Tioga, Palo Alto, CA, 1983.

[Qui87] J. R. Quinlan. Generating production rules from decision trees (Vol 1). In J. McDermott, editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 304–307, Los Altos, CA, 1987. Morgan Kaufmann.

[Qui90] J. R. Quinlan. Learning logical definitions from relations. *Machine learning*, 5(3):239–266, 1990.

[RD89] R. A. Ruff and T. G. Dietterich. What good are experiments? In B. Spatz, editor, *Proceedings of the Sixth International Workshop on Machine Learning*, pages 109–112, San Mateo, CA, 1989. Morgan Kaufmann.

[RG88] S.J. Russell and B.N. Grosof. A sketch of autonomous learning using declarative bias. In P. Brazdil, editor, *Proceedings of the International Workshop on Machine Learning, Metareasoining and Logic*, pages 147 – 166, Sesimbra, Portugal, 1988.

[Rou91] C. Rouveurol. ITOU: Induction of first order theories. In S. Muggleton, editor, *Proceedings of the International Workshop of Inductive Logic Programming*, pages 127–157, Viana de Castelo, Portugal, 1991.

[SB86] C. Sammut and R. B. Banerji. Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, and R.M. Mitchell, editors, *Machine learning: an artificial intelligence approach [Volume 2]*, pages 167–191. Kaufmann, 1986.

[SF86] D. Subramanian and J. Feigenbaum. Factorization in experiment generation. In *Proceedings AAAI-86: Fifth National Conference on Artificial Intelligence*, pages 518 – 522, 1986.

[SG73] H. A. Simon and K. Gilmartin. A Simulation of Memory for Chess Positions. *Cognitive Psychology*, 5:29–46, 1973.

[Sha81] E. Y. Shapiro. Inductive inference of theories from facts. Technical Report *Research Report 192*, Yale University, Department of Computer Science, New Haven, CT, 1981.

[Sha87] A. D. Shapiro. *Structured induction in expert systems*. Turing Institute Press in association with Addison-Wesley, Wokingham, 1987.

[Sha88] C. E. Shannon. A chess-playing machine. In D.N.L. Levy, editor, *Computer Games I*, pages 81–88. Springer-Verlag, New York, 1988.

[Sha89] J. W. Shavlik. An empirical analysis of EBL approaches for learning plan schemata. In B. Spatz, editor, *Proceedings of the Sixth International Workshop on Machine Learning*, pages 183–187, San Mateo, CA, 1989. Morgan Kaufmann.

[Sim81] H. A. Simon. *The sciences of the artificial*. MIT Press, Cambridge, MA, 1981.

[SN82] A. Shapiro and T. Niblett. Automatic induction of classification rules for a chess endgame. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 73–91. Pergamon, Oxford, 1982.

[Tad89] P. Tadepalli. Planning in games using approximately learned macros. In B. Spatz, editor, *Proceedings of the Sixth International Workshop on Machine Learning*, pages 221–223, San Mateo, CA, 1989. Morgan Kaufmann.

[Thi89] S. Thieme. The acquisition of model-knowledge for a model-driven machine learning approach. In K. Morik, editor, *Knowledge Representation and Organization in Machine Learning*, pages 177–191. Springer-Verlag, Berlin, 1989.

[Tur53] A. Turing. Digital computers applied to chess. In B.V. Bowden, editor, *Faster Than Thought: A Symposium on Digital Computing Machines*, pages 286–310. Pitman, London, 1953.

[Tur63] A. Turing. Computing machinery and intelligence. In E.A. Feigenbaum and J. Feldman, editors, *Computers and thought*, pages 11–35. Krieger, 1963.

[Val84] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[Var91] A. Varsek. Qualitative model evolution. In J. Mylopoulos and R. Reiter, editors, *Proceedings IJCAI-91: Twelfth International Joint Conference on Artificial Intelligence*, pages 1311 – 1316, San Mateo, CA., 1991. Morgan Kaufman.

[vT91] A. van Tiggelen. Neural networks as a guide to optimization: the chess middle game explored. *ICCA Journal*, 14(3):115 – 118, 1991.

[vTH91] A. van Tiggelen and H.J. Herik. ALEXS: An optimization approach for the endgame KNNKP(h). In A. Cohn, editor, *Advances in Computer Chess 6*, pages 161 – 177. Ellis Horwood Ltd., Chichester, 1991.

[Wal91] S. Walczak. Predicting actions from induction on past performance. In L.A. Birnbaum and G.C. Collins, editors, *Proceedings of the Eighth International Workshop on Machine Learning*, pages 275 – 279, San Mateo, CA., 1991. Morgan Kaufmann.

[Wil79] D. E. Wilkins. Using patterns and plans to solve problems and control search. Technical Report (AIM-329 ; STAN-CS-79-747), Stanford University, Artificial Intelligence Laboratory, Stanford, CA, 1979.

[Win85] P. H. Winston. Learning structural descriptions from examples. In R.J. Brachman and H.J. Levesque, editors, *Readings in Knowledge Representation*, pages 141–168. Kaufmann, Los Altos, CA, 1985.

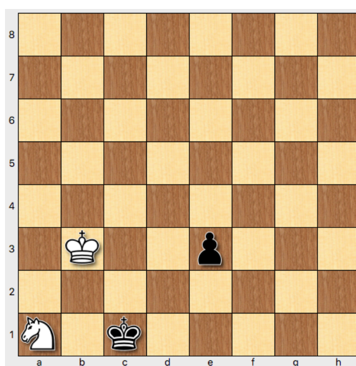
[Wro89] S. Wrobel. Demand-driven concept formation. In K. Morik, editor, *Knowledge Representation and Organization in Machine Learning*, pages 289–319. Springer-Verlag, Berlin, 1989.

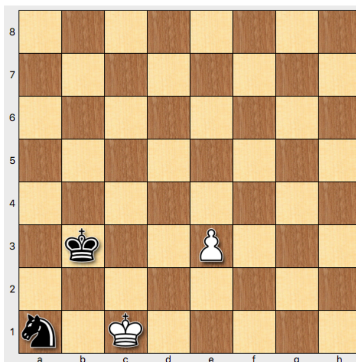
[Zui74] C. Zuidema. Chess, how to program the exceptions? Technical Report *Afdeling informatica, IW21*, Mathematisch Centrum, Amsterdam, 1974.

Appendix A

Learning Can-threat

This appendix shows a trace of the generalisation process involved when learning the concept of *can_threat*, that is, if piece can *threat* another piece after making a legal move. PAL learned this concept after learning the concept of *threat*. The user's input is given in boldface and italics are used to include comments. For presentation purposes *square*(X,Y) is denoted by (X,Y). In the user's initial example the white Knight *can threat* the black Pawn by moving to *square*(3,2). In all clauses ¬ denotes negation. Examples are illustrated with figures.
|?− `pal.`



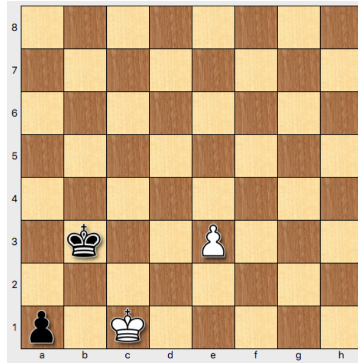


Positive, Negative or Stop (p/n/s)? `p.` (*Initially PAL changes sides*)

tmp(A,king,(2,3),A,knight,(1,1),B,king,(3,1),B,pawn,(5,3),(1,2),(1,3),
 (1,4),(2,1),(2,4),(3,2),(3,3),(3,4),(4,1),(4,2),(5,C),Pos1) ←
 contents(A,knight,(1,1),Pos1), contents(A,king,(2,3),Pos1),
 contents(B,pawn,(5,3),Pos1), contents(B,king,(3,1),Pos1),
 other_side(A,B),
 legal_move(A,knight,(1,1),(3,2),Pos1),
 legal_move(A,king,(2,3),(3,4),Pos1),
 legal_move(A,king,(2,3),(3,3),Pos1),
 legal_move(A,king,(2,3),(2,4),Pos1),
 legal_move(A,king,(2,3),(1,4),Pos1),
 legal_move(A,king,(2,3),(1,3),Pos1),
 legal_move(A,king,(2,3),(1,2),Pos1),
 legal_move(B,pawn,(5,3),(5,C),Pos1),
 legal_move(B,king,(3,1),(4,2),Pos1),

legal_move(B,king,(3,1),(4,1),Pos1),
legal_move(B,king,(3,1),(2,1),Pos1),
¬ threat(A,knight,(1,1),B,pawn,(5,3),Pos1),
¬ threat(B,king,(3,1),A,knight,(1,1),Pos1),
make_move(A,knight,(1,1),(3,2),Pos1,Pos2),
threat(A,knight,(3,2),B,pawn,(5,3),Pos2),
make_move(B,king,(3,1),(2,1),Pos1,Pos3),
threat(B,king,(2,1),A,knight,(1,1),Pos3).

PAL recognises the intended one-ply threat of the Knight, however, it also recognises a possible one-ply threat of the opponent's King
Accept definition Yes/No/Save-Stop (y/n/s):? `y.`

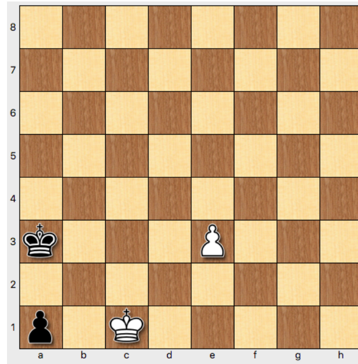


Positive, Negative or Stop (p/n/s)? `p.` (*Changed Knight to Bishop*)

tmp(A,king,(2,3),A,B,(1,1),C,king,(3,1),C,pawn,(5,3),(E,F),(1,2),(1,3),
(1,4),(2,1),(2,4),(3,3),(3,4),(4,1),(4,2),(5,D),Pos1) ←
contents(C,king,(3,1),Pos1), contents(C,pawn,(5,3),Pos1),
contents(A,king,(2,3),Pos1), contents(A,B,(1,1),Pos1),
other_side(A,C),
legal_move(C,king,(3,1),(2,1),Pos1),
legal_move(C,king,(3,1),(4,1),Pos1),
legal_move(C,king,(3,1),(4,2),Pos1),
legal_move(C,pawn,(5,3),(5,D),Pos1),
legal_move(A,king,(2,3),(1,2),Pos1),
legal_move(A,king,(2,3),(1,3),Pos1),
legal_move(A,king,(2,3),(1,4),Pos1),
legal_move(A,king,(2,3),(2,4),Pos1),

legal_move(A,king,(2,3),(3,3),Pos1),
legal_move(A,king,(2,3),(3,4),Pos1),
legal_move(A,B,(1,1),(E,F),Pos1),
¬ threat(C,king,(3,1),A,B,(1,1),Pos1),
¬ threat(A,B,(1,1),C,pawn,(5,3),Pos1),
make_move(C,king,(3,1),(2,1),Pos1,Pos2),
threat(C,king,(2,1),A,B,(1,1),Pos2),
make_move(A,B,(1,1),(E,F),Pos1,Pos3),
threat(A,B,(E,F),C,pawn,(5,3),Pos3).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`

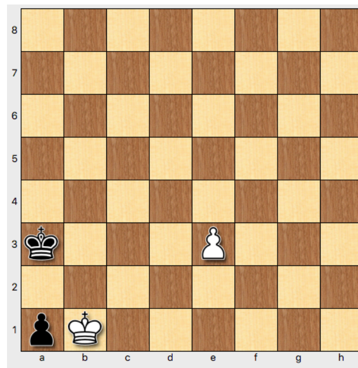


Positive, Negative or Stop (p/n/s)? `p.` (*Changed black King's file*)

tmp(A,king,(B,3),A,C,(1,1),D,king,(3,1),D,pawn,(5,3),(B,4),(E,F),(1,2),
(1,4),(2,1),(2,4),(4,B),(4,1),(4,2),(5,G),Pos1) ←
contents(A,C,(1,1),Pos1), contents(A,king,(B,3),Pos1),
contents(D,pawn,(5,3),Pos1), contents(D,king,(3,1),Pos1),
other_side(A,D),
legal_move(A,king,(B,3),(2,4),Pos1),
legal_move(A,king,(B,3),(B,4),Pos1),
legal_move(A,king,(B,3),(1,4),Pos1),
legal_move(A,king,(B,3),(1,2),Pos1),
legal_move(A,C,(1,1),(E,F),Pos1),
legal_move(D,king,(3,1),(4,2),Pos1),
legal_move(D,king,(3,1),(4,B),Pos1),
legal_move(D,king,(3,1),(4,1),Pos1),
legal_move(D,king,(3,1),(2,1),Pos1),

legal_move(D,pawn,(5,3),(5,G),Pos1),
¬ threat(A,C,(1,1),D,pawn,(5,3),Pos1),
¬ threat(D,king,(3,1),A,C,(1,1),Pos1),
make_move(A,C,(1,1),(E,F),Pos1,Pos2),
threat(A,C,(E,F),D,pawn,(5,3),Pos2),
make_move(D,king,(3,1),(2,1),Pos1,Pos3),
threat(D,king,(2,1),A,C,(1,1),Pos3).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`

