VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Acquiring tactics from examples

Michael Norrish

Technical Report CS-TR-93/10
November 1993

Abstract

This report describes a new method for knowledge-intensive learning in the domain of two-person turn-taking complete information games. The method acquires general patterns from examples of incorrect play that can be used to avoid similar mistakes in the future. The results of this research are embodied in ELGAR, a system that acquires patterns for the games of noughts and crosses and go-moku.

Publishing Information

This report was submitted in partial fulfilment of the requirements for the degree of Bachelor of Science with Honours.

Author Information

Michael Norrish is a graduate student in the Department of Computer Science at Victoria University of Wellington.

Contents

1 Introduction

This project develops a new and general approach to the classic AI problem of game-playing. Currently, most programs to play games like chess and draughts operate by massive brute-force search through large game trees, using some sort of "mini-max" procedure to pick a best move. However, games such as Go are not well suited to this approach, because the branching factor in the game tree is too high, and substantial look-ahead can be required to achieve even modest performance.

An interesting alternative is to start by looking at how people play games. People lack the cognitive resources for massive game tree searches, but instead rely heavily on their past experience. Psychological evidence suggests that past experience is encoded as patterns that describe previous board positions, coupled to their implications for correct play. What little game tree search that people do perform is closely guided by these patterns. In effect, people trade time spent searching for time spent recognising patterns.

I have developed a new approach to knowledge-intensive learning that acquires patterns from examples in the domain of two-person turn-taking complete information games. To my knowledge, this learning method is more powerful than any existing approach to pattern learning. This research makes four significant contributions to the state of the art in machine learning:

1. A general methodology for the design of languages for representing patterns in game-playing domains. This methodology has been successfully applied to noughts and crosses and go-moku, and should straightforwardly extend to a much wider range of games.

2. A new method for computing the weakest preconditions for descriptions involving high-level predicates, using STRIPS operators to represent actions.

3. A novel method for calculating patterns that entail these weakest preconditions.

4. A new method for using examples to guide the learning process.

The results of this research are embodied in ELGAR,[1] a system written in some 2000 lines of SICSTUS Prolog. ELGAR acquires patterns for the games of noughts and crosses and go-moku.

My research has focussed on two problems: the design of languages for representing patterns, and the design of algorithms for acquiring patterns from examples. I have deliberately de-emphasised the issue of pattern use. As a consequence, ELGAR uses only a rudimentary game driver that provides a minimal setting in which patterns can be acquired and usefully applied.

This research is also restricted to the task of acquiring patterns that describe win/loss situations. Indeed, the only goals of ELGAR's game driver are to win and to avoid losing, so it is natural that the patterns that ELGAR learns address only these goals. In contrast, more sophisticated games such as chess and Go involve a range of subsidiary tactical goals such as control of the centre and depriving groups of eyes.

The remainder of this report is structured as follows. Section 2 describes ELGAR's representation of moves, boards, and patterns. During development of ELGAR, two different pattern languages were employed. The first is very simple and was used at an early stage of ELGAR's development. The second pattern language is much more expressive, and it is this language on which most of the report focusses.

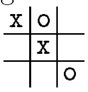
[1] ELGAR stands for *Engine for Learning Game Representations.*

Section 3 presents an example of ELGAR in action. It provides a demonstration of when ELGAR learns patterns, how it uses them once learnt. It turns out that three patterns seem sufficient to cover all of the interesting positions in noughts and crosses.

Section 4 describes ELGAR's learning algorithm. Section 5 describes ELGAR's rudimentary game driver. Section 6 describes the components of ELGAR's implementation. Section 7 discusses previous work in this area, and section 8 presents my conclusions.

2 Representation: boards, moves and patterns

2.1 Board positions and moves

Elgar represents game boards as a set of literals of the form at(*Cell*, *Colour*), where *Cell* is a number referring to one of the locations on the board, numbered left-to-right from the top down and *Colour* is one of x, o, and blank. For example, the position 

is represented as follows:

$$\left[\begin{array}{l}\texttt{at}(1,\texttt{x}),\ \texttt{at}(2,\texttt{o}),\ \texttt{at}(3,\texttt{blank}),\ \texttt{at}(4,\texttt{blank}),\ \texttt{at}(5,\texttt{x}),\\ \texttt{at}(6,\texttt{blank}),\ \texttt{at}(7,\texttt{blank}),\ \texttt{at}(8,\texttt{blank}),\ \texttt{at}(9,\texttt{o})\end{array}\right].$$

Expressions of the form $[c_1, \ldots c_n]$ are Elgar's representation of the conjunctive clause $c_1 \wedge \ldots \wedge c_n$. Throughout this paper, for ease of readability I frequently employ the informal notation of board diagrams, rather than writing board positions in propositional form.

Moves are represented as Strips operators. A Strips operator consists of three components that specify its action when applied to the domain world: its *preconditions*, a *delete list*, and an *add list*. Strips operators take a representation of the initial state of the world as input, and produce a new world state that reflects the impact of the action that the operator represents. In Elgar, states of the world are board positions. An operator can be applied if its preconditions hold in the initial state. The operator's effect on the world is modelled by first removing all known facts from the initial state that match the elements of the delete list, then adding new facts that correspond to the add list.

The move operator for noughts and crosses (and go-moku) is quite simple:

move(*Pos*,*Colour*) :

Preconditions:	at(*Pos*,blank)
Delete list:	at(*Pos*,blank)
Add list:	at(*Pos*,*Colour*)

The straightforward semantics of Strips operators proves useful when Elgar comes to perform pattern learning.

2.1.1 Problems with the Strips model

Admirable though Strips's simplicity may be, it is not a particularly powerful model of action in the world. This has definite repercussions in this game playing application. Though both noughts and crosses and go-moku are modelled without difficulty using the Strips approach, Strips operators are inadequate for representing moves in games such as Othello and go, where the effects of the move are nastily context dependent. For example, in Othello a move can require that an arbitrary number of pieces of the opposite colour be "flipped" depending on the state of the board. These kinds of context dependent effects can not be tractably represented using Strips operators.

An extension to the Strips model, Adl (Pednault, 1986) solves this problem because it permits conditional effects, and preconditions and effects can be universally quantified. Of course, one might expect that reasoning about such operators might be rather more difficult in this model, and such has not been attempted as part of this project. However, where Adl does not use these extensions, it behaves as would Strips, so we can at least be sure that current strategies will not be invalidated by any future shift to an Adl-based implementation.

2.2 Patterns

As Elgar plays games, it acquires patterns that it can use to avoid bad positions that occur during the course of the game. These patterns (with colours reversed) also signal opportunities for Elgar to win. Clearly, the language used to represent patterns is an important component of the whole system. This section of the report describes two pattern languages that have been used in Elgar.

A pattern describes a set of specific board positions. A measure of a pattern language's power is the degree to which it is able to compactly describe many classes of (interesting) positions.

Patterns in Elgar are expressed in a subset of first order predicate logic in disjunctive normal form (DNF). All of the various logical operations that Elgar performs on expressions maintain them in this form. For the DNF expression $(c_{11} \wedge c_{12} \wedge \ldots \wedge c_{1n_1}) \vee (c_{21} \wedge \ldots c_{2n_2}) \vee \ldots \vee (c_{m1} \wedge \ldots c_{mn_m})$, I write $[c_{11}, c_{12}, \ldots, c_{1n_1}]$ `or` $[c_{21}, \ldots c_{2n_2}]$ `or` $\ldots [c_{m1}, \ldots c_{mn_m}]$.

2.2.1 A primitive pattern language

During early experiments with Elgar, I used a very simple pattern language that closely reflects Elgar's internal representation of board positions. In this primitive language, patterns are simply partial specifications of board positions—that is, conjunctions of literals of the form $\texttt{at}(Pos,\ Colour)$ in which some cells of the board are left unspecified. An instance, or concrete board position, matches a pattern if the board position logically entails the pattern, or equivalently, if the set of literals that makes up the pattern is a subset of the literals that make up the instance.

An example of a pattern acquired by Elgar using this vocabulary is the following:

$$[\texttt{at}(1,\texttt{x}), \texttt{at}(2,\texttt{blank}), \texttt{at}(3,\texttt{blank}), \texttt{at}(4,\texttt{blank}), \texttt{at}(5,\texttt{blank}), \texttt{at}(7,\texttt{blank})].$$

This pattern can be depicted using a diagram:

x		
		?
	?	?

. The ?'s represent "don't care" values that permit any value (`x`, `blank` or `o`) for that cell. (The pattern represents a class of positions where `x` can win on `x`'s turn, starting by playing in one of the blank corners.)

This pattern language will be referred to henceforth as the *primitive pattern language*. It is not particularly powerful, but serves as a useful reference point for comparison with Elgar's full pattern language. In particular, while it may be possible to learn a perfect game of noughts and crosses using the primitive pattern language, many improvements are possible. The primitive language has no notion of symmetry and as a result needs to separately learn three-in-a-row threats in all eight possible directions. An obvious fix to this problem is to ensure that the system knows about such concepts as "line," which need not have any orientation information attached to them. Such concepts would allow the system to recognise symmetrical patterns by default. Certainly, this seems to correspond to the way people think about the games they play.

However, it is important that the pattern language of a general purpose system such as Elgar should not be unduly tailored to the needs of a particular game. Optimally, this system should discover which patterns are appropriate and which are not. It is easy, for example, to conceive of a game where lines may occur, but where those lines have no significance whatsoever. This discrimination is the preserve of the learning algorithm, but but it is crucial not to "queer its pitch" by using a pattern language that is perfect for only the task at hand.

2.2.2 ELGAR's full pattern language

ELGAR's second pattern language is much more expressive than the primitive language. It is capable of representing such high-level entities as lines and even "lines with holes in them" (concepts that are clearly useful in noughts and crosses), and can also represent any number of other concepts. To distinguish this pattern language from the primitive pattern language, I refer to it as the *full pattern language*.

In general, the full pattern language allows arbitrary higher level concepts to be defined in terms of a small vocabulary of *elementary predicates*. This turns out to simplify the learning algorithm that ELGAR uses by allowing patterns that use the higher level predicates to be re-expressed in terms of just the elementary predicates.

ELGAR's full pattern language divides its elementary predicates into two classes: those that refer only to properties of board locations, and those that refer to pieces on the board. Propositions built from the former class are *positionally invariant* because no matter the state of the game, they have the same truth value. This proves to be very useful when performing pattern learning.

ELGAR has just one predicate of the latter, *positionally dependent* kind of predicate, `has_elements`:

`has_elements`$(Board, CptPredicate, Els, N)$ is defined to be true if $Board$ has N locations from the set Els which satisfy the predicate $CptPredicate$.

