Learning to Forecast by Explaining the Consequences of Actions

Tristan Cazenave

LAFORIA-IBP
Case 169
Université Pierre et Marie Curie
4, place Jussieu
75252 PARIS CEDEX 05
Tel: 44 27 70 10
Fax: 44 27 70 00
cazenave@laforia.ibp.fr

Abstract

I explain a method to learn to achieve goals in games. In very complex games, such as the game of Go, a search intensive approach is intractable. A knowledge intensive approach is best suited. I represent knowledge using the combinatorial game theory and first order logic. I give a method to learn to predict the consequences of some moves. Each rule is learned using a single example, by explaining the transformations involved by the move. This learning method has been implemented in Gogol, a Go playing system. It learns to evaluate games representing the achievement of some interesting goals of the game of Go. This method is presently applied to other games and to decision making in the management of a firm.

Key words : Explanation Based Learning, Combinatorial Game Theory, Metaknowledge, Game of Go.

1 Introduction

When a domain theory exists, a learning method has been developed: Explanation Based Learning (EBL) [Mitchell 1986] [Dejong 1986]. This learning method is particularly useful in games as they have a strong domain theory. Many projects involving EBL and games have been developed to learn plans [Minton 1984] [Puget 1987] [Pell 1993]. The initial work of J. Pitrat on Chess [Pitrat 1976] has been a precursor of such approaches to learning in games. Meanwhile, the knowledge representation of these programs make them learn overly general knowledge. This obliges them, either to partially verify their assertions, or to use their learned knowledge as heuristics. This also the case in the work of [Tadepalli 1989] on lazy EBL: his program learns overgeneral rules and refines them. Unlike these programs, my approach is to learn only certain rules. In order to achieve this goal, I explicitly represent uncertainty and separate clearly what is uncertain from what is certain. I

do not want to verify the predictions made by the learned rules. My learning algorithm does not learn overgeneral rules. It only learns correct rules.

My approach also differs from project as Prodigy [Minton 1988], in which EBL is used to learn search control knowledge. Our goal is to replace the problem solving activity by the efficient matching of a large base of rules.

In a first part, I show how I manage to represent clearly uncertainty using an adapted combinatorial theory of games. This representation allows to regress goals easily. In a second part, I show the limits of the knowledge representation used in previous works, and I propose a solution to overcome these limits. In a third part, I show how to use my algorithm to learn to achieve goals in the game of Go. The learning algorithm described in this paper can be used in many other domains. It is presently applied to other games (Chess and Abalone) and to decision making in the management of a firm.

2 Introducing Uncertainty in Combinatorial Game Theory

Combinatorial Game Theory is a mathematical theory which has been developed by J. H. Conway [Conway 1976], and adapted to many games by Berlekamp, Conway and Guy [Berlekamp 1982].

Definition : A combinatorial game x = {G|D} is composed of two sets G and D of combinatorial games. Every combinatorial game is constructed this way.

A combinatorial game defines a tree of possible positions. Each position can be assigned two values. The first value will be either Won for Left (WL) or Lost for Left (LL). The second value will be either Won for Right (WR) or Lost for Right (LR). For each node, we can therefore have four possibilities :

	LL	WL
LR	The player who begins loses, the game is NULL (0)	The left player always wins, the game is POSITIVE (+)
WR	The right player always wins, the game is NEGATIVE (-)	The player who begins wins, the game is FUZZY (||)

Table I

This theory is very efficient in simple games such as the game of Nim. It has been applied with success to the endgame of Go [Wolfe 1991]. But this theory requires the subgames of a same position to be independent, it also requires to be able to forecast all the possible moves until the end of the game. These assumptions do not apply in complex games such as Go and Chess. I show in this article that the introduction and the management of uncertainty in Combinatorial Game Theory permits to use it in complex games.

In this section, I describe the introduction of uncertainty in Combinatorial Game Theory. It is necessary in complex games to represent what is unknown because it

is impossible in such games to forecast all the positions, only a small subset of the positions can be forecasted. Representing uncertainty in Combinatorial Game Theory permits to extend this theory to complex games, and to represent the partial deduction a system can make about a position. It also allows to take strategic decisions on the way to handle what is not predictable in a reasonable time.