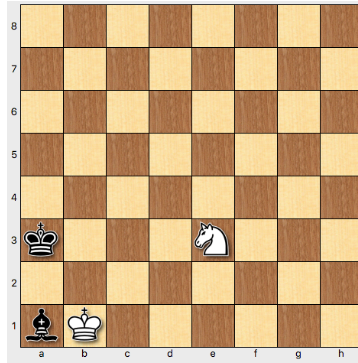


Positive, Negative or Stop (p/n/s)? `p.` (*Changed white King's file*)

tmp(A,king,(B,3),A,C,(1,1),D,king,(E,1),D,pawn,(5,3),(B,4),(F,G),(1,4),
(2,4),(5,H),Pos1) ←
contents(D,king,(E,1),Pos1), contents(D,pawn,(5,3),Pos1),
contents(A,king,(B,3),Pos1), contents(A,C,(1,1),Pos1),
other_side(A,D),
legal_move(A,king,(B,3),(1,4),Pos1),
legal_move(A,king,(B,3),(B,4),Pos1),
legal_move(A,king,(B,3),(2,4),Pos1),
legal_move(A,C,(1,1),(F,G),Pos1),
legal_move(D,pawn,(5,3),(5,H),Pos1),
¬ threat(A,C,(1,1),D,pawn,(5,3),Pos1),
make_move(A,C,(1,1),(F,G),Pos1,Pos2),
threat(A,C,(F,G),D,pawn,(5,3),Pos2).

With this example, only one possible threat (between the black Bishop and the white Pawn) is possible after a legal move.

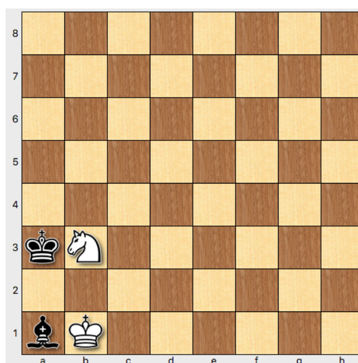
Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



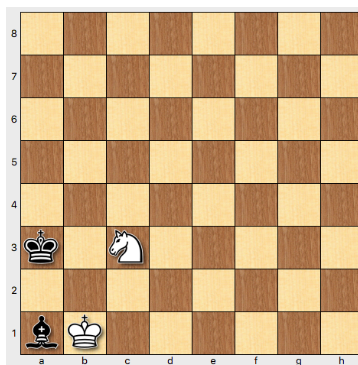
Positive, Negative or Stop (p/n/s)? `p.` (*Changed Pawn to Knight*)

tmp(A,king,(B,3),A,C,(1,1),D,king,(E,1),D,F,(5,3),(B,4),(G,H),(1,4),
(2,4),Pos1) ←
contents(A,C,(1,1),Pos1), contents(A,king,(B,3),Pos1),
contents(D,F,(5,3),Pos1), contents(D,king,(E,1),Pos1),
other_side(A,D),
legal_move(A,king,(B,3),(2,4),Pos1),
legal_move(A,king,(B,3),(B,4),Pos1),
legal_move(A,king,(B,3),(1,4),Pos1),
legal_move(A,C,(1,1),(G,H),Pos1),
¬ threat(A,C,(1,1),D,F,(5,3),Pos1),
make_move(A,C,(1,1),(G,H),Pos1,Pos2),
threat(A,C,(G,H),D,F,(5,3),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`

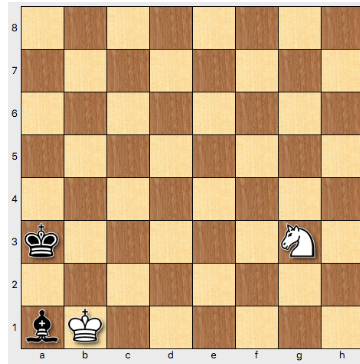


Positive, Negative or Stop (p/n/s)? `n.` (*Changed white Knight's file*)
There are no possible threats after a piece movement. PAL now looks for a place where some of the failed literals will succeed.



Positive, Negative or Stop (p/n/s)? `p.` (*Changed white Knight's file*)
Since the black King can threat the white Knight after a legal move, this is taken as a positive example by the user. However, it is not the threat that PAL has in its current hypothesis (i.e. between the Bishop and Knight) and produces an overgeneralisation which covers the previous negative example. PAL rejects this generalisation and the example is stored. All the saved examples are checked against the final definition.
Too General, it will be ignored

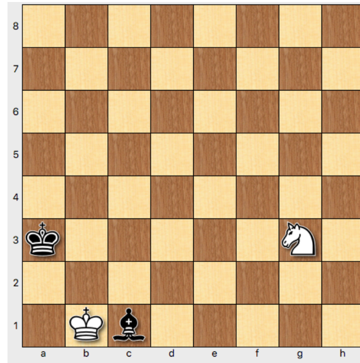
PAL tries to find another position that will succeed on other failed literals



Positive, Negative or Stop (p/n/s)? `p.` (*Changed white Knight's file*)

tmp(A,king,(B,3),A,C,(1,1),D,king,(E,1),D,F,(G,3),(B,4),(H,I),(1,4),
(2,4),Pos1) ←
contents(D,king,(E,1),Pos1), contents(D,F,(G,3),Pos1),
contents(A,king,(B,3),Pos1), contents(A,C,(1,1),Pos1),
other_side(A,D),
legal_move(A,king,(B,3),(1,4),Pos1),
legal_move(A,king,(B,3),(B,4),Pos1),
legal_move(A,king,(B,3),(2,4),Pos1),
legal_move(A,C,(1,1),(H,I),Pos1),
¬ threat(A,C,(1,1),D,F,(G,3),Pos1),
make_move(A,C,(1,1),(H,I),Pos1,Pos2),
threat(A,C,(H,I),D,F,(G,3),Pos2).

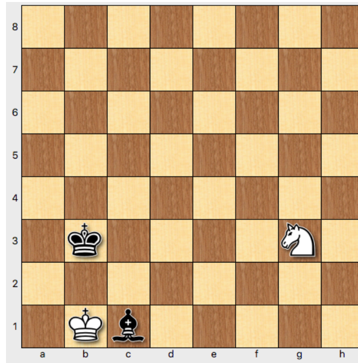
Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



Positive, Negative or Stop (p/n/s)? `p.` (*Changed black Bishop's file*)

tmp(A,king,(B,3),A,C,(D,1),E,king,(F,1),E,G,(H,3),(B,4),(I,J),(1,4),
(2,4),Pos1) ←
contents(A,C,(D,1),Pos1), contents(A,king,(B,3),Pos1),
contents(E,G,(H,3),Pos1), contents(E,king,(F,1),Pos1),
other_side(A,E),
legal_move(A,king,(B,3),(2,4),Pos1),
legal_move(A,king,(B,3),(B,4),Pos1),
legal_move(A,king,(B,3),(1,4),Pos1),
legal_move(A,C,(D,1),(I,J),Pos1),
¬ threat(A,C,(D,1),E,G,(H,3),Pos1),
make_move(A,C,(D,1),(I,J),Pos1,Pos2),
threat(A,C,(I,J),E,G,(H,3),Pos2).

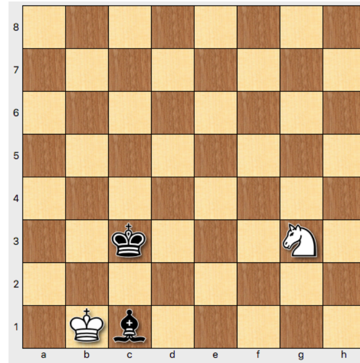
Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



Positive, Negative or Stop (p/n/s)? `p.` (*Changed black King's file*)

tmp(A,king,(B,3),A,C,(D,1),E,king,(F,1),E,G,(H,3),(B,4),(I,J),(1,4),
(2,4),Pos1) ←
contents(E,king,(F,1),Pos1), contents(E,G,(H,3),Pos1),
contents(A,king,(B,3),Pos1), contents(A,C,(D,1),Pos1),
other_side(A,E),
legal_move(A,king,(B,3),(1,4),Pos1),
legal_move(A,king,(B,3),(B,4),Pos1),
legal_move(A,king,(B,3),(2,4),Pos1),
legal_move(A,C,(D,1),(I,J),Pos1),
¬ threat(A,C,(D,1),E,G,(H,3),Pos1),
make_move(A,C,(D,1),(I,J),Pos1,Pos2),
threat(A,C,(I,J),E,G,(H,3),Pos2).

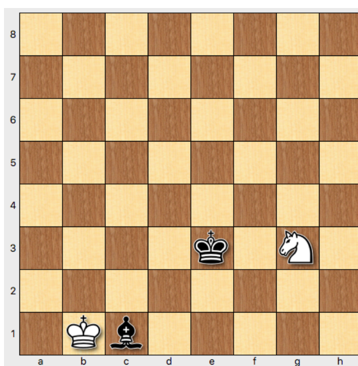
Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



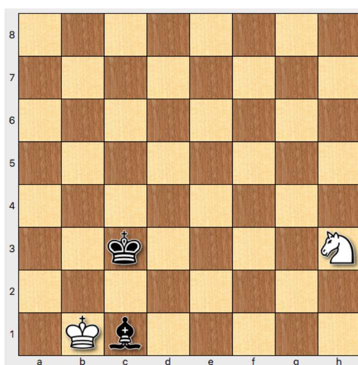
Positive, Negative or Stop (p/n/s)? `p.` (*Changed black King's file*)

tmp(A,king,(B,3),A,C,(D,1),E,king,(F,1),E,G,(H,3),(B,4),(I,J),(2,4),Pos1) ←
contents(A,C,(D,1),Pos1), contents(A,king,(B,3),Pos1),
contents(E,G,(H,3),Pos1), contents(E,king,(F,1),Pos1),
other_side(A,E),
legal_move(A,king,(B,3),(2,4),Pos1),
legal_move(A,king,(B,3),(B,4),Pos1),
legal_move(A,C,(D,1),(I,J),Pos1),
¬ threat(A,C,(D,1),E,G,(H,3),Pos1),
make_move(A,C,(D,1),(I,J),Pos1,Pos2),
threat(A,C,(I,J),E,G,(H,3),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`

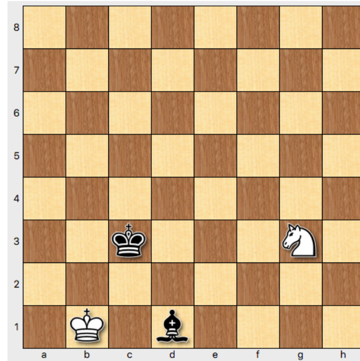


Positive, Negative or Stop (p/n/s)? `p.` (*Changed black King's file*)
The black King can threat the white Knight after a move and the white Knight can also threat the black Bishop after a move. The position is classified as positive by the user, but produces an overgeneralisation since it does not have the 1-ply threat between the Bishop and the Knight.
Too General, it will be ignored



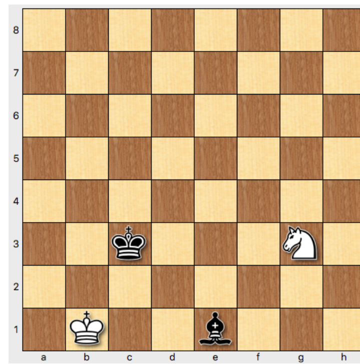
Positive, Negative or Stop (p/n/s)? `n.` (*Changed white Knight's file*)
We want positions without an existing threat, i.e., which can be created after

a move.