If one treats $Board$ as a set of literals, then `has_elements`$(Board, CptPredicate, Els, N)$ is true iff

$$|\{\mathtt{at}(Cell, Colour) : \mathtt{at}(Cell, Colour) \in B \wedge Cell \in Els \wedge CptPredicate(Colour)\}| \geq N$$

$CptPredicate$ is either the "colour" of a board location (one of `x`, `o` or `blank`), or is an arbitrary predicate over possible colours. Predicates can be written using a λ-notation so that they can be composed on the fly. For example, one predicate that ELGAR uses is $\lambda x.x \neq \mathtt{blank}$. In fact, this predicate occurs so frequently that I will substitute $\neg\mathtt{blank}$ for it henceforth.

An example of `has_elements`'s usage is the following: `has_elements`$(B, \mathtt{x}, Els, 2)$ is true if B is of the form $\begin{array}{c|c|c} \mathtt{x} & & \mathtt{o} \\ \hline & \mathtt{o} & \mathtt{x} \\ \hline & \mathtt{x} & \mathtt{o} \end{array}$ and Els is any set that is a superset of any of $[1, 6]$, $[1, 8]$ or $[6, 8]$.

Two examples of predicates that are of the positionally invariant class are `collinear` and `isct`. Both of these predicates are true of sets of board locations, irrespective of the game pieces that occupy those locations. The proposition `collinear`(A) is true of a set A if A is collinear. For example, `collinear`$([1, 5, 9])$ is true in the noughts and crosses domain. The proposition `isct`(A, B, C) is true of sets A, B and C if $A \cap B = C$.

By combining positionally dependent and positionally invariant predicates, ELGAR can build higher level predicates that pick out useful classes of board positions. For example, ELGAR uses the `collinear` predicate as part of the definition of the following predicate:

`has_line`$(Board, CptPredicate, Length, N, Els)$ is true iff $Board$ contains a line of N pieces matching $CptPredicate$ across the set of positions Els which is of length $Length$.

The predicate `has_line` is implemented in Prolog as follows:

```
has_line(B, X, Len, Num, Els) :-
        length(Els, Len),
        collinear(Els),
        has_elements(B, X, Els, Num).
```

The `has_line` predicate is defined entirely in terms of elementary predicates.

3 Elgar in action

As Elgar plays games, it learns useful patterns from its mistakes. Elgar acquires patterns that encode classes of positions to avoid, and uses these patterns to guide its choice of move. To see how Elgar acquires patterns, it is useful to trace its behaviour during its first few games of noughts and crosses. In these games and all subsequent examples in this report, it is assumed that Elgar is playing as `o`, its opponent is `x`, and the opponent always starts. Also, for purposes of readability, the patterns in this report are depicted in terms of `x`, but they are stored internally in a more general fashion that can also be used to match positions involving `o`'s pieces.

The first game that Elgar plays is as follows:

$\mathtt{x}_1$	$\mathtt{o}_2$	$\mathtt{o}_4$
	$\mathtt{x}_3$	
		$\mathtt{x}_5$

(The subscripts indicate the order in which the moves are played.)

When the game is lost, Elgar attempts to learn from this experience, and consequently acquires the following pattern to avoid in the future:

$$[\mathtt{has\_line}(B,\mathtt{x},3,2,El),\ \mathtt{has\_elements}(B,\mathtt{blank},El,1)] \tag{1}$$

This pattern matches the position in the game above just after move 4, with El bound to the set $[1,5,9]$. However, it also matches a large number of other unfavourable positions such as

x		
		o
x		

where the set El is of the board locations that make up the vertical line descending from the top left. Note that this expression would require 24 patterns to encode in the primitive pattern language (eight lines of three on a 3x3 board, and three possible gaps in each), and this fact is clearly some measure of how much more powerful the full pattern language is.

Elgar stores this information away and the next game begins. The first three moves of the game proceed thus:

$\mathtt{x}_1$	$\mathtt{o}_2$	
$\mathtt{x}_3$		

At this point, Elgar explores every legal move, and discovers that all but the move at position 4 leave the board in a state that matches the pattern it learnt in the game before. If it is to avoid the bad position, it has no choice but to play at 4. The opponent responds at 9, leaving the board looking like the following:

$\mathtt{x}_1$	$\mathtt{o}_2$	
$\mathtt{o}_4$		
$\mathtt{x}_3$		$\mathtt{x}_5$

Now Elgar faces a new and more serious problem. By exhaustive enumeration of its possible moves, Elgar determines that no matter where it plays, the board will still be in a state that matches a position it has already learnt to avoid. The term I use for this sort of quandary is *impasse*. ELGAR resigns whenever it sees an impasse and attempts to learn from the experience. The new pattern it learns is the following:

$$\left[\begin{array}{l}\mathtt{has\_line}(B, \mathtt{x}, 3, 1, El_1),\ \mathtt{has\_elements}(B, \mathtt{blank}, El_1, 2),\\ \mathtt{has\_line}(B, \mathtt{x}, 3, 1, El_2),\ \mathtt{has\_elements}(B, \mathtt{blank}, El_2, 2),\\ \mathtt{isct}(El_1, El_2, It_1),\ \mathtt{length}(It_1, LenIt_1),\ LenIt_1 > 0,\\ \mathtt{has\_elements}(B, \mathtt{blank}, It_1, 1),\\ \mathtt{has\_elements}(B, \neg\mathtt{blank}, It_1, LenIt_1 - 1)\end{array}\right] \qquad (2)$$

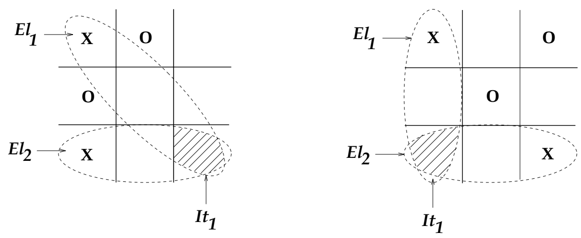
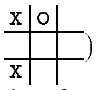


Figure 1: Two board positions matching pattern 2

Just like the first pattern, the new pattern represents a class of positions that ELGAR (playing as o) should attempt to avoid leaving on the board after moving. This pattern matches positions such as those shown in Figure 1. The left-hand board position in Figure 1 is the position from the game that ELGAR just lost. In both board positions in the figure the sets El_1 and El_2 represent two lines, while their intersection It_1 contains 0 non-blanks and 1 blank, as required by the pattern description. This description encodes a general sufficient condition for an impasse: for example, with the move, x could move at location 9 in the board position on the left, or at location 7 in the other and force a win.

ELGAR's job is not yet done however. Before offering to play another game, it first checks that the situation before the most recently analysed impasse is not also an impasse. In effect, ELGAR unwinds forcing sequences in an attempt to discover its earliest possible opportunity to avoid losing. Each stage of a forcing sequence that leads to a loss is effectively an impasse because, knowing where the sequence leads, the victim would prefer not to have to respond to the forcing move. But of course, if the appropriate response is not made, the opponent wins even more directly.

In this case, ELGAR examines the situation back at the time when move 4 was played (i.e ), and realises that this is also an impasse. Whatever move is chosen, the resulting board contains either an incomplete line of three down the left side (matching pattern 1), or the double line threat described by the pattern 2. ELGAR must learn a pattern that will allow it to avoid this impasse as well.

The newly acquired pattern is

$$\left[\begin{array}{l}
\texttt{has_line}(B, \texttt{x}, 3, 1, El_1),\ \texttt{has_line}(B, \texttt{x}, 3, 1, El_2), \\
\texttt{has_elements}(B, \texttt{blank}, El_1, 2),\ \texttt{has_elements}(B, \texttt{blank}, El_2, 2), \\
\texttt{has_elements}(B, \texttt{blank}, Blks, 3),\ \texttt{length}(Blks, 3),\ \texttt{collinear}(Blks), \\
\texttt{isct}(Blks, El_2, It_1),\ \texttt{isct}(El_2 \cup Blks, El_1, It_2), \\
\texttt{length}(It_1, LenIt_1),\ \texttt{length}(It_2, LenIt_2),\ LenIt_1 > 0,\ LenIt_2 > 0, \\
\texttt{has_elements}(B, \texttt{blank}, It_1, 1),\ \texttt{has_elements}(B, \texttt{blank}, It_2, 1), \\
\texttt{has_elements}(B, \neg\texttt{blank}, It_1, LenIt_1 - 1), \\
\texttt{has_elements}(B, \neg\texttt{blank}, It_2, LenIt_2 - 1)
\end{array}\right] \tag{3}$$

This pattern matches quite a large number of board positions, among them that of Figure 2, which is one of the positions in the game under analysis that ELGAR would have considered

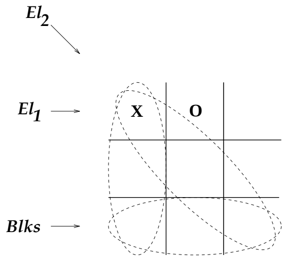


Figure 2: A board position matching pattern 3

after the very first move. As depicted in the figure, the pattern matches the board position with the following assignment to the variables: $\{El_1 = [1, 5, 9], It_1 = [7], El_2 = [1, 4, 7], It_2 = [1, 9], Blks = [7, 8, 9]\}$. It is not difficult to confirm that this assignment does satisfy all of the conjuncts of the pattern above. The next section of the report explains just how ELGAR learns patterns from examples.

4 Pattern learning

This section of the report explains Elgar's learning algorithm. Once again, I use examples from noughts and crosses; it is important to emphasise, however, that the algorithm applies equally well to go-moku and should also extend to other games.

Broadly speaking, Elgar's learning task is to find general descriptions of board positions that must have preceded either the loss or impasse in which the game-playing component now finds itself. If the board is left in one of these positions after Elgar's move, the opponent can force a win. Similarly, if the opponent leaves the board in such a position (with colours reversed), Elgar can force a win.

There are two phases to pattern learning in Elgar. In Phase One, Elgar generalises an unfavourable position S to produce a general description G expressed in Elgar's pattern language. G describes a class of board positions all of which suffer from the same problem that S does.

In Phase Two, Elgar then calculates a pattern P such that

$$P \models wp(\texttt{move}(M, \texttt{x}), G)$$

That is, P entails the weakest precondition on a board such that the move by the opponent is guaranteed to establish G. P can then be remembered as a pattern describing a class of board positions to avoid leaving on the board immediately prior to the opponent's move. P is chosen to be as weak as possible. With colours reversed, P also signals an opportunity to force a win. I call this second phase of the learning algorithm *regression*, and I say that P is produced by *regressing* G *over* Op.

The remainder of this section first elaborates the example from section 3 and then discusses the learning process in terms of the two stages identified above.

4.1 The output of generalisation and regression

The first game played by Elgar in section 3 led to a straightforward loss caused by the opponent getting three x's in a row. The final board position was

x	o	o
	x	
		x

. Generalisation takes this board position and produces a pattern [$\texttt{has\_line}(B, \texttt{x}, 3, 3, El)$] as its description of the loss. The domain information for noughts and crosses includes

```
win(X,B) :- has_line(B,X,3,3,_).
```

and this pattern is derived from the body of this clause.