Theoretically, a game has only two possible values : Won (G) or Lost (P). However, in practice it is often very costly and most of the time intractable to find the value of a game [Allis 1994]. Therefore, we introduce the Unknown (I) value which accounts for the fact that we do not have enough time to compute the value of a particular game. Let J = {G,I,P} be the set of all possible states for a game.
The dynamic definition of a game consists in defining the states of the game after the best friend move and after the best opponent move. If $x,y \in J$: xy defines the game for which the best state the friend player can achieve if he plays first is x, whereas the best state the opponent player can achieve if he plays first is y. We define $J_1 = \{xy / x,y \in J\}$ and $J_n = \{xy / x,y \in J_{n-1}\}$. Computing an element of the set J_n is twice the cost of computing an element of the set J_{n-1}, therefore we limit ourselves to the elements of J_2.
In practice J_1 et J_2 are sufficient to represent the kind of knowledge used in games, the most common games used by players are :

- GG represents a Won game.
- GP represents an unsettled game.
- PP represents a lost game.
- PG represents a Zugswang in Chess or a Seki in Go. The first player to move loses.
- GIII represents a threat for the friend player. After the first move of the friend player, the game is GI. After the second move of the friend player, the game is G (Won).

3 A Declarative Knowledge Representation

The declarativity of knowledge presents many aspects. The first one is the explicit representation of knowledge, it allows the program to manipulate the knowledge it uses. This aspect is typical of logic programming techniques, as in Prolog programs or in expert systems. The second constraint that declarativity imposes on knowledge representation is that the instructions of a program must be given without the order to execute them. A declarative program is both explicit and given without a defined order [Pitrat 1990]. This second aspect is very important because it facilitates explanations and learning. As we will see, it also allows to learn using incomplete domain theories. I have chosen to represent knowledge using Horn clauses. I use the metapredicate ‘exist’ which verifies that a fact exists in the working memory, and the metapredicate ‘append’ which append a fact in the working memory. I do not use the metapredicates ‘absent’ and ‘remove’. I am explaining this choice in four parts. I begin, in a first part, with showing the limits of existing representations, in a second part I expose my solution to overcome these limits. In a third part I give the type of knowledge needed so as to learn in games, and the last part gives an example of a representation of a position in the game of Go.

3.1 Limits of existing representations

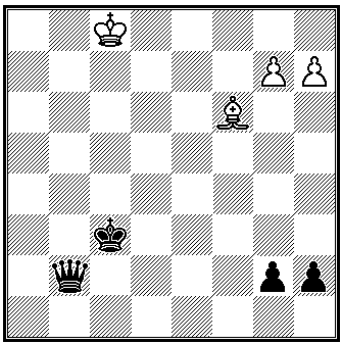

Figure 1

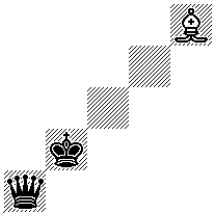

Figure 2

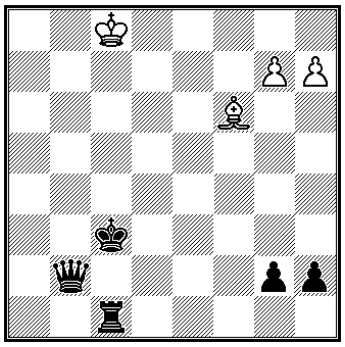

Figure 3

Figure 1 gives an example of a skewer in a chess position, traditional EBL systems create the rule of Figure 2 with the conclusion that the Black Queen can be taken. This rule is a generalization of Figure 1 using an explanation of why the Black Queen can be taken using a Chess domain theory, it appears in [Minton 1984]. However, as [Puget 1987] noticed, the method used by Minton and more generally the method given in [Mitchell 1986] can result in an overgeneralization of examples. Figure 3 gives a position in which the rule in Figure 2 reveals false, it is a too general rule. This overgeneralization of EBL is due to the use of the metapredicates ‘absent’ in the body of a rule or ‘remove’ in the conclusion of a rule. When such a rule exists in the domain theory, it may not fire on the learning example but it may fire later on another similar example on which the learned rule also applies. The rule containing a ‘remove’ metapredicate in its conclusion may remove one of the necessary conditions of the learned rule, in this case the learned rule applies but may lead to a false result. A solution would be to explain why a rule containing a ‘remove’ conclusion removing one of the conditions of the learned rule cannot apply. But negative explanation in first order logic leads to combinatorial explosion. We should not use the metapredicate ‘absent’ in the body of a rule for similar reasons : if we use it, we have to explain why a fact is not present in the working memory. There is a difference between a fact missing because it has not yet been deduced and a fact missing because it cannot be deduced. If we use a declarative representation without order between the firing of the rules, we cannot know whether a fact is missing because it has not yet been deduced or because it is not deductible. We have to use predicates which explicitly tell that a fact is not deductible, and we must not use negation as failure.