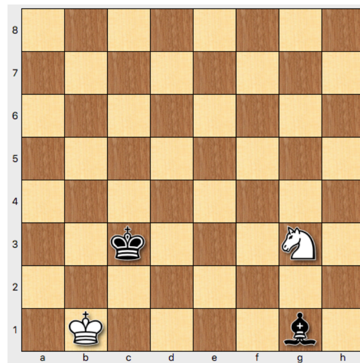


Positive, Negative or Stop (p/n/s)? `p.` (*Changed black Bishop's file*)
The white King can threat the black Bishop after a move. It is a positive example, but creates an overgeneralisation.

Too General, it will be ignored



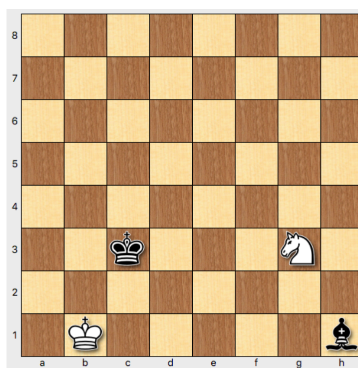
Positive, Negative or Stop (p/n/s)? `n.` (*Changed black Bishop's file*)



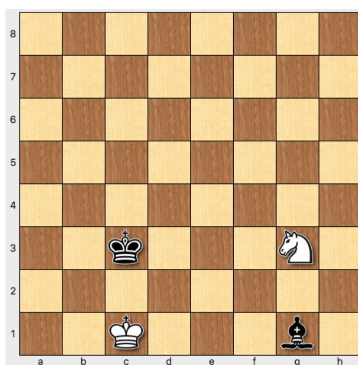
Positive, Negative or Stop (p/n/s)? `p.` (*Changed black Bishop's file*)

tmp(A,king,(B,3),A,C,(D,1),E,king,(F,1),E,G,(H,3),(B,4),(I,J),(2,4),Pos1) ←
contents(E,king,(F,1),Pos1), contents(E,G,(H,3),Pos1),
contents(A,king,(B,3),Pos1), contents(A,C,(D,1),Pos1),
other_side(A,E),
legal_move(A,king,(B,3),(B,4),Pos1),
legal_move(A,king,(B,3),(2,4),Pos1),
legal_move(A,C,(D,1),(I,J),Pos1),
¬ threat(A,C,(D,1),E,G,(H,3),Pos1),
make_move(A,C,(D,1),(I,J),Pos1,Pos2),
threat(A,C,(I,J),E,G,(H,3),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



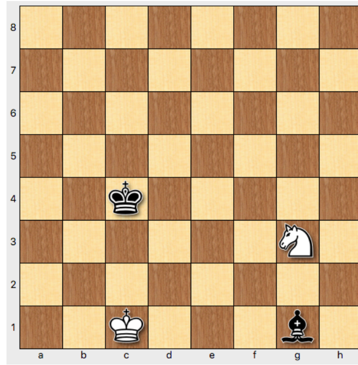
Positive, Negative or Stop (p/n/s)? `n.` (*Changed black Bishop's file*)



Positive, Negative or Stop (p/n/s)? `p.` (*Changed white King's file*)

tmp(A,king,(B,3),A,C,(D,1),E,king,(F,1),E,G,(H,3),(B,4),(I,J),(2,4),Pos1) ←
  contents(A,C,(D,1),Pos1), contents(A,king,(B,3),Pos1),
  contents(E,G,(H,3),Pos1), contents(E,king,(F,1),Pos1),
  other_side(A,E),
  legal_move(A,king,(B,3),(2,4),Pos1),
  legal_move(A,king,(B,3),(B,4),Pos1),
  legal_move(A,C,(D,1),(I,J),Pos1),
  ¬ threat(A,C,(D,1),E,G,(H,3),Pos1),
  make_move(A,C,(D,1),(I,J),Pos1,Pos2),
  threat(A,C,(I,J),E,G,(H,3),Pos2).

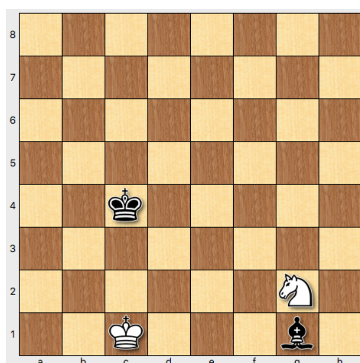
Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



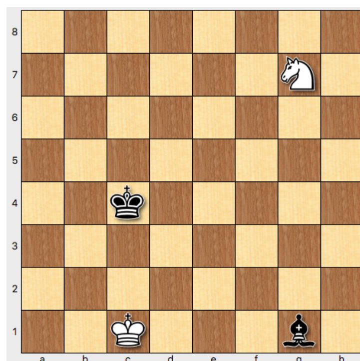
Positive, Negative or Stop (p/n/s)? `p.` (*Changed black King's rank*)

tmp(A,king,(B,C),A,D,(E,1),F,king,(G,1),F,H,(I,3),(J,K),(2,4),Pos1) ←
contents(F,king,(G,1),Pos1), contents(F,H,(I,3),Pos1),
contents(A,king,(B,C),Pos1), contents(A,D,(E,1),Pos1),
other_side(A,F),
legal_move(A,king,(B,C),(2,4),Pos1),
legal_move(A,D,(E,1),(J,K),Pos1),
¬ threat(A,D,(E,1),F,H,(I,3),Pos1),
make_move(A,D,(E,1),(J,K),Pos1,Pos2),
threat(A,D,(J,K),F,H,(I,3),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



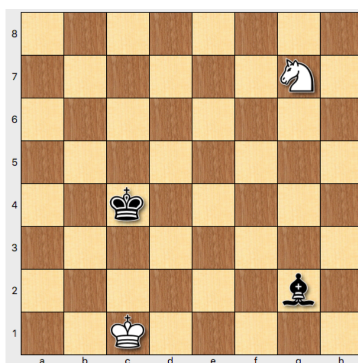
Positive, Negative or Stop (p/n/s)? `n.` (*Changed white Knight's rank*)



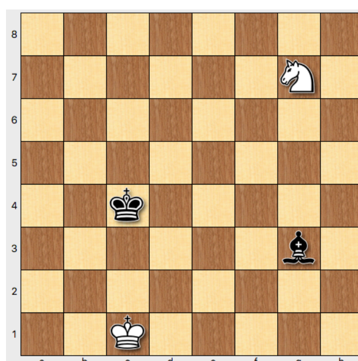
Positive, Negative or Stop (p/n/s)? `p.` (*Changed white Knight's rank*)

tmp(A,king,(B,C),A,D,(E,1),F,king,(G,1),F,H,(I,J),(K,L),(2,4),Pos1) ←
  contents(A,D,(E,1),Pos1), contents(A,king,(B,C),Pos1),
  contents(F,H,(I,J),Pos1), contents(F,king,(G,1),Pos1),
  other_side(A,F),
  legal_move(A,king,(B,C),(2,4),Pos1),
  legal_move(A,D,(E,1),(K,L),Pos1),
  ¬ threat(A,D,(E,1),F,H,(I,J),Pos1),
  make_move(A,D,(E,1),(K,L),Pos1,Pos2),
  threat(A,D,(K,L),F,H,(I,J),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



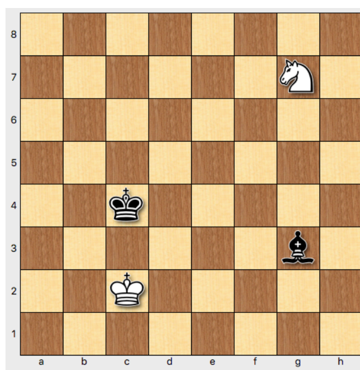
Positive, Negative or Stop (p/n/s)? `n.` (*Changed black Bishop's rank*)



Positive, Negative or Stop (p/n/s)? `p.` (*Changed black Bishop's rank*)

tmp(A,king,(B,C),A,D,(E,F),G,king,(H,1),G,I,(J,K),(L,M),(2,4),Pos1) ←
contents(G,king,(H,1),Pos1), contents(G,I,(J,K),Pos1),
contents(A,king,(B,C),Pos1), contents(A,D,(E,F),Pos1),
other_side(A,G),
legal_move(A,king,(B,C),(2,4),Pos1),
legal_move(A,D,(E,F),(L,M),Pos1),
¬ threat(A,D,(E,F),G,I,(J,K),Pos1),
make_move(A,D,(E,F),(L,M),Pos1,Pos2),
threat(A,D,(L,M),G,I,(J,K),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`

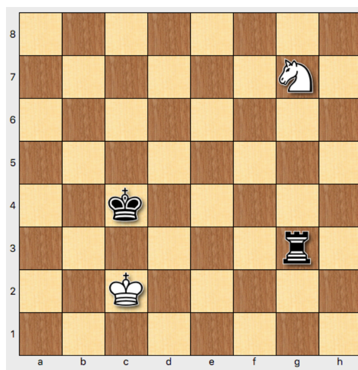


Positive, Negative or Stop (p/n/s)? `p.` (*Changed white King's rank*)

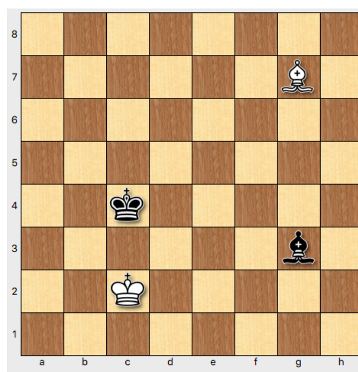
tmp(A,king,(B,C),A,D,(E,F),G,H,(I,J),(K,L),(2,4),Pos1) ←
  contents(A,D,(E,F),Pos1), contents(A,king,(B,C),Pos1),
  contents(G,H,(I,J),Pos1),
  other_side(A,G),
  legal_move(A,king,(B,C),(2,4),Pos1),
  legal_move(A,D,(E,F),(K,L),Pos1),
  ¬ threat(A,D,(E,F),G,H,(I,J),Pos1),
  make_move(A,D,(E,F),(K,L),Pos1,Pos2),
  threat(A,D,(K,L),G,H,(I,J),Pos2).

PAL recognises that the white King is irrelevant to the definition (i.e., its variable arguments in its position are not connected to any other literal in the definition) and eliminates it. It was not eliminated before because it had a constant rank (1).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



Positive, Negative or Stop (p/n/s)? `p.` (*Changed Bishop to Rook*)
The white Knight threatens the Black Rook and it accepted as positive. However the black Rook cannot threat the white Knight after a move (*it is already threatened*).
Too General, it will be ignored

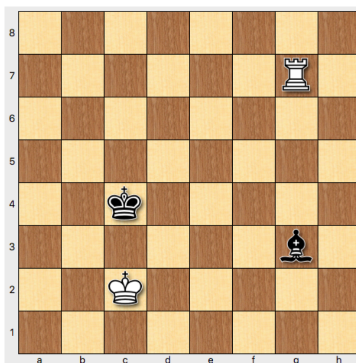


Positive, Negative or Stop (p/n/s)? `p.` (*Changed Knight to Bishop*)

tmp(A,king,(B,C),A,D,(E,F),G,H,(I,J),(K,L),(2,4),Pos1) ←
  contents(G,H,(I,J),Pos1), contents(A,king,(B,C),Pos1),
  contents(A,D,(E,F),Pos1),
  other_side(A,G),
  legal_move(A,king,(B,C),(2,4),Pos1),
  legal_move(A,D,(E,F),(K,L),Pos1),

¬ threat(A,D,(E,F),G,H,(I,J),Pos1),
make_move(A,D,(E,F),(K,L),Pos1,Pos2),
threat(A,D,(K,L),G,H,(I,J),Pos2).