Next, in Phase Two, Elgar must regress this pattern over the move operator $\texttt{move}(M, \texttt{x})$ in order to learn a pattern to avoid. This produces the following expression:

$$[\texttt{has\_line}(B, \texttt{x}, 3, 2, El),\ \texttt{has\_elements}(B, \texttt{blank}, El, 1)]$$

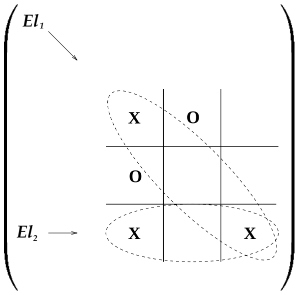
This pattern corresponds to the notion informally expressed as "a line with two x's and a blank," and is clearly something that o (Elgar) should avoid leaving on the board at the beginning of x's move.

The second game played in section 3 leads Elgar to learn two different impasse patterns. First Elgar generalises the impasse position,

x	o	
o		
x		x

, producing the following pattern:

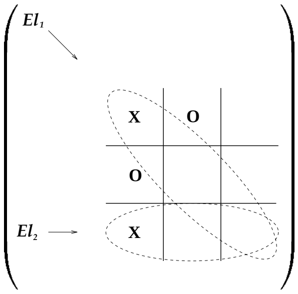
$$\left[\begin{array}{l}
\texttt{has_line}(B, \texttt{x}, 3, 2, El_1),\ \texttt{has_elements}(B, \texttt{blank}, El_1, 1), \\
\texttt{has_line}(B, \texttt{x}, 3, 2, El_2),\ \texttt{has_elements}(B, \texttt{blank}, El_2, 1), \\
\texttt{isct}(El_1, El_2, It_1),\ \texttt{length}(It_1, LenIt_1), \\
\texttt{has_elements}(B, \neg\texttt{blank}, It_1, LenIt_1)
\end{array}\right]$$

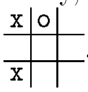


The diagram to the right indicates how this pattern matches the position that occurred in the actual game.

Regressing this pattern over the opponent's move produces another pattern:

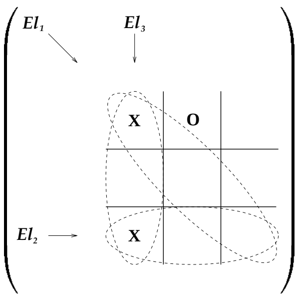
$$\left[\begin{array}{l}
\texttt{has_line}(B, \texttt{x}, 3, 1, El_1),\ \texttt{has_elements}(B, \texttt{blank}, El_1, 2), \\
\texttt{has_line}(B, \texttt{x}, 3, 1, El_2),\ \texttt{has_elements}(B, \texttt{blank}, El_2, 2), \\
\texttt{isct}(El_1, El_2, It_1),\ \texttt{length}(It_1, LenIt_1),\ LenIt_1 > 0, \\
\texttt{has_elements}(B, \neg\texttt{blank}, It_1, LenIt_1 - 1), \\
\texttt{has_elements}(B, \texttt{blank}, It_1, 1)
\end{array}\right]$$



Finally, ELGAR has the task of learning a pattern that will enable it to avoid positions like 

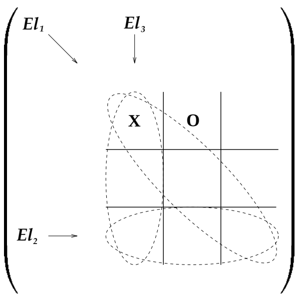
. Generalisation of this impasse position results in the following pattern:

$$\left[\begin{array}{l}
\texttt{has_line}(B, \texttt{x}, 3, 1, El_1),\ \texttt{has_elements}(B, \texttt{blank}, El_1, 2), \\
\texttt{has_line}(B, \texttt{x}, 3, 1, El_2),\ \texttt{has_elements}(B, \texttt{blank}, El_2, 2), \\
\texttt{has_line}(B, \texttt{x}, 3, 2, El_3),\ \texttt{has_elements}(B, \texttt{blank}, El_3, 1), \\
\texttt{isct}(El_1, El_2, It_1),\ \texttt{length}(It_1, LenIt_1),\ LenIt_1 > 0 \\
\texttt{has_elements}(B, \neg\texttt{blank}, It_1, LenIt_1 - 1), \\
\texttt{has_elements}(B, \texttt{blank}, It_1, 1), \\
\texttt{isct}(El_1 \cup El_2, El_3, It_2),\ \texttt{length}(It_2, LenIt_2), \\
\texttt{has_elements}(B, \neg\texttt{blank}, It_2, LenIt_2)
\end{array}\right]$$



and when this pattern is regressed over x's move, the result is as follows:

$$\left[\begin{array}{l}
\texttt{has_line}(B, \texttt{x}, 3, 1, El_1),\ \texttt{has_elements}(B, \texttt{blank}, El_1, 2), \\
\texttt{has_line}(B, \texttt{x}, 3, 1, El_2),\ \texttt{has_elements}(B, \texttt{blank}, El_2, 2), \\
\texttt{has_elements}(B, \texttt{blank}, Blks, 3),\ \texttt{length}(Blks, 3), \\
\texttt{collinear}(Blks),\ \texttt{isct}(Blanks, El_2, It_1), \\
\texttt{isct}(El_2 \cup Blanks, El_1, It_2),\ \texttt{length}(It_1, LenIt_1), \\
\texttt{length}(It_2, LenIt_2),\ LenIt_1 > 0,\ LenIt_2 > 0, \\
\texttt{has_elements}(B, \texttt{blank}, It_1, 1), \\
\texttt{has_elements}(B, \texttt{blank}, It_2, 1), \\
\texttt{has_elements}(B, \neg\texttt{blank}, It_1, LenIt_1 - 1), \\
\texttt{has_elements}(B, \neg\texttt{blank}, It_2, LenIt_2 - 1)
\end{array}\right]$$



In the remainder of this section I explain how generalisation and regression are implemented.

4.2 Generalisation

ELGAR uses the technique of *explanation-based learning* (EBL) in combination with regression to generalise position descriptions. EBL is admirably suited to the task of pruning away what is not relevant in a board position, and focussing one's attention on just what caused a given state to come about. The EBL algorithm examines a proof that the example game supports a *target concept* (here *loss* or *impasse*) and collects up the leaves of the proof tree at a specified level (the level of so-called "operational" predicates). At the same time, a "general" tree is constructed in parallel. The leaves of this tree are a collection of general clauses which, if true, guarantee the existence of an instance of the target concept. EBL is discussed in more detail in section 7. The EBL algorithm that ELGAR uses appears in appendix A.

If pieces on the board do not contribute to the fact of the win (for example), then they are not referred to in the proof tree, so the general description that EBL produces of the position does not refer to them either.

4.2.1 Generalising losses

In both noughts and crosses and go-moku there is only one way in which to lose the game, and the proof tree constructed to support this conclusion is very simple. Part of ELGAR's domain information about noughts and crosses is the following two clauses:

```
loss(X,B) :- opponent(X,Y), win(Y,B).
win(X,B) :- has_line(B,X,3,3,_).
```

Any proof tree of a loss necessarily comes to rest on an instantiation of the body of the `win` clause. The generalisation is straightforward: a lost board is one that has a line of three on it. Generalising impasses, however, is not so straightforward, as I now discuss.

4.2.2 Generalising impasses

Recall that an *impasse* is a situation in which no matter how one plays, the opponent can force a win. Impasses are more difficult to generalise than losses. For example, consider the following board position, in which it is `o`'s turn to move:

x	o	x
x		o

. No matter how `o` plays, the board will be left in a position from which `x` can play and win. Therefore this position is an impasse. Logically, an impasse is a situation B in which

$$(\forall M, B')(apply\_operator(B, \mathtt{move}(M, \mathtt{o}), B') \Rightarrow \mathtt{badpos}(\mathtt{o}, B'))$$

where $apply\_operator(B, Op, B')$ is true if the result of applying operator Op to board position B is B', and where `badpos`($Player$, $Board$) means that $Board$ matches one of ELGAR's existing patterns and therefore is in a state from which $Player$'s opponent can win, if given the move. How then does ELGAR generalise situations it recognises as impasses? Following Minton's lead in PRODIGY (Minton, 1988), ELGAR expands the right hand side of a universally quantified implication into a disjunction of generalised descriptions:

$$(\forall M', B')(apply\_operator(B, \mathtt{move}(M, \mathtt{o}), B') \Rightarrow \bigvee_i genbadposition_i)$$

Each i corresponds to a binding to the free variables in the left-hand side of the expression (i.e. M' and B'). Each $genbadposition_i$ is the result of applying EBL to the result of instantiating the right-hand side of the implication as specified by the bindings to the free variables. In this example, $B = \begin{array}{c|c|c} \mathtt{x} & \mathtt{o} & \mathtt{x} \\ \hline & & \\ \hline \mathtt{x} & & \mathtt{o} \end{array}$, and there are four possible sets of bindings to the free variables:

$$\underset{M=4}{B' = \begin{array}{c|c|c} \mathtt{x} & \mathtt{o} & \mathtt{x} \\ \hline \mathtt{o} & & \\ \hline \mathtt{x} & & \mathtt{o} \end{array}} \; ; \; \underset{M=5}{B' = \begin{array}{c|c|c} \mathtt{x} & \mathtt{o} & \mathtt{x} \\ \hline & \mathtt{o} & \\ \hline \mathtt{x} & & \mathtt{o} \end{array}} \; ; \; \underset{M=6}{B' = \begin{array}{c|c|c} \mathtt{x} & \mathtt{o} & \mathtt{x} \\ \hline & & \mathtt{o} \\ \hline \mathtt{x} & & \mathtt{o} \end{array}} \; ; \; \underset{M=8}{B' = \begin{array}{c|c|c} \mathtt{x} & \mathtt{o} & \mathtt{x} \\ \hline & & \\ \hline \mathtt{x} & \mathtt{o} & \mathtt{o} \end{array}}$$

Applying EBL, ELGAR obtains:

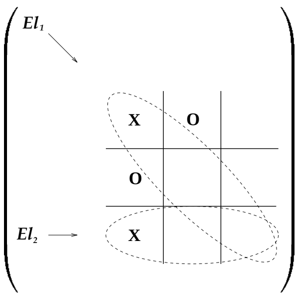
$$\begin{array}{l} apply\_operator(B, \mathtt{move}(M, \mathtt{o}), B') \Rightarrow \\ \qquad [\mathtt{has\_line}(B'_1, \mathtt{x}, 3, 2, El_1),\ \mathtt{has\_elements}(B', \mathtt{blank}, El_1, 1)] \\ \qquad\qquad\qquad\qquad \mathtt{or} \\ \qquad [\mathtt{has\_line}(B'_1, \mathtt{x}, 3, 2, El_2),\ \mathtt{has\_elements}(B', \mathtt{blank}, El_2, 1)] \\ \qquad\qquad\qquad\qquad \mathtt{or} \\ \qquad [\mathtt{has\_line}(B'_1, \mathtt{x}, 3, 2, El_3),\ \mathtt{has\_elements}(B', \mathtt{blank}, El_3, 1)] \\ \qquad\qquad\qquad\qquad \mathtt{or} \\ \qquad [\mathtt{has\_line}(B'_1, \mathtt{x}, 3, 2, El_4),\ \mathtt{has\_elements}(B', \mathtt{blank}, El_4, 1)] \end{array}$$