3.2 A solution

When not using the metapredicates ‘absent’ and ‘remove’, there is a problem for representing the change of the world after an action. Some facts are true before a move and false after. The solution I have chosen is the explicit representation of the evolution of the facts in time. For each fact which can change, I use a predicate which gives its value before the move and another which gives its value after the move. For example, in the game of Go, I have different predicates for the color of an intersection before and after a move.

Explicitly representing the evolution of facts over time avoids overgeneralization because it allows to have independent rules in the domain theory. The rules can be fired in any order, the result will be the same.

The metapredicate 'absent' is used, but only in metarules which tell the rules to fire. It is not used in the body of the rules of the domain theory.

3.3 Which knowledge to represent in this formalism

Knowledge on the topological configuration of the board cannot vary during a game, the corresponding facts are represented using only one predicate. Table II gives examples of some rules of the domain theory for the game of Go. The words beginning by a '?' represent variables. The friend color is represented by a '@', the enemy color by a 'O', and an empty intersection by a '+'. The rule Rule_evaluation_game_1 evaluates the game associated with taking the enemy stone on intersection ?i2. This game is Won if the friend player plays on the intersection ?i1 and if all the conditions in the body of the rule are fulfilled. The rule Rule_legal_move_5 gives sufficient conditions for a friend move on intersection ?i to be a legal move. The rules Rule_intersection_5 and Rule_block_1 are rules of transition between the position before the move and the position after the move.

Rule_evaluation_game_1 : If (exist (Color_intersection_after (?i O)) exist (Number_of_liberties_after (?i 1)) exist (Same_block_after (?i ?i1)) exist (Neighbour (?i1 ?i2)) exist (Legal_move_after (@ ?i2))) Then (append (Move_after (@ ?i2 Take ?i1 GI)))	Rule_legal_move_5 : If (exist (Color_intersection_before (?i +)) exist (Neighbour (?i ?i1)) exist (Color_intersection_before (?i1 +))) Then (append (Legal_move_before (@ ?i)))
Rule_intersection_5 : If (exist (Move (@ ?i)) exist (Legal_move_before (@ ?i)) exist (Neighbour (?i ?i1)) exist (Color_intersection_before (?i1 +))) Then (append (Color_intersection_after (?i @)))	Rule_block_1 : If (exist (Number_of_liberties_before (?i ?n)) greater (?n 1) exist (Same_block_before (?i ?i1))) Then (append (Same_block_after (?i ?i1)))

Table II

3.4 Representation of a position in the game of Go

To save room, I give in Table III the working memory corresponding to the 5x1 board[1] of Figure 4:

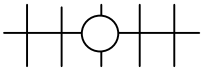

Figure 4

[1] Usually, the game of Go is played on 9x9, 13x13 or 19x19 boards.

The friend color is Black, the enemy color is White.

Color (@) Color (O) Opposite_colors (O @) Opposite_colors (@ O) Neighbour (i1 i2) Number_of_Neighbours (i1 1) Color_intersection_before (i1 +) Neighbour (i2 i1) Neighbour (i2 i3) Number_of_Neighbours (i2 2) Color_intersection_before (i2 +) Neighbour (i3 i2) Neighbour (i3 i4)	Number_of_Neighbours (i3 2) Color_intersection_before (i3 O) Number_of_liberties_before (i3 2) Number_of_stones_before (i3 1) Same_block_before (i3 i3) Neighbour (i4 i3) Neighbour (i4 i5) Number_of_Neighbours (i4 2) Color_intersection_before (i4 +) Neighbour (i5 i4) Number_of_Neighbours (i5 1) Color_intersection_before (i5 +)

Table III

4 Learning

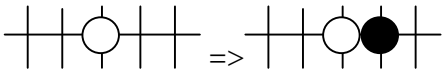

Figure 5

To illustrate the way learning works, I will use the position in Figure 5. The initial position is the one described in Table III and Black plays a move on intersection i4. The program deduces the transformations in the working memory due to the Black move. It selects a fact concluding on the games after the move, explains the deduction of this fact using only facts existing before the Black move. It regresses the rule created this way and generalizes it. Learning can be split in four parts : problem solving and regression, explanation, generalization. I give examples of the four parts of the learning of a new rule using the left part of Figure 5 which is logically equivalent to Table III.

4.1 Problem Solving

Problem solving is the deductive step which consists in firing rules until no new fact can be deduced. The facts deduced during problem solving are the facts created by the move. Table IV gives the facts deduced using our Go domain theory and the working memory in Table III.