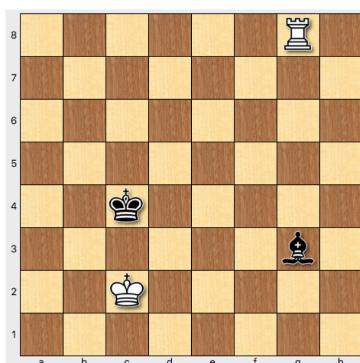
Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



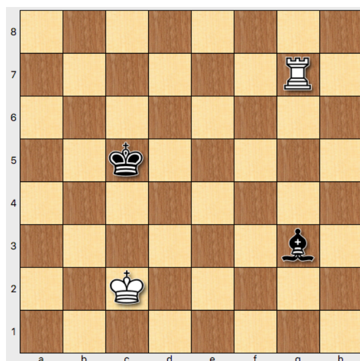
Positive, Negative or Stop (p/n/s)? `p.` (*Changed Bishop to Rook*)

tmp(A,king,(B,C),A,D,(E,F),G,H,(I,J),(K,L),(2,4),Pos1) $\leftarrow$
contents(A,D,(E,F),Pos1), contents(A,king,(B,C),Pos1),
contents(G,H,(I,J),Pos1),
other_side(A,G),
legal_move(A,king,(B,C),(2,4),Pos1),
legal_move(A,D,(E,F),(K,L),Pos1),
¬ threat(A,D,(E,F),G,H,(I,J),Pos1),
make_move(A,D,(E,F),(K,L),Pos1,Pos2),
threat(A,D,(K,L),G,H,(I,J),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



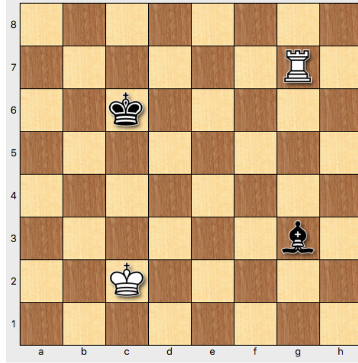
Positive, Negative or Stop (p/n/s)? `n.` (*Changed white Rook's rank*)



Positive, Negative or Stop (p/n/s)? `p.` (*Changed black King's rank*)

tmp(A,king,(B,C),A,D,(E,F),G,H,(I,J),(K,L),(2,4),Pos1) ←
  contents(G,H,(I,J),Pos1), contents(A,king,(B,C),Pos1),
  contents(A,D,(E,F),Pos1),
  other_side(A,G),
  legal_move(A,king,(B,C),(2,4),Pos1),
  legal_move(A,D,(E,F),(K,L),Pos1),
  ¬ threat(A,D,(E,F),G,H,(I,J),Pos1),
  make_move(A,D,(E,F),(K,L),Pos1,Pos2),
  threat(A,D,(K,L),G,H,(I,J),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? `y.`



Positive, Negative or Stop (p/n/s)? `p.` (*Changed black King's rank*)
PAL recognises that the black King is irrelevant and removes it from the definition.

tmp(A,B,(C,D),E,F,(G,H),(I,J),Pos1) ←
contents(A,B,(C,D),Pos1), contents(E,F,(G,H),Pos1),
other_side(A,E),
legal_move(A,B,(C,D),(I,J),Pos1),
¬ threat(A,B,(C,D),E,F,(G,H),Pos1),
make_move(A,B,(C,D),(I,J),Pos1,Pos2),
threat(A,B,(I,J),E,F,(G,H),Pos2).

Accept definition Yes/No/Save-Stop (y/n/s):? s.
At this point the user is satisfied with this definition and stops the process.
I guess is pretty obvious, but ...
How would you like to call the concept? `can_threat.`
PAL checks this definition against all the stored examples (all the positive examples that produced overgeneralisations). Since all of them are covered by this definition, it reduces the definition and the process stops. If one of the stored examples would have failed, PAL would have repeated the whole process with that example, until no more stored examples were left. In this way disjunctive clauses can be learned by PAL. It should be noted that although several overgeneralisations were produced during the learning process, they were all rejected by PAL.

can_threat(A,B,(C,D),E,F,(G,H),(I,J),Pos1) ←
    contents(E,F,(G,H),Pos1),
    other_side(A,E),
    ¬ threat(A,B,(C,D),E,F,(G,H),Pos1),
    make_move(A,B,(C,D),(I,J),Pos1,Pos2),
    threat(A,B,(I,J),E,F,(G,H),Pos2).

yes
|?–

Appendix B

A correct definition of Pin

This appendix, shows the steps followed by PAL to learn the concept of *in_line* and a correct definition of *pin*, as described in chapter 6. It also shows how PAL can learn incrementally by inducing simple concepts first. To obtain the correct definition of *pin*, the system must know that the 3 pieces involved in the concept must be in an horizontal, vertical or diagonal line. Although this could be provided as background knowledge, we can instead include a geometrical concept which recognises wherever two pieces are in a diagonal, vertical, or horizontal line. This can be used to learn a weak version of the 3 pieces in line. The following background vocabulary was given to PAL:

contents(*Side,Piece,Place,Pos*): Describes the positions of each piece. *other_side*(*Side1,Side2*): Side1 is the opponent side of Side2. *all_but_K*(*Piece,Place,Pos*): Piece in Place is not a King. *line*(*Place1,Place2,Pos*): Place1 and Place2 (of two pieces) are in vertical, horizontal or diagonal line.

With the above background knowledge, the following definition was learned by PAL. For presentation purposes *square(X,Y)* is denoted by *(X,Y)* throughout.

w_line(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(S1,S2),
    all_but_K(P3,(X3,Y3),Pos), all_but_K(P2,(X2,Y2),Pos),
    line((X1,Y1),(X3,Y3),Pos), line((X1,Y1),(X2,Y2),Pos),
    line((X2,Y2),(X3,Y3),Pos), line((X2,Y2),(X1,Y1),Pos),
    line((X3,Y3),(X2,Y2),Pos), line((X3,Y3),(X1,Y1),Pos).
```

This pattern definition succeeds whenever there is a King in line with a Piece, and both are in line with an opponent's Piece. This however, does not ensure that the 3 pieces must be in a *straight* line. PAL needs some way of comparing between the relative position of the pieces. The following background vocabulary was added for this purpose.

coordx(Place,X,Pos):	The file of Place is X.
coordy(Place,Y,Pos):	The rank of Place is Y.
less_than(N1,N2):	N1 is less than N2.

PAL learned the concepts where the ranks and/or files of 3 pieces are in 'ascending' or 'descending' order. That is, *ltx3*/10, succeeds whenever there are 3 pieces, a King (K), a piece (P) and an opponent's piece (OP), and their files have the following relation: $K_X < P_X < OP_X$ or $K_X > P_X > OP_X$, where Z_X denotes the file of piece Z.

```
ltx3(Side1,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(S1,S2),
    all_but_K(P2,(X2,Y2),Pos), all_but_K(P3,(X3,Y3),Pos),
    coordx((X1,Y1),X1,Pos), coordx((X2,Y2),X2,Pos),
    coordx((X3,Y3),X3,Pos),
    less_than(X1,X2), less_than(X1,X3), less_than(X2,X3).
ltx3(Side1,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(S1,S2),
    all_but_K(P2,(X2,Y2),Pos), all_but_K(P3,(X3,Y3),Pos),
    coordx((X1,Y1),X1,Pos), coordx((X2,Y2),X2,Pos),
```

```
        coordx((X3,Y3),X3,Pos),
        less_than(X2,X1), less_than(X3,X1), less_than(X3,X2).
```

Similarly, PAL learned an equivalent definition for the ranks of the pieces.

```
lty3(Side1,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(S1,S2),
        all_but_K(P2,(X2,Y2),Pos), all_but_K(P3,(X3,Y3),Pos),
        coordy((X1,Y1),Y1,Pos), coordy((X2,Y2),Y2,Pos),
        coordy((X3,Y3),Y3,Pos),
        less_than(Y1,Y2), less_than(Y1,Y3), less_than(Y2,Y3).
lty3(Side1,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(S1,S2),
        all_but_K(P2,(X2,Y2),Pos), all_but_K(P3,(X3,Y3),Pos),
        coordy((X1,Y1),Y1,Pos), coordy((X2,Y2),Y2,Pos),
        coordy((X3,Y3),Y3,Pos),
        less_than(Y2,Y1), less_than(Y3,Y1), less_than(Y3,Y2).
```

Both definitions can be added to the background knowledge and used by PAL to learn when both relations hold, arriving to the following definition:

```
ltxy3(Side1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos) ←
         ltx3(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos),
         lty3(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos).
```

All of the above definitions can now be added to the background knowledge and used by PAL to learn the concept definition of 3 pieces in a vertical, diagonal, or horizontal line (respectively). PAL produces the following definition:

```
in_line(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos) ←
         w_line(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos),
         ltx3(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos).
in_line(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos) ←
         w_line(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos),
```

```
        lty3(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos).
in_line(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos) ←
        w_line(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos),
        ltxy3(S1,king,(X1,Y1),S1,P2,(X2,Y2),S2,P3,(X3,Y3),Pos).
```

The definition of *in_line*/10 was added to the background knowledge and used to learn a static definition of *pin* after generating 22 + and 60 − examples.

```
pin(S1,P1,(X1,Y1),S2,king,(X2,Y2),S2,P3,(X3,Y3),Pos) ←
    sliding_piece(P1,(X1,Y1),Pos),
    stale(S2,P3,(X3,Y3),Pos),
    threat(S1,P1,(X1,Y1),S2,P3,(X3,Y3),Pos),
    in_line(S2,king,(X2,Y2),S2,P3,(X3,Y3),S1,P1,(X1,Y1),Pos).
```

Appendix C

KRK concepts

This appendix has the concept definitions learned by PAL and used in the King and Rook against King endgame (KRK) strategy. It is assumed that the opponent's King is in the loosing side. Throughout the definitions *square(X,Y)* is denoted as *(X,Y)*, *King* and *Rook* refer to the King and Rook of the winning side, while *OKing* refers to the opponent's King.

- ThreatkR/7: *OKing* threatens *Rook* (Figure 7.2-right).

```
threatkR(A,king,(B,C),D,rook,(E,F),Pos1) ←
    contents(D,rook,(E,F),Pos1),
    contents(A,king,(B,C),Pos1),
    other_side(D,A),
    sliding_piece(rook,(E,F),Pos1),
    legal_move(A,king,(B,C),(E,F),Pos1),
    make_move(A,king,(B,C),(E,F),Pos1,Pos2),
    not contents(D,rook,(E,F),Pos2).
```

- Rook_divs/10: *Rook* divides both, *King* and *OKing*, either vertically or horizontally (Figure 7.1-center after the move).

```
rook_divs(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
    contents(F,king,(G,H),Pos),
    contents(A,king,(B,C),Pos),
    contents(A,rook,(D,E),Pos),
    other_side(A,F),
```

```
        sliding_piece(rook,(D,E),Pos),
        coordx((G,H),G,Pos), coordx((D,E),D,Pos),
        coordx((B,C),B,Pos),
        less_than(G,D), less_than(G,B), less_than(D,B).
rook_divs(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(F,king,(G,H),Pos),
        contents(A,king,(B,C),Pos),
        contents(A,rook,(D,E),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordx((B,C),B,Pos), coordx((D,E),D,Pos),
        coordx((G,H),G,Pos),
        less_than(B,D), less_than(B,G), less_than(D,G).
rook_divs(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(A,rook,(D,E),Pos),
        contents(A,king,(B,C),Pos),
        contents(F,king,(G,H),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordy((B,C),C,Pos), coordy((D,E),E,Pos),
        coordy((G,H),H,Pos),
        less_than(E,C), less_than(H,C), less_than(H,E).
rook_divs(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(A,rook,(D,E),Pos),
        contents(A,king,(B,C),Pos),
        contents(F,king,(G,H),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordy((G,H),H,Pos), coordy((D,E),E,Pos),
        coordy((B,C),C,Pos),
        less_than(E,H), less_than(C,H), less_than(C,E).
```

- Opposition/8: *King* and *OKing* are on the same rank or file with one square between them (Figure 7.1-left).

```
opposition(A,king,(B,C),D,king,(B,E),2,Pos) ←
        contents(D,king,(B,E),Pos),
        contents(A,king,(B,C),Pos),
```

```
        other_side(A,D),
        distance((B,C),(B,E),2,Pos),
        distance((B,E),(B,C),2,Pos).
opposition(A,king,(B,C),D,king,(E,C),2,Pos) ←
        contents(A,king,(B,C),Pos),
        contents(D,king,(E,C),Pos),
        other_side(D,A),
        distance((E,C),(B,C),2,Pos),
        distance((B,C),(E,C),2,Pos).
```

- Almost_opposition/9: *King* is "almost" in opposition with *OKing* (Figure 7.2-left after the move).

```
almost_opposition(A,king,(B,C),D,king,(E,F),2,3,Pos) ←
        contents(D,king,(E,F),Pos),
        contents(A,king,(B,C),Pos),
        other_side(A,D),
        distance((B,C), (E,F),2,Pos),
        distance((E,F), (B,C),2,Pos),
        manh_dist((B,C),(E,F),3,Pos),
        manh_dist((E,F),(B,C),3,Pos).
```

- R_edge/4: *Rook* is on an edge of the board (Figure 7.2-right after the move).