The patterns on the right side of the implication operator are generalised descriptions of the possible outcomes. ELGAR then simplifies this expression, and obtains the following:

$$\begin{array}{l} apply\_operator(B, \mathtt{move}(M, \mathtt{o}), B') \Rightarrow \\ \qquad [\mathtt{has\_line}(B', \mathtt{x}, 3, 2, El), \mathtt{has\_elements}(B', \mathtt{blank}, El, 1)] \end{array}$$

This is as far as EBL can take ELGAR. EBL provides a general description of the board that results *after* the move by `o`. However, to produce a pattern that ELGAR can usefully employ, it is necessary to compute a description of the position B that obtained one move earlier, just after `x`'s move. To accomplish this, ELGAR regresses its general description of B' over the operator $\mathtt{move}(M, \mathtt{o})$. M is free in this expression because ELGAR must compute a weakest precondition for B', irrespective of its choice of move. In summary, when generalising an impasse, EBL only computes a postcondition that is used as input to regression. When the pattern above is so regressed, ELGAR produces

$$\left[\begin{array}{l} \mathtt{has\_line}(B, \mathtt{x}, 3, 2, El_1),\ \mathtt{has\_elements}(B, \mathtt{blank}, El_1, 1), \\ \mathtt{has\_line}(B, \mathtt{x}, 3, 2, El_2),\ \mathtt{has\_elements}(B, \mathtt{blank}, El_2, 1), \\ \mathtt{isct}(El_1, El_2, It_1),\ \mathtt{length}(It_1, LenIt_1), \\ \mathtt{has\_elements}(B, \neg\mathtt{blank}, It_1, LenIt_1) \end{array} \right]$$



which is a general description of the impasse, as required.

4.3 Regression

This section explains how ELGAR regresses descriptions in the full pattern language. Regression is used in two places in ELGAR: as part of the generalisation process (as discussed above) and in Phase Two. The regression task is depicted in Figure 3.

the regression task: Given a description B of a board position in disjunctive normal form and an arbitrary STRIPS operator Op, find a pattern P such that application of the operator to a position matching P is guaranteed to produce a position matching B. Formally, find a description P such that $P \models wp(Op, B)$. P is chosen to be as weak as possible.

Figure 3: Regression in ELGAR

4.3.1 Computation of weakest preconditions

The basic representation of both the game boards and the move operator in ELGAR is in terms of the literals of the form $\texttt{at}(Pos, Colour)$: sets of them to describe board positions and the preconditions, add and delete lists of the `move` operator. In contrast, the full pattern language uses higher-level constructs such as the `has_elements` and `has_line` predicates to describe board positions. As a consequence, the standard algorithm for regressing over STRIPS operators (Nilsson, 1980) is inadequate, so I had to design my own.

The standard algorithm only works if the patterns used are expressed in the same vocabulary as the operators. Here, our patterns are two levels of abstractions removed from the level of the `at` literal: high level concepts such as `has_line` are defined in terms of elementary predicates such as `has_elements`, which are in turn abstractions of classes of board positions that can be defined using `at` literals. This section describes how ELGAR regresses such high-level patterns.

As the first step of the calculation of weakest preconditions, ELGAR expands high level descriptions such as `has_line(...)` into their constituent elementary predicates. After the calculation of weakest preconditions, ELGAR attempts to reform similar literals from the lower level literals present in the result. In other words, if there is a `has_line` literal in the expression whose weakest precondition is to be calculated, then ELGAR tries to infer one in the result as well. This decomposition reduces the problem of regressing arbitrary higher level predicates to the problem of regressing just elementary predicates.

When ELGAR comes to implement regression of elementary predicates in the full pattern language, recall that the pattern language is structured in such a way that only one predicate (`has_elements`) refers to pieces on the board; others refer only to positionally invariant properties of sets of board locations. This property greatly simplifies regression, as predicates other than `has_elements` regress over the `move` operator unchanged.

When regressing clauses involving `has_elements` literals, however, it is necessary to analyse the wp-semantics of STRIPS operators, as the action of a move operator may well affect the `has_elements` literals. Given a `move` operator with precondition pre, delete list del and add list add, ELGAR's task is to compute

$$\begin{aligned} & pre \wedge wp(\texttt{del\_list}(del); \texttt{add\_list}(add), P) \\ \equiv \; & pre \wedge wp(\texttt{del\_list}(del), wp(\texttt{add\_list}(add), P)) \end{aligned}$$

where `add_list` and `del_list` are the operations of adding and deleting facts from the global state. The first line of the expression above is in close agreement with the usual understanding of the semantics of STRIPS operators: first the precondition must hold, then the facts listed in the delete list are deleted, and then those in the add list are added. (The ; is the "sequential composition" operator.)

The semantics of the `add_list` and `del_list` operations can be defined as follows:

$$wp(\texttt{del\_list}(\langle del_1, \ldots, del_n \rangle), P) = wp(\texttt{del}(del_1); \ldots; \texttt{del}(del_n), P)$$
$$wp(\texttt{add\_list}(\langle add_1, \ldots, add_m \rangle), P) = wp(\texttt{add}(add_1); \ldots; \texttt{add}(add_m), P)$$

where the weakest preconditions of `add` and `del` operations are defined operationally by the implementations of `regress_overadd` and `regress_overdel` in Figures 4 and 5.

The first argument to the `regress_overadd` predicate is the result of regression, the second is the operator, and the third is the original `has_elements` literal. The disjunction comes about because there is no way of telling whether the move (at `Pos`) was played in the set `Els` or not. (The Prolog code uses `member(M,Els)` to stand for $M \in Els$.) If it was, and the piece played is of the same type as the `Type` of the `has_elements` literal, then it must be the case that in the previous position there was one less piece of that colour present (hence the `N-1`). The `colour_match` predicate confirms whether or not a given piece type matches the component predicate of the `has_elements` term. Recall that a predicate is either the contents of a board location (`x`, `o` or `blank`) or an arbitrary predicate which may be a λ-expression. For example, `colour_match`($\lambda x.x \neq$ `blank`, `x`) is $true$, but `colour_match`(`x`,`o`) is $false$.

```
regress_overadd(Clause,
                at(M, Colour),
                has_elements(B, Type, Els, N)) :-
   colour_match(Type, Colour),
   Clause = [has_elements(B, Type, Els, N-1), N>0, member(M, Els)]
                         or
            [has_elements(B, Type, Els, N), not(member(M, Els))].

regress_overadd([A], _, A).
```

Figure 4: Regression over an add condition

The code for `regress_overdel` is very similar to that of `regress_overadd`. If the move was in the area covered by the element-set of the `has_elements` term and the move deleted an `at` term of the appropriate colour, then it must be the case that before the move there was one more such "piece" than there is now.

```
regress_overdel(Clause,
                at(M, Colour),
                has_elements(B, Type, Els, N)) :-
   colour_match(Type, Colour),
   Clause = [has_elements(B, Type, Els, N), not(member(M, Els)]
                         or
            [has_elements(B, Type, Els, N+1), member(M, Els)].

regress_overdel([A], _, A).
```

Figure 5: Regression over a delete condition

The final requirement is to handle the operator's preconditions. This is quite straightforward. The list of literals that make up the operator's preconditions is simply conjoined with every clause of the DNF expression. In order to have applied an operator to a board position, its precondition must have been satisfied.

Currently, ELGAR cannot derive the rules in Figures 4 and 5 itself. However, I believe that all the necessary information is available to it. In particular, the Horn clauses specifying the implementation of the pattern matching algorithm provide information on the relation between `has_elements` and `at` literals.

ELGAR also makes use of the following structural properties of wp:

- $wp(Op, P \wedge Q) = wp(Op, P) \wedge wp(Op, Q)$
- $wp(Op, P \vee Q) = wp(Op, P) \vee wp(Op, Q)$

From (Dijkstra, 1975) I know that wp distributes over disjunction because the `move` operator (the only one used) is fully deterministic. These properties allow ELGAR to regress expressions a literal at a time.

Finally, note also that at this stage ELGAR can not reason about regression if any of the three operator components (i.e. precondition, add list, or delete list) are anything other than `at` literals.

4.3.2 Using the example as a guide

Implementation of code to calculate weakest preconditions as in the section above is not particularly difficult. However, on its own this is not enough. If left as presented so far, ELGAR will produce weakest precondition expressions that are exponentially large in the number of sets in the patterns being regressed. The problem is best illustrated by an example.

Consider the following game:

$\mathtt{x}_1$	$\mathtt{o}_2$	
$\mathtt{x}_3$		

When ELGAR concludes that this board position is an impasse, generalisation reveals that it is an impasse because it matches the following pattern:

$$\left[\begin{array}{l} \mathtt{has\_line}(B, \mathtt{x}, 3, 1, El_1),\ \mathtt{has\_elements}(B, \mathtt{blank}, El_1, 2), \\ \mathtt{has\_line}(B, \mathtt{x}, 3, 1, El_2),\ \mathtt{has\_elements}(B, \mathtt{blank}, El_2, 2), \\ \mathtt{has\_line}(B, \mathtt{x}, 3, 2, El_3),\ \mathtt{has\_elements}(B, \mathtt{blank}, El_3, 1), \\ \mathtt{isct}(El_1, El_2, It_1),\ \mathtt{length}(It_1, LenIt_1),\ LenIt_1 > 0, \\ \mathtt{has\_elements}(B, \neg\mathtt{blank}, It_1, LenIt_1 - 1), \\ \mathtt{has\_elements}(B, \mathtt{blank}, It_1, 1), \\ \mathtt{isct}(El_1 \cup El_2, El_3, It_2),\ \mathtt{length}(It_2, LenIt_2), \\ \mathtt{has\_elements}(B, \neg\mathtt{blank}, It_2, LenIt_2) \end{array}\right]$$

with $El_1 = [1, 5, 9]$, $El_2 = [7, 8, 9]$, $El_3 = [1, 4, 7]$, $It_1 = [9]$ and $It_2 = [1, 7]$ (among other bindings). This pattern is the input to Phase Two of learning, when regression over `x`'s move calculates a description of the preceding board position.

There are 9 `has_elements` literals in the description above (remembering that each `has_line` will expand out to one `has_elements`), so unconstrained calculation of weakest preconditions will result in $2^9 = 512$ clauses in the resulting DNF description. To see this, consider Figures 4 and 5 above. Every time a `has_elements` literal matches a literal of the form `at(M, Colour)`, a disjunction results, corresponding to two possibilities: either the move `M` is in the set `Els`, or it isn't. Because ELGAR maintains all of its expressions in DNF, the size of the resulting expression will double each time this occurs.