Legal_move_before (@ i1) Color_intersection_after (i1 +) Legal_move_after (@ i1) Legal_move_before (@ i2) Color_intersection_after (i2 +) Legal_move_after (@ i2) Color_intersection_after (i3 O) Number_of_liberties_after (i3 1) Number_of_stones_after (i3 1) Same_block_after (i3 i3) Move (@ i4)	Legal_move_before (@ i4) Color_intersection_after (i4 @) Number_of_liberties_after (i4 1) Number_of_stones_after (i4 1) Same_block_after (i4 i4) Legal_move_before (@ i1) Color_intersection_after (i5 +) Legal_move_after (@ i5) Move_after (@ i2 Take i3 GI) Move_before (@ i4 Take i3 GIII)

Table IV

In Table IV, the interesting fact is the fact ‘Move_before (@ i4 Take i3 GIII)’. It is a fact that was not deduced before, but that has been deduced after the problem solving step. This fact has been deduced using a regression rule representing knowledge about the uncertain combinatorial game theory. The Table V gives some examples of regression rules.

Rule_regression_1: If (exist (Color (?c)) exist (Move (?c ?i2)) exist (Move_after (?c ?i2 Take ?i1 GI))) Then (append (Move_before (?c ?i2 Take ?i1 GIII)))	Rule_regression_2: If (exist (Color (?c)) exist (Opposite_colors (?c ?c1)) exist (Move (?c ?i2)) exist (Color_intersection_before (?i1 c1)) exist (Color_intersection_after (?i1 +))) Then (append (Move_before (?c ?i2 Take ?i1 GI)))

Table V

4.2 Explanation

The explanation module finds the facts related to a game before the move. Its goal is to create a rule explaining why this fact is present using only facts that are present before the move and the fact representing the move itself. To do this, it goes back through the rules fired during problem solving, replacing facts representing the position after the move by facts representing the position before the move. In our example, it selects the fact ‘Move_before (@ i4 Take i3 GIII)’ and finds the explanation given in Table VI:

Color_intersection_before (i1 +) Neighbour (i2 i1) Neighbour (i2 i3) Color_intersection_before (i2 +) Neighbour (i3 i2) Neighbour (i3 i4) Color_intersection_before (i3 O)	Same_block_before (i3 i3) Number_of_liberties_before (i3 2) Neighbour (i4 i3) Color_intersection_before (i4 +) Neighbour (i4 i5) Color_intersection_before (i5 +) Move (@ i4)

Table VI

4.3 Generalization

The generalization step consists in transforming the rule which specifically applies on the example, and which contains only constants, in a more general rule which will match on many more examples and which contains variables. A constant is replaced by a variable only in some special cases, we must avoid to be too general in replacing constants by variables. We only generalize the constants that are instanciations of variables, not the 'true' constants that are also constants in the fired rules. The new general rule is given in Table VII.

If (exist (Color_intersection_before (?i1 +)) exist (Neighbour (?i2 ?i1)) exist (Neighbour (?i2 ?i3)) exist (Color_intersection_before (?i2 +)) exist (Neighbour (?i3 ?i2)) exist (Neighbour (?i3 ?i4)) exist (Color_intersection_before (?i3 O)) exist (Same_block_before (?i3 ?i3))	exist (Number_of_liberties_before (?i3 2)) exist (Neighbour (?i4 ?i3)) exist (Color_intersection_before (?i4 +)) exist (Neighbour (?i4 ?i5)) exist (Color_intersection_before (?i5 +))) Then (append (Move_before (@ ?i4 Take ?i3 GIII)))

Table VII

4.4 Conclusion on the learning method

It has been claimed that EBL is equivalent to a well known logic compilation technique : Partial Evaluation (PE) [van Harmelen 1988]. The advantage of EBL on Partial Evaluation (PE) is that EBL only creates knowledge that can be used in at least one position. However PE creates useless rules which make the matching process slower. Moreover, PE needs a lot of memory. The advantage of PE is that we do not need a training process to compile logic programs.

Another objection made to EBL is the reported negative effect of learning due to the utility problem [Minton 1988]. This effect does not appear when using our technique as we unify multiple searches in one matching process which conclusions have not to be verified after. Our rules are compiled using metaknowledge about the probability of the facts in the working memory. What can be easily done in the matching process is the sharing of conditions between rules, therefore what would have been done many times during the search process, is only done once during the matching process. Moreover, during the matching process, we do not have to evaluate all the intermediate predicates evaluated during the search process. The matching of the discovered rules is far more efficient than using a search algorithm (like alpha-beta).