```
r_edge(A,rook,(B,1),Pos) ←
       contents(A,rook,(B,1),Pos),
       sliding_piece(rook,(B,1),Pos).
r_edge(A,rook,(B,8),Pos) ←
       contents(A,rook,(B,8),Pos),
       sliding_piece(rook,(B,8),Pos).
r_edge(A,rook,(8,B),Pos) ←
       contents(A,rook,(8,B),Pos),
       sliding_piece(rook,(8,B),Pos).
r_edge(A,rook,(1,B),Pos) ←
       contents(A,rook,(1,B),Pos),
       sliding_piece(rook,(1,B),Pos).
```

- Kings_same_side/10: Both, *King* and *OKing*, are to the left, right, above, or below of *Rook* (Figure 7.1-center before the move).

```
kings_same_side(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
    contents(F,king,(G,H),Pos),
    contents(A,king,(B,C),Pos),
    contents(A,rook,(D,E),Pos),
    other_side(A,F),
    sliding_piece(rook,(D,E),Pos),
    coordx((D,E),D,Pos), coordx((G,H),G,Pos),
    coordx((B,C),B,Pos),
    less_than(D,G), less_than(D,B).
kings_same_side(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
    contents(A,king,(B,C),Pos),
    contents(A,rook,(D,E),Pos),
    contents(F,king,(G,H),Pos),
    other_side(F,A),
    sliding_piece(rook,(D,E),Pos),
    coordy((G,H),H,Pos), coordy((D,E),E,Pos),
    coordy((B,C),C,Pos),
    less_than(H,E), less_than(C,E).
kings_same_side(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
    contents(F,king,(G,H),Pos),
    contents(A,king,(B,C),Pos),
    contents(A,rook,(D,E),Pos),
    other_side(A,F),
    sliding_piece(rook,(D,E),Pos),
    coordy((D,E),E,Pos), coordy((G,H),H,Pos),
    coordy((B,C),C,Pos),
    less_than(E,H), less_than(E,C).
kings_same_side(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
    contents(A,rook,(D,E),Pos),
    contents(A,king,(B,C),Pos),
    contents(F,king,(G,H),Pos),
    other_side(A,F),
    sliding_piece(rook,(D,E),Pos),
    coordx((D,E),D,Pos), coordx((G,H),G,Pos),
    coordx((B,C),B,Pos),
```

less_than(G,D), less_than(B,D).

- L_patt/12: The 3 pieces form an "L-shaped" pattern with *OKing* in check by *Rook* and both Kings in *oppositions* (Figure 7.1-left after the move).

```
l_patt(A,king,(B,C),D,king,(E,C),D,rook,(B,F),2,(E,F),Pos) ←
    contents(D,king,(E,C),Pos),
    contents(D,rook,(B,F),Pos),
    contents(A,king,(B,C),Pos),
    other_side(A,D),
    sliding_piece(rook,(B,F),Pos),
    in_check(A,(B,C),rook,(B,F),Pos),
    legal_move(D,rook,(B,F),(E,F),Pos),
    opposition(A,king,(B,C),D,king,(E,C),2,Pos),
    opposition(D,king,(E,C),A,king,(B,C),2,Pos).
l_patt(A,king,(B,C),D,king,(B,E),D,rook,(F,C),2,(F,E),Pos) ←
    contents(A,king,(B,C),Pos),
    contents(D,king,(B,E),Pos),
    contents(D,rook,(F,C),Pos),
    other_side(D,A),
    sliding_piece(rook,(F,C),Pos),
    in_check(A,(B,C),rook,(F,C),Pos),
    legal_move(D,rook,(B,F),(F,E),Pos),
    opposition(D,king,(B,E),A,king,(B,C),2,Pos),
    opposition(A,king,(B,C),D,king,(B,E),2,Pos).
```

- RKk/10: *OKing* divides *Rook* and *King* either vertically or horizontally (Figure 7.2-right before the move).

```
rKk(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
      contents(F,king,(G,H),Pos),
      contents(A,king,(B,C),Pos),
      contents(A,rook,(D,E),Pos),
      other_side(A,F),
      sliding_piece(rook,(D,E),Pos),
      coordx((D,E),D,Pos), coordx((G,H),G,Pos),
```

```
        coordx((B,C),B,Pos),
        less_than(D,G), less_than(D,B), less_than(G,B).
rKk(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(F,king,(G,H),Pos),
        contents(A,king,(B,C),Pos),
        contents(A,rook,(D,E),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordx((B,C),B,Pos), coordx((G,H),G,Pos),
        coordx((D,E),D,Pos),
        less_than(B,G), less_than(B,D), less_than(G,D).
rKk(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(A,rook,(D,E),Pos),
        contents(A,king,(B,C),Pos),
        contents(F,king,(G,H),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordy((B,C),C,Pos), coordy((G,H),H,Pos),
        coordy((D,E),E,Pos),
        less_than(H,C), less_than(E,C), less_than(E,H).
rKk(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(A,rook,(D,E),Pos),
        contents(A,king,(B,C),Pos),
        contents(F,king,(G,H),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordy((D,E),E,Pos), coordy((G,H),H,Pos),
        coordy((B,C),C,Pos),
        less_than(H,E), less_than(C,E), less_than(C,H).
```

- RkK/10: *King* divides *Rook* and *OKing* either vertically or horizontally (Figure 7.2-right after the move).

```
rkK(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(F,king,(G,H),Pos),
        contents(A,king,(B,C),Pos),
        contents(A,rook,(D,E),Pos),
        other_side(A,F),
```

```
        sliding_piece(rook,(D,E),Pos),
        coordx((D,E),D,Pos), coordx((B,C),B,Pos),
        coordx((G,H),G,Pos),
        less_than(D,B), less_than(D,G), less_than(B,G).
rkK(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(A,rook,(D,E),Pos),
        contents(A,king,(B,C),Pos),
        contents(F,king,(G,H),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordx((D,E),D,Pos), coordx((B,C),B,Pos),
        coordx((G,H),G,Pos),
        less_than(B,D), less_than(G,D), less_than(G,B).
rkK(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(A,rook,(D,E),Pos),
        contents(A,king,(B,C),Pos),
        contents(F,king,(G,H),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordy((G,H),H,Pos), coordy((B,C),C,Pos),
        coordy((D,E),E,Pos),
        less_than(C,H), less_than(E,H), less_than(E,C).
rkK(A,king,(B,C),A,rook,(D,E),F,king,(G,H),Pos) ←
        contents(F,king,(G,H),Pos),
        contents(A,king,(B,C),Pos),
        contents(A,rook,(D,E),Pos),
        other_side(A,F),
        sliding_piece(rook,(D,E),Pos),
        coordy((G,H),H,Pos), coordy((B,C),C,Pos),
        coordy((D,E),E,Pos),
        less_than(H,C), less_than(H,E), less_than(C,E).
```

- CloserKk/10: The Manhattan distance of *King* to *OKing* decreases after a move (e.g., Figure 7.2-left).

```
closerKk(A,king,(B,C),D,king,(E,F),G,H,(I,J),Pos1) ←
      contents(D,king,(E,F),Pos1),
      contents(A,king,(B,C),Pos1),
```

```
    other_side(D,A),
    manh_dist((E,F),(B,C),G,Pos1),
    make_move(A,king,(B,C),(I,J),Pos1,Pos2),
    manh_dist((E,F),(I,J),H,Pos2),
    less_than(H,G).
```

- RcloserKk/10: The distance of *King* to *OKing* decreases after a move.

```
rcloserKk(A,king,(B,C),D,king,(E,F),G,H,(I,J),Pos1) ←
    contents(A,king,(B,C),Pos1),
    contents(D,king,(E,F),Pos1),
    other_side(A,D),
    distance((E,F),(B,C),G,Pos1),
    make_move(A,king,(B,C),(I,J),Pos1,Pos2),
    distance((E,F),(I,J),H,Pos2),
    less_than(H,G).
```

- AwayRk/10: The distance of *OKing* to *Rook* increases after a move by *Rook* (e.g., Figure 7.2-right after the move).

```
awayRk(A,rook,(B,C),D,king,(E,F),G,H,(I,J),Pos1) ←
    contents(A,rook,(B,C),Pos1),
    contents(D,king,(E,F),Pos1),
    other_side(A,D),
    sliding_piece(rook,(B,C),Pos1),
    restr_distance((E,F),(B,C),G,Pos1),
    make_move(A,rook,(B,C),(I,J),Pos1,Pos2),
    restr_distance((E,F),(I,J),H,Pos2),
    less_than(G,H).
```

- RawayRk/10: The Manhattan distance of *OKing* to *Rook* increases after a move by *Rook* (e.g., Figure 7.2-right after the move).

```
rawayRk(A,rook,(B,C),D,king,(E,F),G,H,(I,J),Pos1) ←
    contents(A,rook,(B,C),Pos1),
    contents(D,king,(E,F),Pos1),
    other_side(A,D),
    sliding_piece(rook,(B,C),Pos1),
    restr_manh_dist((E,F),(B,C),G,Pos1),
    make_move(A,rook,(B,C),(I,J),Pos1,Pos2),
    restr_manh_dist((E,F),(I,J),H,Pos2),
    less_than(G,H).
```

- RcloserRk/10: The Manhattan distance between *Rook* and *OKing* decreases after a move by *Rook* (Figure 7.1-right).

```
rcloserRk(A,king,(B,C),D,rook,(E,F),G,H,(E,I),Pos1) ←
    contents(D,rook,(E,F),Pos1),
    contents(A,king,(B,C),Pos1),
    other_side(A,D),
    sliding_piece(rook,(E,F),Pos1),
    restr_manh_dist((B,C),(E,F),G,Pos1),
    make_move(D,rook,(E,F),(E,I),Pos1,Pos2),
    restr_manh_dist((B,C),(E,I),H,Pos2),
    less_than(H,G).
rcloserRk(A,king,(B,C),D,rook,(E,F),G,H,(I,F),Pos1) ←
    contents(A,king,(B,C),Pos1),
    contents(D,rook,(E,F),Pos1),
    other_side(A,D),
    sliding_piece(rook,(E,F),Pos1),
    restr_manh_dist((B,C),(E,F),G,Pos1),
    make_move(D,rook,(E,F),(I,F),Pos1,Pos2),
    restr_manh_dist((B,C),(I,F),H,Pos2),
    less_than(H,G).
```

- In_line/7: *Rook* and *King* are on the same rank or file.

```
in_lineRK(A,king,(B,C),A,rook,(B,D),Pos) ←
  contents(A,rook,(B,D),Pos),
```

```
    contents(A,king,(B,C),Pos),
    sliding_piece(rook,(B,D),Pos).
in_lineRK(A,king,(B,C),A,rook,(D,C),Pos) ←
    contents(A,king,(B,C),Pos),
    contents(A,rook,(D,C),Pos),
    other_side(A,E),
    sliding_piece(rook,(D,C),Pos).
```

- CloserRK2/8: The distance between *Rook* and *King* is 2 (e.g., Figure 7.1-center).

```
closeRK2(A,rook,(B,C),A,king,(D,E),2,Pos) ←
    contents(A,rook,(B,C),Pos),
    contents(A,king,(D,E),Pos),
    sliding_piece(rook,(B,C),Pos),
    distance((D,E),(B,C),2,Pos),
    distance((B,C),(D,E),2,Pos).
```

- DistkR6/8: The distance between *Rook* and *OKing* is 6.