It turns out that the 512 clauses of the DNF expression can be usefully categorised into three distinct classes:

Contradictions: Most of the clauses are contradictions. Some clauses contain literals representing $M \in S$ and $M \notin S$. Others are more indirect. For example, some require that $[M \in El_1, M \in El_2, M \notin It_1]$, where recall that $It_1 = El_1 \cap El_2$.

Previously identified bad positions: Yet other clauses describe classes of positions that ELGAR would already recognise as a bad position. In other words, a move prior to the one under consideration brought about a bad position. This is impossible however, as such a situation could only have arisen if an impasse existed earlier: if there wasn't, ELGAR would have moved so as to eliminate the threat. But ELGAR always resigns upon seeing an impasse, so such a situation could not have come about. In the example above, these clauses include memberships of the form $[M \notin El_3]$. If the move isn't in El_3, then the board position before `x`'s move would have already had the "line of two x's with a hole" pattern on it (as described by [`has_line`$(B, \mathtt{x}, 3, 2, El_3)$,`has_elements`$(B, \mathtt{blank}, El_3, 1)$]). Clauses of this kind can also be discarded.

Useful clauses: All other clauses represent possible, non-contradictory situations. In the example above, there are just three of these. Consider: we know that information about intersection memberships can be derived from what we know of memberships in the sets El_1, El_2 and El_3. There are therefore eight possible clauses produced by varying the membership of these. We can eliminate half immediately because the analysis of the preceding paragraph tells us that $M \in El_3$. Similarly, M must be in one of El_1 or El_2 because it would otherwise mean that the double threat that those two sets represent would have been left on the board before the move.

Of the three, two are in fact symmetrical. ($[M \in El_1, M \notin El_2]$ is equivalent to $[M \notin El_1, M \in El_2]$ because El_1 and El_2 share exactly the same properties.) This reduces the number of clauses of interest in this example to just two.

One fix for the problem of having to analyse so many clauses might be to build in sophisticated machinery for reasoning about sets that could allow ELGAR to only consider 8 instead of 512 clauses. However this would still represent an exponential blowup—no longer in the number of sets in the pattern, but in the number of lines. Although this number is smaller (3 in the example instead of 9), this still leads to intractability for large patterns.

Consequently, ELGAR takes a different approach. When regressing over the opponent's move (i.e. the move that brought about the loss or impasse), ELGAR uses the example of the game itself to closely guide its calculation of weakest preconditions. In fact, in the above example, rather than analysing all 512 clauses, ELGAR generates only one. ELGAR picks the one clause that is consistent with the most recent move by `x` and the match between the game position and the pattern to be regressed.

More specifically, assuming that the `move` operator to be regressed over is `move`$(M, \mathtt{x})$, ELGAR only generates the single clause corresponding to the situation where $[M \notin El_1,\ M \in$

El_2, $M \in El_3$]. ELGAR generates these set membership constraints by examining the status of board location 7 (where x last moved), with respect to the sets $El_1 = [1,5,9], El_2 = [7,8,9], El_3 = [1,4,7]$ determined by matching the pattern to the board position. ELGAR packages up the set membership constraints into what I call an "advice clause," which is then passed to the algorithm that computes weakest preconditions. The advice clause allows ELGAR to select particular disjuncts when working with the `regress_overadd` and `regress_overdel` procedures, and thus avoid the exponential blowup in the number of clauses in the DNF representation of the weakest preconditions. When given this advice, regression produces the pattern one expects.

This technique necessarily produces a description of a class of board positions that ELGAR should avoid, but it is narrower than it needs to be. Using the advice above, ELGAR fails to consider the remaining possibility that $[M \in El_1, M \in El_2, M \in El_3]$.

What would happen with this different piece of advice? Regression returns the following expression:

$$\left[\begin{array}{l}
\texttt{collinear}(El_1),\ \texttt{length}(El_1,3), \texttt{has_elements}(B,\texttt{blank},El_1,3),\\
\texttt{collinear}(El_2),\ \texttt{length}(El_2,3), \texttt{has_elements}(B,\texttt{blank},El_2,3),\\
\texttt{has_line}(B,\texttt{x},3,1,El_3),\ \texttt{has_elements}(B,\texttt{blank},El_3,2),\\
\texttt{isct}(El_1,El_2,It_1),\ \texttt{length}(It_1,LenIt_1),\ LenIt_1 - 1 > 0\\
\texttt{has_elements}(B,\neg\texttt{blank},It_1,LenIt_1 - 2),\\
\texttt{has_elements}(B,\texttt{blank},It_1,2),\\
\texttt{isct}(El_1 \cup El_2,El_3,It_2),\ \texttt{length}(It_2,LenIt_2),\ LenIt_2 > 0,\\
\texttt{has_elements}(B,\neg\texttt{blank},It_2,LenIt_2 - 1),\\
\texttt{has_elements}(B,\texttt{blank},It_2,1)
\end{array}\right]$$

Interestingly, this pattern describes a situation that can never occur on a noughts and crosses board. The crucial factor is that the length of the intersection It_1 of the two collinear sets El_1 and El_2 must be at least two. This is impossible to achieve with lines of length three on a 3×3 board unless the two lines are actually the same. But this cannot be the case above because of $\texttt{has\_elements}(B,\neg\texttt{blank},It_1,LenIt_1 - 2)$. If two lines of three blanks duplicate each other, then the length of their intersection will be 3, but the term just quoted would then require that one of the locations be non-blank.

This is not to say that the pattern above is completely useless. If ELGAR were playing on a bigger board (but still had to get just three in a row), this pattern would match a pattern such as the one below in Figure 6. This is a bad position to leave x in because,

?	x	?	?
?		?	?

Figure 6: A position to avoid...

with the move, x can play immediately below its piece already on the board, and then play immediately to the right leaving o with the task of defending both open ends of the two in a row.

This advice strategy for guiding computation of weakest preconditions is not confined to only the situation when regressing over the opponent's move. When regressing over an arbitrary move by o (as is done to generalise impasses), ELGAR can generate one advice clause for each of the specific moves that can be made in the specific board position. In this way, the number of disjunctions will be equal to the number of possible moves (which in bigger games will not necessarily be small), but will not grow exponentially as the complexity of the board patterns increases.

4.3.3 Description strengthening

Strengthening is the part of regression that turns expressions such as

$$\left[\begin{array}{l}\mathtt{has\_line}(B,\mathtt{x},3,2,El),\ \neg\mathtt{member}(M,El),\\ \mathtt{has\_elements}(B,\mathtt{blank},El,1)\end{array}\right]$$

into the more useful

$$\left[\begin{array}{l}\mathtt{has\_line}(B,\mathtt{x},3,2,El_1),\ \mathtt{has\_elements}(B,\mathtt{blank},El_1,1),\\ \mathtt{has\_line}(B,\mathtt{x},3,2,El_2),\ \mathtt{has\_elements}(B,\mathtt{blank},El_2,1),\\ \mathtt{isct}(El_1,El_2,It_1),\ \mathtt{length}(It_1,LenIt_1),\\ \mathtt{has\_elements}(B,\neg\mathtt{blank},It_1,LenIt_1)\end{array}\right]$$

where this process has eliminated the references to M in the original description. This is a necessary step because the reference to M in the pattern is effectively a constraint on the move that o (ELGAR) is to make. This is not a useful description of a board position to avoid. In removing the reference to M, ELGAR computes instead a stronger pattern described entirely in terms of board elements that entails the weakest preconditions.

These strengthening techniques are chosen to be as weak as practicable. If ELGAR were given licence to strengthen to an extreme, it could simply replace all patterns with *false* which matches the empty set of board positions. Although it is easy to dismiss *false* as an option, it is not clear that the strategies above produce the weakest possible strengthening. At this stage, the strategies ELGAR uses are essentially justified by pragmatics and intuition. I make no attempt to prove that they do in fact produce the weakest possible strengthening.

Three strengthening techniques are currently used by ELGAR:

1. Replace $[P, \neg\mathtt{member}(M,S)]$ with $([P,\neg\mathtt{member}(M,S)]\mathtt{or}[Q,\neg\mathtt{member}(M,T)])$ where Q is a copy of P with unbound variables created afresh. ELGAR expects to then be able to apply strategies 2 and 3 below. Intuitively, this kind of strengthening is useful when the situation is that of a "fork," when two bad positions with identical pattern descriptions exist on the board, but any given move can only eliminate one of them.

2. Replace $([P,\ \neg\mathtt{member}(M,S)]\ \mathtt{or}\ [Q,\ \neg\mathtt{member}(M,T)])$ with $[P,\ Q,\ \mathtt{isct}(S,T,U),\ \neg\mathtt{member}(M,U)]$.

3. Replace $[P, \neg\mathtt{member}(M,S)]$ with $[P,P']$, where P' is a has_elements literal constructed to ensure that the preconditions for the relevant move operator will not be satisfied in set S—that is, so that $P' \models \neg(pre(\mathtt{move}(M,Colour)) \wedge M \in S)$.

It is easy to show that techniques 1 and 3 always rewrite old expressions to new ones that imply the old. To see that this is true of strategy 2 as well, note that the result of that strategy can be written as $p \wedge q \wedge x \notin (S \cap T)$. We have that

$$\begin{array}{ll} & p \wedge q \wedge x \notin (S \cap T) \\ \Rightarrow & (p \vee q) \wedge (p \vee x \notin T) \wedge (q \vee x \notin S) \wedge (x \notin S \vee x \notin T) \\ \equiv & (p \wedge x \notin S) \vee (q \wedge x \notin T) \end{array}$$

which can be rewritten as $[P, \neg\texttt{member}(M, S)]$ `or` $[Q, \neg\texttt{member}(M, T)]$, as required.

The way in which these strategies apply to the example above is as follows:

$$[\texttt{has\_line}(B, \texttt{x}, 3, 2, El_1),\ \texttt{has\_elements}(B, \texttt{blank}, El_1, 1),\ \neg\texttt{member}(M, El_1)]$$

$$\rightarrow \left[\begin{array}{l} \texttt{has\_line}(B, \texttt{x}, 3, 2, El_1), \\ \texttt{has\_elements}(B, \texttt{blank}, El_1, 1), \\ \neg\texttt{member}(M, El_1) \end{array}\right] \texttt{ or } \left[\begin{array}{l} \texttt{has\_line}(B, \texttt{x}, 3, 2, El_2), \\ \texttt{has\_elements}(B, \texttt{blank}, El_2, 1), \\ \neg\texttt{member}(M, El_2) \end{array}\right] \quad \text{by 1}$$

$$\rightarrow \left[\begin{array}{l} \texttt{has\_line}(B, \texttt{x}, 3, 2, El_1),\ \texttt{has\_elements}(B, \texttt{blank}, El_1, 1), \\ \texttt{has\_line}(B, \texttt{x}, 3, 2, El_2),\ \texttt{has\_elements}(B, \texttt{blank}, El_2, 1), \\ \texttt{isct}(El_1, El_2, It_1),\ \neg\texttt{member}(M, It_1) \end{array}\right] \quad \text{by 2}$$

$$\rightarrow \left[\begin{array}{l} \texttt{has\_line}(B, \texttt{x}, 3, 2, El_1),\ \texttt{has\_elements}(B, \texttt{blank}, El_1, 1), \\ \texttt{has\_line}(B, \texttt{x}, 3, 2, El_2),\ \texttt{has\_elements}(B, \texttt{blank}, El_2, 1), \\ \texttt{isct}(El_1, El_2, It_1),\ \texttt{length}(It_1, LenIt_1), \\ \texttt{has\_elements}(B, \neg\texttt{blank}, It_1, LenIt_1) \end{array}\right] \quad \text{by 3}$$

All the references to M have been removed, and the resulting pattern is one that can be usefully matched against board positions.