This method has been tested using Go problems from books. On a set of 100 problems, it has discovered, generalized and transformed into rules all the solutions to each problem. On the set of problems used for learning, our method discovers and generalize the solutions of all the problems. Given a test set of another 100 problems, the rules created when solving the first 100 problems gives the correct answer 80% of the time on the test set, they do not find the solution 20% of the time. Discovered rules never give a wrong answer, they simply fail to give an answer 20% of the time on new problems of the same complexity.

5 Application to the learning of the management of a firm

This learning method has been applied to the learning of the management of a firm, using the formal analysis of a firm given in [Alia 1992]. This model has four hierarchical levels represented in Figure 6. Each level is related to a goal. My system learns to achieve this goal for each level.

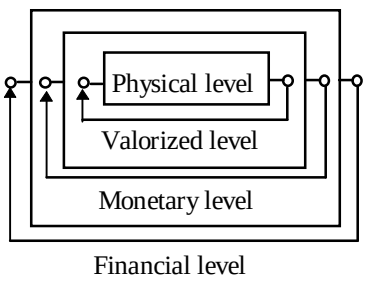

Figure 6

On the physical level, it learns to buy and to produce according to the expected sales. On the valorized level, it learns to calculate the price the product should be

sold. On the monetary level, it learns how to have a positive cash. On the financial level, it learns how to have a good return on investment. As in the Go domain theory, it is important to explicitly represent the evolution of facts over time.

6 Conclusion

I have described a method to create tactical rules in games using the combinatorial game theory and a restrictive knowledge representation, avoiding side effects and facilitating explanations and learning. I have given an example of the application of this method to the game of Go. This method works for incomplete domain theories, our Go domain theory is not complete and our theory creates reliable rules. The rules created with my method are general but not overgeneral. My system plays an entire game of Go. It is ranked 5th out of 12 in the Internet Computer Go Ladder [Pettersen 1994]. It is better than some professional hand coded Go programs. All the tactical knowledge it uses has been learned by the method presented in this paper. I am presently applying my system to other games (Chess and Abalone). I also apply it to decision making in the management of a firm.

References

Alia C. (1992). *Conception et réalisation d'un modèle didactique d'enseignement de la gestion en milieu professionnel*. Ph.D. Thesis, Montpellier II University, 1994.

Allis L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Thesis, Vrije Universitat Amsterdam, Maastricht 1994.

Berlekamp E., Conway J.H., Guy R.K. (1982). *Winning Ways*. Academic Press, London 1982.

Bouzy B. (1995). *Modelisation cognitive du joueur de Go*. Ph.D. thesis of Paris 6 University, 1995.

Conway J. (1976). *On Numbers and Games,* Academic Press, London/New-York, 1976.

Dejong G., Mooney R. (1986). *Explanation Based Learning : an alternative view.* Machine Learning 2, 1986.

Minton S. (1984). *Constraint-Based Generalization - Learning Game-Playing Plans from Single Examples*. Proceedings of the Fourth National Conference on Artificial Intelligence, 251-254. Los Altos, William Kaufmann, 1984.

Minton S. (1988). *Learning Search Control Knowledge - An Explanation Based Approach*. Kluwer Academic, Boston, 1988.

Mitchell T. M., Keller R. M., Kedar-Kabelli S. T. (1986). *Explanation-based Generalization : A unifying view*. Machine Learning 1 (1), 1986.

Müller M. (1995). *Computer Go as a Sum of Local Games : An Application of Combinatorial Game Theory*. Ph.D. thesis of the Swiss Federal Institute of Technology Zürich, 1995.

Pell B. (1993). *Logic Programming for General Game-Playing*. Proceedings of the workshop on Knowledge Compilation and Speedup Learning, at Machine Learning Conference, Amherst, Mass., 1993.

Pettersen E. (1994). *The Computer Go Ladder*. World Wide Web page: http://cgl.ucsf.edu/go/ladder.html, 1994.

Pitrat J. (1976). *Realization of a Program Learning to Find Combinations at Chess*. Computer Oriented Learning Processes, Simon J. Ed., Noordhoff, 1976.

Pitrat J. (1990). *Métaconnaissances*. Hermès, 1990.

Puget J. F. (1987). *Goal Regression with Opponent*. Progress in Machine Learning, Sigma Press, Wilmslow, 1987.

Tadepalli P. (1989). *Lazy Explanation-Based Learning: A Solution to the Intractable Theory Problem*. IJCAI 1989, pp. 694-700, 1989.

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

Wolfe D. (1991), *Mathematics of Go : Chilling Corridors*, Dissertation, University of California at Berkeley, Berkeley, 1991.