```
distkR6(A,rook,(B,C),D,king,(E,F),6,Pos) ←
    contents(A,rook,(B,C),Pos),
    contents(D,king,(E,F),Pos),
    other_side(A,D),
    sliding_piece(rook,(B,C),Pos),
    distance((E,F),(B,C),6,Pos),
    distance((B,C),(E,F),6,Pos).
```

Appendix D

KRK rules

This appendix has the rules used for the playing strategy of the KRK endgame. The rules are tried in order. If the conditions before a move hold, and the move makes the conditions after the move to succeed, then that move is followed. The 1-ply strategy rules use the concept definitions described in appendix C.

R1: **if** opposition, **and**
with move move rook,
ensure check_mate.

R2: **if** rook_divs, **and**
opposition, **and**
with move move rook,
ensure in_check, **and**
not threatkR, **and**
l_patt.

R3: **if** rook_divs, **and**
opposition, **and**
with move move rook,
ensure r_edge, **and**
rook_divs, **and**
opposition.

R4: **if** rook_divs, **and**
threatkR, **and**
rKk, **and**
with move move rook,
ensure r_edge, **and**
rook_divs, **and**
not threatkR, **and**
rkK.

R5: **if** rook_divs, **and**
threatkR, **and**
not r_edge, **and**
with move move rook,
ensure r_edge, **and**
rook_divs, **and**
not threatkR.

R6: **if** rook_divs, **and**
threatkR, **and**
r_edge, **and**
with move move rook,
ensure r_edge, **and**
distkR6, **and**
rook_divs.

R7: **if** rook_divs, **and**
r_edge, **and**
closeRK2, **and**
almost_opposition, **and**
with move move rook,
ensure r_edge, **and**
not stale, **and**
not in_check, **and**
almost_opposition, **and**
rook_divs, **and**
not threatkR.

R8: **if** rook_divs, **and**
almost_opposition, **and**
with move move rook,
ensure not stale, **and**
not in_check, **and**
almost_opposition, **and**
rook_divs, **and**
not threatkR.

R9: **if** **with move** move king,
ensure almost_opposition, **and**
rook_divs, **and**
not stale, **and**
not threatkR.

R10: **if** rook_divs, **and**
closerKk, **and**
with move move king,
ensure opposition, **and**
not stale.

R11: **if** rook_divs, **and**
str_closerKk, **and**
closerKk, **and**
with move move king,
ensure not threatkR, **and**
not stale, **and**
rook_divs.

R12: **if** rook_divs, **and**
rcloserRk, **and**
with move move rook,
ensure rook_divs, **and**
not threatkR.

R13: **if** rook_divs, **and**
closerKk, **and**
with move move king,
ensure not threatkR, **and**

R14: **if** rook_divs, **and**
awayRk, **and**
with move move rook,
ensure rook_divs, **and**

not stale, **and**
rook_divs.

not threatkR.

R15: **if** kings_same_side, **and**
with move move rook,
ensure rook_divs, **and**
not stale, **and**
not threatkR.

R16: **if** kings_same_side, **and**
in_lineRK, **and**
with move move rook,
ensure not in_lineRK, **and**
not stale, **and**
not threatkR.

R17: **if** kings_same_side, **and**
r_edge, **and**
with move move rook,
ensure not r_edge, **and**
not stale, **and**
not threatkR.

R18: **if** kings_same_side, **and**
rawayRk, **and**
with move move rook,
ensure r_edge, **and**
not stale, **and**
not threatkR.

Appendix E

The improved KRK strategy

This appendix has the concepts learned by PAL and the rules used in the improved KRK strategy. Besides the background knowledge specified in chapter 7, the definition of "confined area" was provided to the system.

area(*Place1,Place2,Room,Pos*):
Rook at Place1 confines the opponent's King at Place2 to an area of size Room

E.1 Concepts

Throughout the definitions *square(X,Y)* is denoted as *(X,Y)*. *King* and *Rook* refer to the King and Rook of the winning side, while *OKing* refers to the opponent's King.

- Squeeze/12: The area on which *OKing* is confined by *Rook* is reduced after a movement of *Rook*. PAL recognises as well that the area disappears when *OKing* is in check.

```
squeeze(A,king,(B,C),D,rook,(E,F),G,H,(E,C),(B,F),(I,J),Pos1) ←
        contents(D,rook,(E,F),Pos1),
        contents(A,king,(B,C),Pos1),
        other_side(D,A),
        sliding_piece(rook,(E,F),Pos1),
        legal_move(D,rook,(E,F),(E,C),Pos1),
        legal_move(D,rook,(E,F),(B,F),Pos1),
```

```
        area((E,F),(B,C),G,Pos1),
        make_move(D,rook,(E,F),(E,C),Pos1,Pos2),
        ¬ area((E,C),(B,C),G,Pos2),
        make_move(D,rook,(E,F),(I,J),Pos1,Pos3),
        area((I,J),(B,C),H,Pos3),
        less_than(H,G),
        make_move(D,rook,(E,F),(B,F),Pos1,Pos4),
        ¬ area((B,F),(B,C),G,Pos4).
```

- <u>CloserkKR/12</u>: *OKing* is closer to *Rook* than *King*.

```
closerkKR(A,king,(B,C),D,king,(E,F),D,rook,(G,H),I,J,Pos) ←
        contents(A,king,(B,C),Pos),
        contents(D,king,(E,F),Pos),
        contents(D,rook,(G,H),Pos),
        other_side(D,A),
        sliding_piece(rook,(G,H),Pos),
        distance((G,H),(B,C),J,Pos),
        distance((E,F),(G,H),I,Pos),
        less_than(J,I).
```

- <u>CloserKkR/12</u>: *King* is closer to *Rook* than *OKing*.

```
closerKkR(A,king,(B,C),D,king,(E,F),D,rook,(G,H),I,J,Pos) ←
        contents(A,king,(B,C),Pos),
        contents(D,king,(E,F),Pos),
        contents(D,rook,(G,H),Pos),
        other_side(D,A),
        sliding_piece(rook,(G,H),Pos),
        distance((G,H),(E,F),J,Pos),
        distance((B,C),(G,H),I,Pos),
        less_than(J,I).
```

- <u>K_trap/7</u>: *OKing* is "trapped" by *Rook* in a border.

```
k_trap(A,rook,(B,2),C,king,(D,1),Pos) ←
```

```
        contents(A,rook,(B,2),Pos),
        contents(C,king,(D,1),Pos),
        other_side(A,C),
        sliding_piece(rook,(B,2),Pos).
k_trap(A,rook,(2,B),C,king,(1,D),Pos) ←
        contents(A,rook,(2,B),Pos),
        contents(C,king,(1,D),Pos),
        other_side(A,C),
        sliding_piece(rook,(2,B),Pos).
k_trap(A,rook,(B,7),C,king,(D,8),Pos) ←
        contents(A,rook,(B,7),Pos),
        contents(C,king,(D,8),Pos),
        other_side(A,C),
        sliding_piece(rook,(B,7),Pos).
k_trap(A,rook,(7,B),C,king,(8,D),Pos) ←
        contents(A,rook,(7,B),Pos),
        contents(C,king,(8,D),Pos),
        other_side(A,C),
        sliding_piece(rook,(7,B),Pos).
```

- DistKR1/12: The distance between *Rook* and *King* is one.

```
distKR1(A,king,(B,C),A,rook,(D,E),F,king,(G,H),I,1,Pos) ←
            contents(F,king,(G,H),Pos),
            contents(A,king,(B,C),Pos),
            contents(A,rook,(D,E),Pos),
            other_side(A,F),
            sliding_piece(rook,(D,E),Pos),
            distance((D,E),(B,C),1,Pos),
            distance((B,C),(D,E),1,Pos),
            distance((G,H),(B,C),I,Pos),
            less_than(1,I).
```

- mvKcloserR/10: *King* gets closer to *Rook* after a move.

```
mvKcloserR(A,king,(B,C),A,rook,(D,E),F,G,(H,I),Pos1) ←
                contents(A,king,(B,C),Pos1),
```

contents(A,rook,(D,E),Pos1),
sliding_piece(rook,(D,E),Pos1),
restr_distance((B,C),(D,E),F,Pos1),
make_move(A,king,(B,C),(H,I),Pos1,Pos2),
restr_distance((H,I),(D,E),G,Pos2),
less_than(G,F).

- Corner/7: *OKing* is in a corner with *King* in diagonal opposition.

corner(A,king,(3,3),B,king,(1,1),Pos) ←
contents(A,king,(3,3),Pos), contents(B,king,(1,1),Pos),
other_side(A,B).
corner(A,king,(3,6),B,king,(1,8),Pos) ←
contents(A,king,(3,6),Pos), contents(B,king,(1,8),Pos),
other_side(A,B).
corner(A,king,(6,3),B,king,(8,1),Pos) ←
contents(A,king,(6,3),Pos), contents(B,king,(8,1),Pos),
other_side(A,B).
corner(A,king,(6,6),B,king,(8,8),Pos) ←
contents(A,king,(6,6),Pos), contents(B,king,(8,8),Pos),
other_side(A,B).

- Corner2/7: *OKing* is "almost" in a corner with *King* "almost" in opposition.

corner2(A,king,(3,3),B,king,(1,2),Pos) ←
contents(A,king,(3,3),Pos), contents(B,king,(1,2),Pos),
other_side(A,B).
corner2(A,king,(3,3),B,king,(2,1),Pos) ←
contents(A,king,(3,3),Pos), contents(B,king,(2,1),Pos),
other_side(A,B).
corner2(A,king,(3,6),B,king,(1,7),Pos) ←
contents(A,king,(3,6),Pos), contents(B,king,(1,7),Pos),
other_side(A,B).
corner2(A,king,(3,6),B,king,(2,8),Pos) ←
contents(A,king,(3,6),Pos), contents(B,king,(2,8),Pos),
other_side(A,B).