4.4 Simplification

Elgar spends much of its time manipulating boolean expressions in DNF. Throughout the learning process, it constantly tries to simplify these expressions so as to reduce the workload of any subsequent phases.

The following is a table of the strategies used. All strategies replace an expression with another that is logically equivalent.

$$\begin{array}{lcl} [\ldots p, \ldots \neg p \ldots] & \rightarrow & [] \\ [\ldots \mathit{false} \ldots] & \rightarrow & [] \\ [P, \ldots \mathit{true}, Q] & \rightarrow & [P, \ldots, Q] \\ [] \texttt{ or } P & \rightarrow & P \\ [\texttt{has\_elements}(B, X, [El], 1), El \in S)] & \rightarrow & [\texttt{has\_elements}(B, X, S, 1)] \\ [P, \texttt{has\_elements}(B, X, S, 0)] & \rightarrow & [P] \\ [\ldots x \in A, x \in B, x \notin A \cap B] & \rightarrow & [] \\ [\ldots x \notin A, \ldots x \in A \cap B \ldots] & \rightarrow & [] \\ [\ldots \texttt{at}(Pos, Col) \ldots] & \rightarrow & [\ldots \texttt{has\_elements}(B, Col, [Pos], 1) \ldots] \\ \left[\begin{array}{l} \texttt{has\_elements}(B, X, S, N), \\ \texttt{has\_elements}(B, X, S, M) \end{array}\right] & \rightarrow & [\texttt{has\_elements}(B, X, S, M)] \text{ (if } M > N) \\ \left[\begin{array}{l} \texttt{has\_elements}(B, X, S, N),\ \texttt{length}(S, L), \\ \texttt{has\_elements}(B, Y, S, M) \end{array}\right] & \rightarrow & [] \text{ (if } M + N > L \text{ and } X \text{ and } Y \text{ don't match)} \end{array}$$

The majority of these simplifications eliminate contradictions. The references in the table to *true* and *false* come about because ELGAR's simplification stage also looks through the whole expression and evaluates any literals that are completely ground (such as `3>4`).

4.5 Summary

The learning algorithm that ELGAR uses can be summarised as follows:

Input: A board position, either a loss or an impasse for `o` (ELGAR).

Output: A general pattern that describes a class of board positions such that the opponent could move on any of the positions and bring about a board position that would in turn be a loss or impasse in just the same way as the specific example.

Algorithm: The algorithm proceeds in two phases:

1. **Phase One:** If the position is a loss, use EBL to derive a general description G of a class of board positions that are all losses in the same way. Otherwise, if the position is an impasse, use EBL to produce a disjunction of generalisations to describe the board position that must come about after ELGAR's move. Next, regress this pattern over $\texttt{move}(M, \texttt{o})$ to produce a general pattern G characterising the impasse.
2. **Phase Two:** Regress G over the move by `x` to produce a pattern describing a class of positions to avoid.

The regression algorithm that ELGAR uses can be summarised as follows:

Input: A pattern P and a move at M by player C.

Output: A new pattern P', such that $P' \models wp(\texttt{move}(M, C), P)$.

Algorithm: The algorithm proceeds in two steps:

1. Calculate $wp(\texttt{move}(M, C), P)$, constraining M with advice clauses as appropriate.
2. Strengthen the weakest preconditions to remove any inappropriate references to M.

At each stage, the expression being manipulated is simplified as much as possible.

5 Pattern use: how Elgar plays games

Before playing, Elgar requires a description of the game that it is to learn. This description consists of two parts: a specification of what constitutes a won board position, and an operator that specifies how to make legal moves. This description is expressed in terms of Elgar's pattern language. Elgar alternates turns with an opponent, and on its turn proceeds as follows.

1. If the current position is a loss for Elgar, then this is a learning opportunity; go to step 4. Otherwise continue.

2. Perform a one ply search of the game tree. Evaluate each resulting position by attempting to match it against stored patterns. If any pattern matches, then return a verdict as to whether the match signals a win for the program or the opponent. If all moves lead to losses, then this is an impasse and a learning opportunity; go to step 4. Otherwise, if there is a move that leads to a win for Elgar, pick that move. If there isn't one, play a move at random and continue.[2]

3. Apply the chosen move and wait for the opponent's response. Return to step 1.

4. This is a learning opportunity brought about either by losing the game or by reaching an impasse. In either case, invoke pattern learning to acquire a general pattern characterising the class of positions that must have occurred before the loss or impasse. Add this pattern to Elgar's list of stored patterns.

   Assess the board position two turns before. If in the light of Elgar's new knowledge, this turns out to have been an impasse, then recursively call the learning algorithm to acquire another pattern.

5.1 Competing forcing sequences

The algorithm given above does a reasonable job as the game driver for the Elgar system. However, it fails to evaluate the possible effect of its own moves on the situation. In particular, it does not recognise the possibility that a forcing sequence of its own may negate a threat of its opponent. Assuming it has learnt the three patterns for noughts and crosses of section 3, Elgar will resign in the following position:

x		
	o	
		x

Elgar sees this position as an impasse because, no matter how it plays, it thinks that the board will still be left in a position where `x` can play in one of the unoccupied corners and create a double threat. This analysis is correct insomuch as it correctly sees the opponent's patterns, but it fails because Elgar does not recognise that if it doesn't play in a corner, it can set up a situation where the opponent doesn't have time to set up the double threat.

That is, in looking at a position such as

[2] Alternatively, and this is really an extension, one attaches a numeric valuation to game states based on matched patterns rather than just a win/loss verdict and use this as a basis for the selection of the next move.

x		
	o	
	o	x

Elgar needs to be able to realise that x is unable to exploit the double threat (by playing at 3) because the threat of the line of o's is more pressing. In fact, to evaluate this position, Elgar would really have to play out the following sequence of forced moves:

x_1	x_5	o_6
o_8	o_2	x_9
x_7	o_4	x_3

and see that the game ends in a draw. This would require a more sophisticated game driver that employed some form of tightly constrained game-tree search. A suitable game driver is beyond the scope of this project, but is a useful direction for future work. Elgar's current game driver is intended only to provide a minimal setting in which patterns can be acquired and usefully applied.

6 Systems and their implementation

6.1 General implementation of Elgar

Elgar is implemented in a series of relatively discrete modules. These are as follows:

Game driver: This module implements the basic game-playing algorithm described in section 5. It calls the generalisation and regression code provided by other modules to implement pattern learning, and maintains the database of learnt patterns.

Pattern regression: This module provides the pattern regression code that is described in section 4.3. There have been two versions of this module; one for the primitive pattern language used in `nac-2` and `gm-1` (essentially the Strips algorithm) and one for the full pattern language used in `nac-3` and `gm-2`. This module also includes code for strengthening as per section 4.3.3.

Generalisation: This module provides an elegant implementation of the EBL algorithm along the lines suggested by (van Harmelen and Bundy, 1988; DeJong and Mooney, 1986; Kedar-Cabelli, 1988).

Domain information: This module provides the system with a description of the game being played in order to allow it to reason about it. There are two such modules, one for noughts and crosses, and one for go-moku. The rest of the system is sufficiently general that switching from one game to the other is simply a matter of swapping the appropriate domain information modules.

Support: This module provides a variety of useful routines used by most of the other modules. For example, it includes a Prolog implementation of list functions *map*, *reduce*, and *zip*.

6.2 Specific game-playing sub-systems

The following is a survey of the various sub-systems that enabled Elgar to learn the various games of the study to varying degrees of success.

6.2.1 `nac-1`

This was the first game-playing subsystem implemented for Elgar. It uses the primitive pattern language.

After losing just one game, the system will learn a rule that it can use to try and avoid one of 24 possible three-in-a-row threats. `nac-1` takes three games to learn that $\begin{array}{c|c|c} \mathtt{x} & & \\ \hline & & ? \\ \hline & ? & ? \end{array}$ is a position to avoid.

Because it uses the primitive pattern language, whose representational vocabulary is identical to the vocabulary of the add and delete lists of the move operator, `nac-1` can employ the standard Strips regression algorithm (Nilsson, 1980).

6.2.2 `gm-1`

This was a "proof of concept" system for playing go-moku on a 10×10 board. The system used the same pattern language as `nac-1`. This system was very inefficient, because it needed to enumerate 192 different five-in-a-rows just in order to be able to recognise won positions.[3] However, it did demonstrate that the regression algorithm used in `nac-1`, in conjunction with the general game driver sub-systems, worked over a different domain.

6.2.3 `nac-2`

This system was the second noughts and crosses system developed as part of ELGAR. It uses the same basic game driver components as `nac-1`, but uses the full pattern language to describe board position descriptions, and uses the new methods of section 4 to learn patterns. It learns the three patterns for noughts and crosses (see section 3) in just two losing games.

It was at this stage of development that I realised for the first time that even noughts and crosses suffers from the problem of "threats and counter-threats" that is discussed in section 5.1. The problem had not arisen up until that point because the patterns stored in `nac-1` had never been sophisticated enough to match situations in the required way.

6.2.4 `gm-2`

Using the approach of `nac-2` (i.e. the full pattern language and the associated generalisation and regression code), `gm-2` learns a number of useful patterns in the game of go-moku. Murray and Elcock, in their 1968 paper (Murray and Elcock, 1968) present a list of 21 go-moku patterns that I attempted to teach `gm-2` by playing the appropriate training games. Unfortunately, the system only learnt four of the most basic patterns (involving up to four intersecting lines) before playing so slowly as to make further work too painful. The bottleneck was not the learning component, but the pattern-matching algorithm. Informal analysis indicates that the matching is exponential in the number of lines in the pattern. A list of the patterns that `gm-2` did acquire appears in appendix B.

[3] The number of lines of length k on a $n \times n$ sized board is $2(n-k+1)^2 + 2n(n-k+1)$.

7 Previous work

7.1 Game learning programs

The task of getting a computer to not only play a board game, but to learn as it does so is not a new one. A very early paper, (Koffman, 1968), looks at this problem over a specified class of "line-games" including Hex, go-moku, *Qubic* (three-dimensional $4 \times 4 \times 4$ noughts-and-crosses) and "Bridge-it". A contemporary piece of work, (Murray and Elcock, 1968) learns how to play go-moku. Both learn to play quite strongly. In fact, it is the claim in (Murray and Elcock, 1968) that their go-moku program plays at near expert level.