```
corner2(A,king,(6,3),B,king,(8,2),Pos) ←
    contents(A,king,(6,3),Pos), contents(B,king,(8,2),Pos),
    other_side(A,B).
corner2(A,king,(6,3),B,king,(7,1),Pos) ←
    contents(A,king,(6,3),Pos), contents(B,king,(7,1),Pos),
    other_side(A,B).
corner2(A,king,(6,6),B,king,(8,7),Pos) ←
    contents(A,king,(6,6),Pos), contents(B,king,(8,7),Pos),
    other_side(A,B).
corner2(A,king,(6,6),B,king,(7,8),Pos) ←
    contents(A,king,(6,6),Pos), contents(B,king,(7,8),Pos),
    other_side(A,B).
```

In addition to the above concepts, the following concepts, used in the first KRK strategy, were used in the improved strategy (they are described in appendix C).

- *rook_divs*: *Rook* divides both, *King* and *OKing* either vertically or horizontally.
- *opposition*: Both, *King* and *OKing* are in opposition.
- *threatkR*: *OKing* threatens *Rook*.
- *alm_oppos*: Both, *King* and *OKing*, are almost in opposition.
- *r_edge*: *Rook* is on a border.
- *rkK*: *King* divides *Rook* and *OKing* either vertically or horizontally.

Table E.1 shows the examples generated by PAL with the additional background knowledge used for each concept.

E.2 Rules

The following rules are used in the improved playing strategy of the KRK endgame. The rules are tried in order and followed a similar format as those given in appendix D.

Concept	Generated Examples	Add. Back. Knowledge
squeeze/12	27 + 20 −	area
closerkKR/12	20 + 4 −	distance
closerKkR/12	16 + 3 −	distance
k_trap/7	43 + 12 −	
distKR1/12	16 + 7 −	
mvKcloserR/10	20 + 2 −	restr_distance
corner/6	4 + 0 −	
corner2/7	8 + 0 −	

Table E.1: Table of results for the new KRK concepts

R1: **if** k_edge **and**
with move move rook
ensure check_mate.

R2: **if** corner **and**
not distKR1 **and**
k_trap **and**
rook_divs **and**
with move move king
ensure almost_opposition **and**
rook_divs.

R3: **if** corner **and**
k_trap **and**
with move move rook
ensure k_trap **and**
rkK.

R4: **if** corner2 **and**
almost_opposition **and**
k_trap **and**
rook_divs **and**
with move move rook
ensure not distKR1 **and**
k_trap **and**
rkK.

R5: **if** squeeze **and**
with move move
ensure distKR1 **and**
not stale.

R6: **if** squeeze **and**
with move move rook
ensure not closerkKR **and**
not stale.

R7: **if** k_trap **and**
almost_opposition **and**

R8: **if** rook_divs **and**
with move move king

with move move king
ensure opposition **and**
not threatkR **and**
not stale.

ensure almost_opposition **and**
rook_divs **and**
not stale **and**
not threatkR.

R9: **if** rook_divs **and**
mvKcloserk **and**
with move move king
ensure opposition **and**
not stale **and**
not threatkR

R10: **if** mvKcloserk **and**
mvKcloserR **and**
with move move king
ensure not closerkKR **and**
not stale.

R11: **if** not rook_divs **and**
with move move rook
ensure rook_divs **and**
not stale **and**
not threatkR.

R12: **if** mvKcloserR **and**
with move move king
ensure not stale **and**
not threatkR.

R13: **if** rook_divs **and**
mvKcloserk **and**
with move move king
ensure rook_divs **and**
not threatkR **and**
not stale.

R14: **if** mvKcloserk **and**
with move move king
ensure not closerkKR **and**
not stale.

R15: **if with move** move king
ensure not closerkKR **and**
not stale.

R16: **if** k_trap **and**
threatkR **and**
with move move rook
ensure k_trap **and**
closerKkR **and**
not stale **and**
not threatkR.

R17: **if** k_trap **and**
threatkR **and**
with move move rook
ensure k_trap **and**
r_edge **and**
not stale **and**

R18: **if** k_trap **and**
stale **and**
with move move rook
ensure k_trap **and**
not stale **and**
not threatkR.

not threatkR.

R19: **if with move** move rook
ensure closerKkR.

Appendix F

Learning Can-Run for KPK

This appendix has the description of the patterns learned by PAL to decide whether a white Pawn can safely promote from any white to move position without moving the white King, in a white King and Pawn against black King endgame. In addition to the background knowledge specified in Chapter 5, the following background vocabulary was included for this endgame.

distance(*Place1*,*Place2*,*Dist*,*Pos*):
    Distance (*Dist*) between a piece in *Place1* and a piece in *Place2*.
manh_dist(*Place1*,*Place2*,*Dist*,*Pos*):
    Manhattan distance between two pieces.
dist_qs(*Place*,*Dist*,*Pos*):
    Distance (*Dist*) between a piece in *Place* and the queening square.
coordx(*Place*,*X*,*Pos*):
    The file of a piece in *Place*.
coordy(*Place*,*Y*,*Pos*):
    The rank of a piece in *Place*.

The perturbation strategy was limited to changes only in the places of the pieces (i.e., no changes in the sides or the pieces were made).

F.1 Concepts

Only a broad description of the patterns is given below. Similarly to the construction of the KRK playing strategies, an initial set of patterns was learned by PAL, which was refined through experimentation. The final set was checked against an exhaustive database generated for this purpose.

The strategy consists of 18 "main" patterns with the following format:

$$can_run \textbf{ if } pattern_X,\ pattern_Y,\ \ldots$$

All the patterns learned by PAL are static, except one which involves a 1-ply movement to check whether a stalemate can be created. An additional rule, not learned by PAL but constructed with patterns learned by PAL, was included to the strategy to consider the case where the white Pawn is on the second rank.

The following strategy decides whether a white Pawn can be safetely promoted by moving exclusively the Pawn (i.e., can run). The patterns are tried in order. Given a KPK position a white Pawn can run if one of the main patterns is satisfied for that position. In total, 34 "sub"-patterns were learned by PAL and used in the definition of the main patterns. There are two main reasons for this sub-division:

- It is easier to learn simplier patterns.
- A sub-pattern can be used in several patterns.

Similarly to the KRK endgame, the "quality" of the background knowledge can affect the strategy. A much simplier strategy can be constructed for this endgame if the definition of distance to the queening square considers the possible interaction between Kings.

In the descriptions P refers to the white Pawn, K to the white King, and k to the black King. P_x and P_y refers to the file and rank of the white Pawn (similar notations are used for the other pieces). $dist(A,B,D)$ means that the distance between a piece A and a piece B is D. $dist_qs(A,D)$ means that the distance of piece A to the queening square is D. $manh_dist(A,B,D)$ and $manh_dist_qs(A,D)$, refer to the Manhattan distance between two pieces and the Manhattan distance between a piece and the queening square. A at(X,Y) denotes a piece A at $square(X,Y)$. $move(P)$ denotes a movement of the white Pawn and $stale(k)$ denotes that the black King cannot move.

can_run **if** { [k at $(1,4)$, P at $(2,2)$, K at $(3,4)$] or
[k at $(1,4)$, P at $(2,2)$, K at $(3,5)$] or
[k at $(8,4)$, P at $(7,2)$, K at $(6,4)$] or
[k at $(8,4)$, P at $(7,2)$, K at $(6,5)$] }

can_run **if** $P_y = 2$, *move*(P), $P_y = 3$, *can_run*

can_run **if** not { [k at $(1,8)$, K at $(3,7)$, P at $(2,3)$] or
[k at $(1,7)$, K at $(3,7)$, P at $(2,4)$] or
[k at $(8,8)$, K at $(6,7)$, P at $(7,3)$] or
[k at $(8,7)$, K at $(6,7)$, P at $(7,4)$] or
[*move*(P), *stale*(k)] },
[$K_x = P_x, K_y > P_y$],
can_run1.

can_run1 **if** *dist_qs*($P, D1$), *dist_qs*($k, D2$), $D1 < D2$.

can_run1 **if** *dist_qs*($K, 1$), ($P_y = 6$ or $P_y = 7$).

can_run1 **if** *dist_qs*($K, 1$), $K_y = 7, P_y = 5$.

can_run1 **if** ([$k_x < K_x < P_x$] or [$k_x > K_x > P_x$]),
$K_y = 6, K_y > k_y$.

can_run1 **if** ([$k_x < K_x < P_x$] or [$k_x > K_x > P_x$]),
$K_y = 7$, *dist*(K, P, D),
($D = 1$ or $D = 2$ or $D = 3$).

can_run1 **if** ([$k_x < K_x < P_x$] or [$k_x > K_x > P_x$]),
$K_y = 7$, *dist*(K, P, D),
($D \neq 1$ or $D \neq 2$ or $D \neq 3$),
dist($K, P, D2$), *manh_dist_qs*($K, D2$).

can_run1 **if** {([$k_x < K_x < P_x$] or [$k_x > K_x > P_x$]) or
[$K_x = k_x$, *dist*($K, k, 2$)] },
$P_y = k_y = K_y - 2$.

can_run1 **if** ([$k_x < K_x < P_x$] or [$k_x > K_x > P_x$]),

$K_y = 6,\ dist_qs(K, 2),\ dist_qs(P, D1),\ dist_qs(k, D1).$

can_run1 **if** $([k_x < K_x < P_x]$ or $[k_x > K_x > P_x]),$
$K_y \neq 8,$
$([\ K_y = k_y,\ dist(K, k, 2)\ ]$ or $[\ K_x = k_x,\ dist(K, k, 2)\ ]),$
$manh_dist(K, k, D),\ (D \neq 3$ or $D \neq 4).$
(For $P_y = 3$ and $P_y = 4$ additional conditions are required)

can_run1 **if** $([k_x < K_x < P_x]$ or $[k_x > K_x > P_x]),$
$K_y = 8,\ dist(P, k, D1),\ dist_qs(P, D1),\ dist_qs(K, 2).$

can_run1 **if** $([k_x < K_x < P_x]$ or $[k_x > K_x > P_x]),$
$K_y = 8, P_y = 3,$
$([\ K_y = k_y,\ dist(K, k, 2)]$ or $[\ K_x = k_x,\ dist(K, k, 2)]).$

can_run1 **if** $k_y = 8, K_y = 6, P_y = 6,\ dist_qs(K, 2),\ dist_qs(P, 2),\ dist_qs(k, 2).$

can_run1 **if** $dist_qs(K, 1),\ dist_qs(P, D1),\ dist(P, k, D2),$
$(D1 = D2$ or $D1 + 1 = D2),$
not $\{K_y \neq 8, k_y \neq 8,\ ([k_x < P_x < K_x]$ or $[k_x > P_x > K_x])\ \}.$

can_run1 **if** $dist_qs(K, 1),\ K_y = 7,\ dist(k, P, D1),\ dist(K, P, D2),\ D1 + 1 = D2.$

can_run1 **if** $dist_qs(K, 1),\ P_y = 4, K_y = k_y, Ky \neq 8.$

can_run1 **if** $dist_qs(K, 1),\ P_y = 3,\ dist_qs(k, 3),\ K_y = k_y, Ky \neq 8.$

Appendix G

Background knowledge definitions

This appendix has the background knowledge definitions used by PAL. It is assumed that the reader is familiar with Prolog. An estimated time of no more than one week was required to define the whole background knowledge definitions.

- legal_move: Defines legal moves of chess pieces. A piece *Piece* can move to place *Place* if it does not create a check on its own King.

```
legal_move(Side,Piece,Place,NewPlace,Pos1) :-
        contents(Side,Piece,Place,Pos1),
        piece_move(Side,Piece,Place,NewPlace,Pos1),
        do_move(Side,Piece,Place,NewPlace,Pos1,Pos2),
        not in_check(Side,_,_,_,Pos2).
```

- in_check: An opponent's piece has a plausible move to the place of the King.

```
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).
```

- check_mate: A King is in check and cannot move (it does not consider possible blockades from other pieces).

```
check_mate(Side,Place,Pos) :-
        contents(Side,king,Place,Pos),
        in_check(Side,Place,_,_,Pos),
        not legal_move(Side,king,Place,_,Pos).
```

- stale: A piece cannot move and the opponent's King is not in check.

```
stale(Side,Piece,Place,Pos) :-
      contents(Side,Piece,Place,Pos),
      not legal_move(Side,Piece,Place,_,Pos),
      other_side(Side,OSide),
      not in_check(OSide,_,_,_,Pos).
```

- make_move: Performs the actual movement of a piece changing the state description. It checks first that the opponent's King in not in check.

```
make_move(Side,Piece,Place,NewPlace,Pos1,Pos2) :-
          legal_move(Side,Piece,Place,NewPlace,Pos1),
          other_side(Side,OSide),
          not in_check(OSide,_,_,_,Pos1),
          do_move(Side,Piece,Place,NewPlace,Pos1,Pos2).
```

- other_side: White is the opponent side of Black (and vice verse).

```
other_side(white,black).
other_side(black,white).
```

- sliding_piece: A Queen, Bishop or Rook.

```
sliding_piece(Piece,Place,Pos) :-
      contents(_Side,Piece,Place,Pos),
      member(Piece,[queen,bishop,rook]).
```

The following predicates are used to safe/restore current chess board descriptions.

- do_move: changes the current state description.

```
do_move(Side,Piece,Place,NewPlace,Pos1,Pos2) :-
      current_state(State),
      create_new_state(Pos1,State,Pos2,NState),
      retract_if_there(contents(_,_,NewPlace,Pos2)),
      retract(contents(Side,Piece,Place),Pos2),
      asserta(contents(Side,Piece,NewPlace),Pos2),
      !,
      restore_if_redo(State).
```

- current_state: Returns the current state description (i.e., all the pieces).

```
current_state(State) :-
      description_pred(Descript),
      findall(Descript,State).
```

- restore_state: Restores a state description.

```
restore_state(State) :-
      description_pred(Descript),
      retractall(Descript),
      asserta_all(State).
```

- restore_if_redo: Used in the *do-move* predicate to restores a state in case the rest of the literals fail for some reason.

```
restore_if_redo(_).
restore_if_redo(State) :-
      restore_state(State),
      !,
      fail.
```

- create_new_state: Creates a copy of the current state description.

```
create_new_state(Pos,State,NPos,NState) :–
    new_pos(NPos),
    replace(Pos,NPos,State,NState),
    asserta_all(NState).
```

PAL is informed of which of the predicates are considered as background definitions by a list of "features". It is from this list that PAL derives its relevant facts (see also appendix H).

```
feature(contents(_,_,square(_,_),_)).
feature(legal_move(_,_,square(_,_),square(_,_),_)).
feature(in_check(_,square(_,_),_,square(_,_),_)).
feature(check_mate(_,square(_,_),_)).
feature(stale(_,_,square(_,_),_)).
feature(stale(_,_),_).
feature(sliding_piece(_,square(_,_),_)).
```

make_move/6, which is used in the definition of *dynamic* patterns, is derived from the *legal_move*/5 predicates (i.e., for each *legal_move* fact derived from an example description, a *make_move* predicate is considered). In the above concepts, *piece_move*/5 defines all the possible moves for the pieces.

In addition, PAL needs to know what is the domain and type of the arguments used in the example description, and which are the predicates that describe the examples.

```
description_pred(contents/4).