However, both (Koffman, 1968) and (Murray and Elcock, 1968) take rather a special-purpose approach that makes the logical analysis of the game situations implicit, and which also requires specially tailored data structures. It is my belief that the approach embodied by my work on ELGAR is useful because it performs explicit reasoning about the game states in a manner which is both general, and also powerful enough to handle games such as chess and go.[4]

In both articles, the systems described use representations that encode information about lines of pieces (possibly with holes in them), and about the intersections between them. Even a very superficial analysis of the games concerned reveals that these are indeed important concepts, but these approaches mean that the systems commit themselves to just these games while failing to allow for the possibility of extending to others.

Moving forward almost 20 years, we come to Steven Minton's (Minton, 1985) which describes the implementation of a system similar to ELGAR. However, his system is based around a high-level description of the game that seems to make undue assumptions about which aspects of the game are most important. Rather than allow the system to develop its own descriptions of the game using high-level abstractions, these are forced onto it by describing the game state in terms of lines from the very outset.

Regression, an important component of ELGAR, is finessed by Minton's approach because of the way in which these patterns are calculated. When a move is made in Minton's system, it automatically calculates the effect that this move has had on the existing lines, and passes forward the new description of the board. When Minton comes to perform the equivalent of regression, all his system need do is reverse the action of the forward chaining already performed. By carefully casting the game representation at exactly the right level of generality from the outset (i.e. at the level of lines and intersections), Minton dodges many of the interesting issues that ELGAR addresses.

None of these systems explicitly address the problem of evaluating competing forcing sequences (identified in section 5.1).

7.2 Explanation-based generalisation

The main sources I have used in investigating EBL have been (DeJong and Mooney, 1986) and (Mitchell et al., 1986). Both of these present quite clear explanations of exactly what constitutes example-based generalisation. While they are useful introductions to the concept, they consider EBL in general, and so discuss some issues that seem of little relevance to a game-playing system.

---

[4]Subject to provisoes about extending my model beyond STRIPS; see section 2.1.1.

7.2.1 Explanation-based generalisation : a unifying view

This article (Mitchell et al., 1986) introduces the general concepts of explanation-based generalisation (EBG), and spends some time discussing EBG's limits and further research directions. (EBG is a particular implementation of EBL.) After describing EBG in general, Mitchell *et al.* spend much of their article going through a set of examples which illustrate EBG techniques.

Mitchell *et al.*'s characterisation of the explanation-based generalisation problem is summarised in Figure 7.

Given:

- *Goal Concept:* A concept definition describing the concept to be learned. . . .
- *Training Example:* An example of the goal concept.
- *Domain Theory:* A set of rules and facts to be used in explaining how the training example is an example of the goal concept.
- *Operationality Criterion:* A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed.

Determine:

- A generalisation of the training example that is a sufficient concept definition for the goal concept and that satisfies the operationality criterion.

Figure 7: The explanation-based generalisation problem (from (Mitchell et al., 1986)).

This terminology can be applied without difficulty to ELGAR's game-playing task. In the course of a game, if ELGAR determines that a given position is a loss or an impasse, then the goal concept is a proposition representing that a loss or an impasse exists; the training example is the board position reached; the domain theory is the set of predicates that allow ELGAR to prove positions are impasses or won positions; and the operationality criterion is the rule that tells ELGAR that it should express its generalisation in terms of the board position that must have held before the opponent made its move.

The examples in this article make it clear just how Mitchell *et al.*'s formulation of EBG would work. The first example is that of the *SAFE-TO-STACK* problem, where the system must learn general conditions for being able to stack one item on top of another in a blocks-world domain. Mitchell *et al.*'s method is to generate an explanation (or proof) that the training example satisfies the goal concept, and to then *regress* a generalised form of the goal concept back through this "explanation structure". (Mitchell et al., 1986) explains regression thus:

> In general, *regressing* a given formula F through a rule R is a mechanism for determining the necessary and sufficient weakest conditions under which that rule R can be used to infer F. We employ a slightly modified version of the goal-regression algorithm described in Waldinger (1977) and Nilsson (1980). Our . . . algorithm computes an expression that represents only a sufficient condition (rather than necessary and sufficient conditions) under which rule R can be used to infer formula F. . . . In particular, whereas the general goal regression algorithm considers all possible variable bindings (unifications) under which R can infer F, our modified algorithm considers only the specific variable

bindings used in the explanation of the training example.[5]

Mitchell *et al.*'s other two examples are constructed to enable a comparison between EBG and other machine learning systems. The first illustrates the way in which an EBG system would handle the *CUP* generalisation problem. This problem, taken from Winston *et al.*'s ANALOGY system, is to find those properties that will in general ensure an object is a cup, after being given a fully described instance of a cup (including irrelevant details such as colour and owner) to examine.

The third example in (Mitchell et al., 1986) is that of LEX. This system is one which acquires heuristics that enable it to solve symbolic integration problems. This consists of analyzing a successful solution of a given problem to find conditions on the general problem that ensure the chosen strategy will lead to a solution. This, as Mitchell *et al.* point out, is similar in feel to Minton's approach to playing go-moku.

Mitchell *et al.* finish with a discussion of possible areas for future research in the area of EBG.

7.2.2 Explanation-based learning : an alternative view

It is this article (DeJong and Mooney, 1986) rather than (Mitchell et al., 1986) that has most directly inspired the approach to EBL followed in this project. In particular, the method presented here for the construction of generalised proofs is followed rather than that presented in (Mitchell et al., 1986), primarily because of the simplicity with which it could be implemented in Prolog. DeJong and Mooney's algorithm is given in Figure 8. Incidentally, DeJong and Mooney prefer the term *Explanation-Based Learning* (hence their title) because they feel that it is "both possible and desirable to apply the approach to concept refinement (i.e., specialisation) as well as concept generalisation."[6]

---

(1) If there is no observed operator input, use the domain theory to construct a plan which achieves the goal. Otherwise, use the domain theory to build a causally complete interpretation of the observed operator input. In either case, maintain GENERAL and SPECIFIC substitutions....

(2) Eliminate operators and states which do not causally support the goal. The remaining structure is the solution explanation.

(3) Identify nominal instantiations of known schemata and eliminate them by retracting the unifications between the general specification of the schema and the specific operators used in the instantiation.

(4) Remove actions and states which only support inferences to more abstract actions and states. This is done by removing those abstraction inferences along with their antecedents and retracting the unifications binding them to the rest of the explanation.

(5) ...[optimize proof tree by checking for subgoals that can be achieved more efficiently]....

(6) Generate the final generalised explanation by applying the resulting GENERAL substitution to the remaining explanation structure.

Figure 8: The generalisation method.

[5] p54,(Mitchell et al., 1986). For more on regression see section 4.3.

[6] p146,(DeJong and Mooney, 1986)

DeJong and Mooney's work is presented in the context of their system *GENESIS* (Mooney and DeJong, 1985) which "processes short English narratives and is able to acquire, from a single narrative, a new schema for a stereotypical set of actions."[7] If *GENESIS* is given a sufficiently detailed story that demonstrates a new schema, it can learn that schema and use it to identify applications of the same schema in other, less detailed narratives. The example used in (DeJong and Mooney, 1986) is that of a kidnapping narrative where the system is given a very detailed story and uses this to construct a general schema for kidnapping.

The detailed story given to *GENESIS* includes information such as the fact that the victim Mary was wearing jeans, that the kidnapper forced her into his car with a gun, and that he tied her up in his hotel room. However, the important aspects of the story that *GENESIS* retains in its generalisations are just those facts that allow it to notice that both a *CAPTURE* and a *BARGAIN* event have occurred (schemata that it already recognises). Subsequent stories that *GENESIS* is exposed to can omit or change some of the details, without affecting its ability to recognise the new narrative as a *KIDNAPPING*.

The section of DeJong and Mooney's article most relevant to work on Elgar is a presentation of their own approach to the task of explanation-based learning. There are two main differences between this approach and Mitchell *et al.*'s work. The first is the revision of the algorithm so that the general proof tree is not constructed by regression through a specific proof, but is constructed along side it: the general proof acquires variable bindings in parallel with the specific proof.

Secondly, DeJong and Mooney present two useful tree-pruning strategies that allow the derivation of more powerful generalisations by losing unnecessary detail. The first strategy involves discarding components of higher level actions or operators that do not contribute to the proof except in as much as they support the higher level action. In the kidnapping example used, this allows their system to discard the information that the kidnapper captured the victim by threatening her and forcing her into his car. These actions are recognised as being no more than components of the higher level *CAPTURE* action, and because they do not contribute to any other part of the tree, DeJong and Mooney's system allows them to be discarded.

The second strategy is to discard actions or operators whose only role is as antecedents of inferences that lead to more abstract actions and states. In this way, the *TELEPHONE* action used by the kidnapper to relay his ransom demand is discarded from the proof tree because its only role is to allow the system to infer that a *COMMUNICATE* action took place (which in turn is a precondition of the *BARGAIN* event).

[7] from the abstract in (Mooney and DeJong, 1985).

8 Conclusions and further work

ELGAR is an implemented system that acquires patterns from examples in the domain of the two-person turn-taking complete information games. Patterns encode opportunities to win and positions to avoid. The current implementation acquires patterns for the games of noughts and crosses and go-moku. ELGAR makes four significant contributions to the state of the art in knowledge intensive machine learning:

1. *A methodology for the design of pattern languages.*

   ELGAR's full pattern language embodies two important design features that greatly facilitate its use in pattern learning:

   (a) The full pattern language allows the definition of higher level concepts such as `has_line`in terms of a small set of *elementary* predicates. This reduces the task of regressing high-level predicates to the task of regressing elementary predicates.

   (b) Elementary predicates are divided into two classes: those which are positionally invariant, and those which are positionally dependent. Regression of predicates in the first class is trivial: they regress over the `move` operator unchanged. It is left to the user to specify how positionally dependent predicates are regressed. However, this class is small in ELGAR (just the `has_elements` predicate), so this task is relatively straightforward. It also seems likely that these specifications can be generated automatically.

   ELGAR's full pattern language is described in section 2.2.2.

2. *Weakest precondition computation.*

   ELGAR embodies a *wp*-semantics for STRIPS operators that allows weakest preconditions to be calculated for expressions involving high-level predicates whose vocabulary is not that of the add and delete lists of the operators. High level descriptions are decomposed into their components described using elementary predicates, and then these components are regressed by considering the elements of the operator's add and delete list one by one. The user defines how elementary predicates are regressed by specifying the contribution of individual elements of the add and delete lists to the computation of their weakest preconditions.

   Computation of weakest preconditions is discussed in section 4.3.1.

3. *Strengthening.*

   Weakest preconditions often contain variables referring to the move over which a post-condition has just been regressed. ELGAR uses three novel strategies to strengthen these weakest preconditions, producing patterns that no longer include the inappropriate variable references. These patterns are guaranteed to entail the weakest precondition, and appear to be no stronger than necessary.

   Strengthening is discussed in section 4.3.3.

4. *Use of examples in regression.*

   ELGAR uses "advice clauses" to restrict the number of disjunctions it is forced to consider when calculating a DNF representation of the weakest preconditions of a given move. These advice clauses are derived by matching the example to a general pattern to be regressed. ELGAR uses examples to guide learning in a more flexible way than existing EBL techniques, while retaining the efficiency advantages of using examples to drive learning.

Advice clauses and their construction are discussed in section 4.3.2.

This research has also uncovered a number of promising directions for future work, as I now discuss.

8.1 Further work

It is clear that ELGAR is not a complete game-playing system. At the level of tactical knowledge, there are still two important areas that need further investigation. Currently, ELGAR's game driver makes no attempt to play aggressively. A trivial improvement to this would be to have the game driver recognise whenever it finds itself in a situation that it knows is a bad position for its opponent. It could then exploit the position so as to bring about a win. This is entirely fortuitous aggression however. To actually plan to bring about winning situations seems to be a more difficult task.

ELGAR's simple game driver also fails in situations where a forcing sequence of its own might negate the threat of an impasse (see section 5.1). To fix this problem would seem to require integrating ELGAR's pattern learning with some sort of game-tree search. In this way, search could be guided by previously learnt patterns. Wilkins' work on the use of plans in the middle game of chess may provide a useful guide in this area (Wilkins, 1979).

Another important extension to ELGAR would be to allow it to learn patterns that represent a broad range of *tactics*. Tactics are standard plans to achieve local goals, without consideration of the broader game situation. In a game as small as noughts and crosses, tactical knowledge is essentially all there is to the play of the game; moreover, the only tactical goal in noughts and crosses is to win. Consequently, all of ELGAR's patterns describe only win/loss situations. Further kinds of tactics must be acquired for more complex games such as Othello or Go. In othello, a tactic might aim to acquire a corner square, while in Go a tactic might aim to capture a cutting stone.

Even if ELGAR were extended to handle a range of tactical situations, it would not suffice as a general game-player in games where strategic considerations are important. In particular, ELGAR's framework offers no advice as to how to select between various tactical situations. In some larger system, an ELGAR-like pattern matcher could instead be used as a component which served to identify areas of interest for a higher level, strategic decision-maker.

Unlike the pattern languages used by a number of previous systems (Koffman, 1968; Murray and Elcock, 1968; Minton, 1985), ELGAR's pattern language instantiates a general methodology for pattern language design that can be readily extended to other games. In particular, it should be relatively easy to extend the set of positionally invariant elementary predicates (such as `collinear`, `isct`, and `length`) to suit different games. If all that is changed are these unvarying predicates, then pattern learning should not need to change at all. As a demonstration of how this extension might be made, Figure 9 below includes code that defines a pattern matching a set of board positions including an *eye* in Go, an important tactical concept.

The four new positionally invariant predicates used in this definition are straightforwarded to specify. The first, `diamondshape`, is true of a set of four board locations if they form a diamond shape around a central point. On a noughts and crosses board, `diamondshape` is true only of the set $[2, 4, 6, 8]$. The predicate `outlinesquare` is true of a set of board locations and a size if the locations delineate a square of the specified size, with the locations as the square's corners. The third predicate, `union` is true if the union

```
has_eye(Board, Colour, Centre, Corners, Edges) :-
    diamondshape(Edges),
    outlinesquare(Corners,3),
    union([Centre, Corners, Edges], WholeArea),
    forms3by3square(WholeArea),
    has_elements(Board, Colour, Corners, 3),
    has_elements(Board, Colour, Edges,   4),
    has_elements(Board, blank, Centre,   1).
```

Figure 9: A pattern to match a Go "eye"

of the sets in the list given as its first parameter is equal to the second parameter. Finally, `forms3by3square` is true of a set of board locations if they form a 3×3 square on the board. This is far from a complete definition of the concept, but it captures a reasonable number of instances in the class.

To reiterate, ELGAR wouldn't be sufficient to play a game such as Go, but its pattern-based approach might be used to guide search in order to reduce the otherwise enormous branching factor. Many problems would still remain, in particular the problem of high-level strategy, but Go might become a little more feasible.

In the current implementation, ELGAR's regression of descriptions expressed in the full pattern language relies on the user providing a *wp* definition of the effect of add and delete operations on the `has_elements` predicate. (All other elementary predicates are positionally invariant, so they are not affected by regression and do not require this analysis.) It would be a useful extension if ELGAR could perform this analysis itself. Such a task would seem to require the development of a theory that articulates the relationship between the positionally dependent predicates and the low level `at` literals used to represent board positions and STRIPS operators.

Finally, it is clear that to scale up to more complex games, the pattern-matching code that ELGAR uses should be improved. Currently, ELGAR uses a very naïve backtracking approach that works without difficulty on the 9-cell board of noughts and crosses, but which effectively grinds to a halt on the larger boards and more complicated patterns that arise in go-moku. There may well be potential in investigating the possibility of "compiling" these logical descriptions of patterns into code designed for efficient recognition. Just reducing the amount of backtracking done by the system would improve performance considerably.

Indeed, many of the earlier pattern-matching systems (Koffman, 1968; Murray and Elcock, 1968) employed representations that are pre-compiled in exactly this way. In contrast to these systems, I believe it is important to distinguish the logical descriptions of patterns from the representation used to accomplish recognition. This is important because a pattern described so as to facilitate recognition will not necessarily be as easy to reason about when it comes to perform processes like generalisation or regression.

In summary, I have identified five possible directions for further work:

1. Build a more sophisticated game driver that makes better use of the patterns that ELGAR learns.

2. Learn patterns that represent tactical situations instead of just those that represent win/loss situations.

3. Develop an algorithm to allow the automatic construction of clauses describing how to regress elementary predicates over elements of the add and delete lists of STRIPS operators.

4. Improve the efficiency of pattern recognition through some form of pattern compilation.

5. Extend ELGAR to play new games. Perhaps Othello is a reasonable next step. Attendant with this is a switch from STRIPS operators to ADL operators.

References

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

Dijkstra, E. W. (1975). Guarded commands, non-determinacy and formal derivations of programs. *Communications of the ACM*, 18(8):453–457.

Kedar-Cabelli, S. T. (1988). *Formulating concepts and analogies according to purpose.* PhD thesis, Rutgers University.

Koffman, E. B. (1968). Learning games through pattern recognition. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(1):12–16.

Minton, S. (1985). A game-playing program that learns by analyzing examples. Technical Report CMU-CS-85-130, Carnegie-Mellon University.

Minton, S. (1988). *Learning effective search control knowledge: an explanation-based approach.* PhD thesis, Department of Computer Science, Carnegie-Mellon University.

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

Mooney, R. and DeJong, G. (1985). Learning schemata for natural language processing. In *Proceedings of IJCAI-85*, pages 681–687, Los Angeles, CA. IJCAI, Inc.

Murray, A. M. and Elcock, E. W. (1968). Automatic description and recognition of board patterns in go-moku. In *Machine Intelligence 2*, pages 75–88. American Elsevier, New York.

Nilsson, N. J. (1980). *Principles of artificial intelligence.* Morgan Kaufmann, Los Altos, CA.

Pednault, E. P. D. (1986). *Toward a mathematical theory of plan synthesis.* PhD thesis, Stanford University.

van Harmelen, F. and Bundy, A. (1988). Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36:401–412.

Wilkins, D. E. (1979). *Using Patterns and Plans to Solve Problems and Control Search.* PhD thesis, Stanford University.

A The EBL algorithm used in Elgar

```
generalise(TrainingExample, TargetConcept, NewClause) :-
        ebl(TrainingExample, TargetConcept, Reformulation),
        NewClause = (TargetConcept :- Reformulation).

ebl((Goal1 ; _), (GenGoal1 ; _), Leaves) :-
        ebl(Goal1, GenGoal1, Leaves), !.

ebl((_ ; Goal2), (_ ; GenGoal2), Leaves) :-
        ebl(Goal2, GenGoal2, Leaves), !.

ebl((Goal1, Goal2), (GenGoal1, GenGoal2), (Leaves1, Leaves2)) :-
        !,
        ebl(Goal1, GenGoal1, Leaves1),
        ebl(Goal2, GenGoal2, Leaves2).

ebl(Leaf, GenLeaf, GenLeaf) :-
        operational(Leaf),
        !,
        Leaf.

ebl(Leaf, GenLeaf, GenLeaf) :-
        predicate_property(Leaf, built_in),
        % true if Leaf is a built-in predicate
        !,
        Leaf.

ebl(Goal, GenGoal, Leaves) :-
        clause(GenGoal, GenBody),
        copy_term((GenGoal :- GenBody), (Goal :- Body)),
        ebl(Body, GenBody, Leaves).
```

B Go-moku patterns learnt by Elgar

The following are the Prolog descriptions for the four patterns that `gm-2` has currently learnt. While far from a complete list of useful pattterns, they do ensure that Elgar plays a reasonable defensive game. More patterns haven't been acquired to date because of the slowness with which the more complicated patterns are matched, resulting in games that become unplayably slow.

The predicate used to define the patterns is `badpos`. It is true of parameters `A` and `B` if `B` is a board position that, with the move, player `A` can play on and force a win.

```
badpos(A, B) :-
        has_line(B,A,5,4,C),
        has_elements(B, blank, C, 1).

badpos(A, B) :-
        has_line(B,A,5,3,G),
        has_line(B,A,5,3,F),
```

```
        isct(F, G, E),
        length(E, D),
        D>0,
        has_elements(B, blank, F, 2),
        has_elements(B, blank, G, 2),
        has_elements(B, (C\==blank):C, E, D-1).

badpos(A, B) :-
        has_line(B,A,5,2,F),
        has_line(B,A,5,2,G),
        has_line(B,A,5,3,H),
        isct(F, G, E),
        isct(H, G v F, I),
        length(E, D),
        length(I, J),
        D-1>0,
        J>0,
        has_elements(B, blank, H, 2),
        has_elements(B, (C\==blank):C, E, D-1-1),
        has_elements(B, blank, G, 3),
        has_elements(B, blank, F, 3),
        has_elements(B, (K\==blank):K, I, J-1).

badpos(A, B) :-
        has_line(B,A,5,2,M),
        has_line(B,A,5,2,L),
        has_line(B,A,5,2,K),
        has_line(B,A,5,2,J),
        isct(K, J, I),
        isct(M, L, H),
        isct(J v K, L v M, G),
        length(I, F),
        length(H, N),
        length(G, E),
        F-1>0,
        N-1>0,
        E>0,
        has_elements(B, (D\==blank):D, I, F-1-1),
        has_elements(B, blank, J, 3),
        has_elements(B, blank, K, 3),
        has_elements(B, (D\==blank):D, H, N-1-1),
        has_elements(B, blank, L, 3),
        has_elements(B, blank, M, 3),
        has_elements(B, (C\==blank):C, G, E-1).
```