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

type_arg(white,side) :– !.
type_arg(black,side) :– !.
type_arg(square(_,_),place) :– !.
type_arg(_,piece).
```

PAL is also given predicates to recognise illegal positions for the example generator (i.e., one King per side, do not allow Pawns in the first or last rank, etc.).

Appendix H

A mini-PAL Prolog program

The purpose of this appendix is to provide a simplified implementation in Prolog of the principal predicates of PAL. In particular, dynamic patterns and disjunctive definitions are not considered. Not all the predicates are given, however, sufficient comments are provided to allow an interested reader to reconstruct them.

There are 5 main predicates in PAL:

- *gopos* and *goneg* are the top predicates which call the example generator and the generalisation method.
- *extract_features* is the predicate that derives all the ground atoms from the background knowledge definitions and the current example description.
- *generalise* constructs a generalisation between two clauses following the constrained *lgg* algorithm described in chapter 5.
- *perturb* generates new examples.
- *reduce_defn* reduces the definition considering the variable connection constraints.

/**/
/* *pal*/0: makes sure that there are no previous examples and perturbation classes, loads an initial example given by the user, constructs the initial perturbation classes, an initial head (*Head*) and the body of the concept clause, and calls the main predicates. *Feats* and *CFeats* represent the body of the clause and its associated labels. */

```
pal :-
        initialise,
        load_example(Head,Feats,CFeats),
        gopos(Head,Feats,CFeats).
```

/**/
/* *gopos*/3: Calls the example generator (*perturb*) to produce a new example, until the perturbation level fails (no more perturbation classes) or stopped by the user. Each new example is shown to the user (*display*) and stored (*store_example*) to avoid producing duplicates. If the example is positive, it call the generalisation method (*generalise*) and sorts the list of perturbation levels (*order_list_of_levels*). If the example is negative it takes the literals that fail with that example (*Fail* and *CFail* in *perturb*) and tries to produce a new example that will succeed in at least one of them (see *goneg*). *stop* accepts a name from the user to the new concept definition and incorporates the definition to the background knowledge.
GH = current head of the hypothesis.
GFts = current body of the hypothesis clause.
CFts = labels of the current body. */

```
gopos(GH,GFts,CFts) :-
       perturb(posit,GH,GFts,CFts,ExH,OldExDes,NewExDes,Fail,CFail),
       !,
       display,
       pos_neg_stop(Type),
       store_example(Type,NewExDes),
       !,
       ( Type = positive,
         generalise(GH,ExH,GFts,CFts,FinH,FinFts,NCFts),
         !,
         order_list_of_levels(NCFts),
         gopos(FinH,FinFts,NCFts)
       ; Type = negative,
         restore_exam_descript(NewExDes,OldExDes),
         !,
         goneg(GH,GFts,CFts,Fail,CFail)
       ; Type = stop,
       reduce_defn(GH,FFts,NH,NFts,CFts,_),
```

```
          stop(NH,NFts)
        ).
gopos(Head,Feats,CFeats) :–
        reduce_defn(Head,Feats,NHead,NFeats,CFeats,_),
        stop(NHead,NFeats).
```

/* *goneg*/5: almost the same as *gopos* except that it uses the list of literals (*Failed*) and their corresponding labels (*CFailed*) that failed with an example to guide the perturbation process. */

```
goneg(GH,GFts,CFts,Failed,CFailed) :–
        perturb(neg,GH,Failed,CFailed,ExH,OldExDes,NewExDes,_,_),
        !,
        display,
        pos_neg_stop(Type),
        store_example(Type,NewExDes),
        !,
        ( Type = positive,
          generalise(GH,ExH,GFts,CFts,FinH,FinFts,NCFts),
          !,
          order_list_of_levels(NCFts),
          gopos(FinH,FinFts,NCFts)
        ; Type = negative,
          restore_exam_descript(NewExDes,OldExDes),
          !,
          goneg(GH,GFts,CFts,Failed,CFailed)
        ; Type = stop,
          stop(GH,GFts)
        ).
goneg(GH,GF,CF,_,_) :– /* in case it is in the last perturb. level */
        gopos(GH,GF,CF).
```

/* *load_example*/3: Accepts an example from the user. It construct a new Head, add the description to the example to the background knowledge, and derives all the ground atoms from it (see *extract_features* below). */

```
load_example(Head,Feats,CFeats) :–
       clear_board,
```

```
      which_example(Head),
      !,
      display,
      extract_features(Feats,CFeats).
```

/**/
/* *extract_features*/2: all the definitions that are considered as relevant background definitions are identified by the predicate *feature* (see also appendix G) and declared as "dynamic". *extract_features* first picks all the background names and then call the predicates to derive facts from them. */

```
extract_features(Fts,CFts) :–
      setof(X,X^feature(X),ListFeats),
      all_features(ListFeats,Fts,CFts).
```

/* *all_features*/3: takes each one of the feature predicates and calls *all*/5. *all*/5 generates all the possible solutions for the predicate (*Feat*) using *mebg* to produce a "copy" (*CFeats*) with the assigned labeled symbols. It returns a list of literals with new label symbols in all the arguments, except in those places where there is an argument of a description of a piece (from which a previously assigned label is used). */

```
all(Feat,Feats,CFeats,Inter,Fin) :–
    copy_variable(Feat,CFeat),
    setof(Feat/CFeat,Feat^CFeat^mebg(Feat,CFeat),All),
    divide_2(All,Feats,CFeats),
    numbervars(CFeats,Inter,Fin),
    !.
all(_,[ ],[ ],F,F).
```

/* *mebg*/2: is similar to EBG. It forms a list of substitutions and calls *mebg*. *mebg* updates the list of substitutions (*new_subst*/3) with the particular arguments of the atoms used to describe example positions that are involved in the derivation process. All the predicates that involve the predicate use to describe example positions (*description_pred*) are declared as *dynamic*. */

```
mebg((A,B),Subs) :–
      !,
      mebg(A,Subs),
```

```
        mebg(B,Subs).
mebg((A;B),Subs) :-
        !,
        ( mebg(A,Subs)
        ; mebg(B,Subs)
        ).
mebg(true,_) :-
        !.
mebg(A,_) :-
        not predicate_property(A,(dynamic)),
        !,
        call(A).
mebg(Pred,Subs) :-
        description_pred(P/A),
        functor(Pred,P,A),
        !,
        current_exam(L),
        member(Pred/Args,L),
        new_subst(Pred,Args,Subs).
mebg(A,Subs) :-
        clause(A,ATail),
        mebg(ATail,Subs).
```

/**/
/* *generalise*/7: makes a generalisation between two clauses. It produces a generalisation, reduces it, and shows it to the user.
GH = Hypothesis head.
ExH = Example Head
GFts = The body of the hypothesis
CFts = The copy of the body (the labels)
FH = New hypothesis head
FFts and *NCFts* = Body and associated labels of the new hypothesis. */

```
generalise(GH,ExH,GFts,CFts,FH,FFts,NCFts) :-
        create_generalisation(GH,ExH,GFts,CFts,NGH,NGFts,CFts1),
        !,
        reduce_defn(NGH,NGFts,FH,FFts,CFts1,NCFts),
        pprint_defn(FH,FFts).
```

/* *create_generalisation*/7: It derives the relevant facts of the new example description, finds the *lgg*, between the head of the hypothesis (*Head*) and the clause constructed for the example (*ExHead*), and uses the list of substitutions generated to guide the *lgg* of the rest of the clauses. */

```
create_generalisation(GH,ExH,GFts,CFts,NGH,NGFts,NCFts) :-
        extract_features(ExFts,CExFts),
        heads_lgg(GH,ExH,NGH,Subst),
        all_feats_lgg(GFts,CFts,ExFts,CExFts,Subst,NGFts,NCFts).
```

/* *all_feats_lgg*/7: first selects compatible literals (same name and predicate symbol) and then calls *all_lgg*. *all_lgg* match compatible literals based on the copy or labels of the clause (*match_up_to_N*), and then calls *clgg* to make the *lgg* between two compatible literals with matching labels. */

/* *match_up_to_N*/2: takes two lists of associated symbols and matchs them iff they differ only in label symbols that are not related to the symbols used in the arguments of the atoms that describe example positions (it assumes that there are no variables as the matching process is between the "copy" or labels of the literals). */

```
match_up_to_N([H|T],[H|R]) :-
        match_up_to_N(T,R).
match_up_to_N([Meta1|T],[Meta2|R]) :-
        metavar(Meta1),
        metavar(Meta2),
        match_up_to_N(T,R).
match_up_to_N([ ],[ ]).
```

/* Identifies labels assigned with 'numbervars' (i.e., not from example descriptions). */

```
metavar(MV) :-
       nonvar(MV),
       MV = '$VAR'(N),
       integer(N).
```

/* *clgg*/6: It returns the *lgg* of two terms, with the number of new substitutions created and a new list of substitutions. */

```
clgg(N,Term1,P2,GP12,Sofar,Subst) :-
      term(Term1),
      !,
      add_to_subs(Term1/P2/GP12,Sofar,Subst,N).
clgg(N,P1,Term2,GP12,Sofar,Subst) :-
      term(Term2),
      !,
      add_to_subs(P1/Term2/GP12,Sofar,Subst,N).
clgg(N,P1,P2,GP12,Sofar,Subst) :-
      functor(P1,F,L),
      functor(GP12,F,L),
      unif(L,P1,P2,GP12,Sofar,Subst,N).
```

/* *unif*/7: fills-in the arguments of the generalisation by going through the arguments of the two predicates to generalise. It returns the number of new substitutions added. */

```
unif(0,_,_,_,L,L,0) :-
     !.
unif(M,P1,P2,GP,Sofar,Subs,N) :-
     arg(M,P1,Arg1),
     arg(M,P2,Arg2),
     clgg(N1,Arg1,Arg2,GArg,Sofar,InterSubs),
     !,
     arg(M,GP,GArg),
     M1 is M - 1,
     unif(M1,P1,P2,GP,InterSubs,Subs,N2),
     !,
     N is N1 + N2.
```

/* *add_to_subs*/4: adds a substitution to a list if it is new and counts it. */

```
add_to_subs(A/B/A,L,L,0) :-
       A == B,
       !.
add_to_subs(New,[ ],[New],1).
add_to_subs(A/B/Old,[X/Y/Old|R],[X/Y/Old|R],0) :-
       A == X,
```

```
        B == Y,
        !.
add_to_subs(Subs,[H|R],[H|T],N) :-
        add_to_subs(Subs,R,T,N).
```

/***/
/* *perturb*/9: creates new examples. It picks the first perturbation class (*PC*) and tries to create new examples. If it fails (cannot create a new example that will fail on some literals) then it changes the perturbation space (*change_levels*/0) and continues. */

```
perturb(Type,H,Fts,CFts,ExH,OldExDes,NewExDes,Fail,CFail) :-
        list_of_levels([PC|_]),
        change(Type,PC,H,Fts,CFts,ExH,OldExGen,NewExGen,Fail,CFail).
perturb(Type,H,Fts,CFts,NH,OldExDes,NewExDes,F,CF) :-
        change_levels,
        !,
        perturb(Type,H,Fts,CFts,NH,OldExDes,NewExDes,F,CF).
```

/* *change*/10: involves the following steps:

1. Extract the current arguments from the current example description which corresponds to the particular perturbation class (*change_aux*).

2. Change the arguments (*change_args*) of those arguments by selecting new values from their domains.

3. Check if the new example corresponds to a legal position (*check_if_legal*).

4. Replace the new values in all the literals where the arguments appear in the body of the hypothesis (*replace_all_args*).

5. Check which of those new literals fail with the new arguments (*check_defn*). It uses the list of literals of the current concept definition (*Fts*), the list of literals in the definition instantiated by the new arguments (*NewFts*) and the labels associated with the literals (*CFts*) to guide the process.

6. Check that the failures are indeed new to avoid producing examples that fail for the same reasons (*check_fails*).

7. Return a head for the new example description (*produce_head*).

Flag = for positive or negative examples
PC = Perturbation Class
H = Current head of the hypothesis
Fts and *CFts* = body and labels of the hypothesis
OldP and *NewP* = old and new description of the board
Fails and *CFails* = literals and labels that failed with the new example. */

```
change(Fl,PC,H,Fts,CFts,NH,OldP,NewP,Fail,CFail) :–
    change_aux(PC,H,Fts,OldP,Args,CArgs),
    change_args(Args,PC,NArgs,NewP),
    check_if_legal,
    replace_all_args(NArgs,CArgs,CFts,Fts,NFts),
    check_defn(Fl,Fts,NewFts,CFts,Fail,CFail),
    check_fails(Fl,Fail,CFail,Args,NArgs,PC),
    !,
    produce_head(Args,NArgs,H,NH,CH).
```

/**/
/* *reduce_defn*/6: Reduces the current concept definition, by removing variable arguments in the head of the clause that are not connected to literals in the body (others than those used to describe examples), and by removing literals with variable arguments which are unconnected to any other variable in the body of the clause. */

```
reduce_defn(H,Fts,NH,NFts,CFts,NCFts) :–
      reduce_head(H,Fts,CFts,NH),
      reduce_body(Fts,NFts,CFts,NCFts),
      !.
```