A paraître dans 'Logic Programming and Soft Computing', livre édité chez Research Studies Press (John Wiley & Sons).

Learning with Fuzzy Definitions of Goals

Tristan Cazenave

LIP6
Université Pierre et Marie Curie
4, place Jussieu
75252 PARIS CEDEX 05

email: Tristan.Cazenave@laforia.ibp.fr

Abstract

This paper explains a method to learn from fuzzy definitions of goals. This method has been applied to learn strategic rules in the Game of Go and decision rules for the management of a firm. The learning algorithm uses a representation of knowledge mainly based on the predicate logic. The goal of this paper is to extend this method of learning to systems using fuzzy logic. It is not useful to have gradual knowledge in order to learn tactical knowledge, but it becomes necessary when learning strategic knowledge. Strategic knowledge is knowledge about long term and global goals, it is fuzzy by nature. I give a method to learn using explanations of how the achievement of a gradual goal has been influenced by an action. This method is supported by an example of strategic learning in the game of Go. I show how this method can be applied in complex domains.

1 INTRODUCTION

Logic Programming provides a nice and convenient way to represent knowledge. An important goal of Logic Programming is declarativity, it involves that a logic program states *what* is to be computed, but not necessarily *how* it is to be computed. In the terminology of Kowalski's equation *algorithm = logic + control*, it involves stating the logic of an algorithm, but not necessarily the control. Giving only the logic of an algorithm is very convenient and enables to give easily a lot of knowledge to a program, however it is very inefficient and often leads to a combinatorial explosion in the application of the algorithm. Introspect [Cazenave 1996c] is a system that is designed to observe its own problem solving activity, to detect its own inefficiencies and to create automatically control rules to avoid them. Informally, it creates the control of an algorithm given the logic and some running of the algorithm on examples. This research is related to learning systems like Soar [Laird 1986], Prodigy [Minton 1989] or Theo [Cheng 1995] which learn to achieve their goals faster using some examples of problem solving, it is also related to declarative logic programming systems like Gödel [Hill 1994] or systems based on metaknowledge like Maciste [Pitrat 1990].

Many learning systems have been applied to crisp goals, the goals of the system are defined using a crisp definition: either the goal is achieved or not. This article extends the learning method to goals defined by a fuzzy measure of achievement.

A fuzzy definition of goals is necessary in complex domains where it is impossible to forecast precisely, in all cases, if a goal can be achieved. This is particularly true for the strategy in the game of Go. It is intractable to know the status of a group using a crisp definition. So we need to fuzzify the status of a group. Introspect is a system which learns to improve its problem solving abilities by observing its own problem solving activities. It learns by explaining to itself how it has deduced interesting facts about its goals [Cazenave 1996c]. It creates new rules that enable itself to deduce how to achieve its goals faster. Introspect is a general learning and problem solving system based on an extension of predicate logic, its most successful application is the learning of rules to achieve goals in the game of Go. Representing gradual knowledge is not necessary from a tactical point of view, but it becomes necessary on a strategic point of view. Fuzzy logic has already been applied to search, and especially to Chess [Junghanns 1995], but it has been used to control search. My purpose is rather to automatically create fuzzy knowledge bases of rules.

In a first part, I show why a fuzzy knowledge representation of goals is adapted to represent the achievement of long term goals in complex domains, and especially to represent the strategic knowledge of Go players. In a second part, I explain how this fuzzy knowledge can be used by a self fuzzy learning system to develop itself from a small set of initial rules. The following parts detail the different steps of the learning algorithm when applied to gradual goals. I finish with the description of the applications of my system to the game of Go and to the management of a firm.

2 Fuzzy definition of goals

2.1 A simple example

The goal of my system is to forecast efficiently in the long term the consequences of its actions on the achievement of its goals. In complex domains it is intractable to calculate the long term consequences of the actions. So we need to express the intractable goal in terms of more tractable goals. I will illustrate this using a simple example: suppose that the system is connected to a robot arm in the cube world. The goal of the system is to build a high tower of red cubes. It only has some rules that tell it the direct

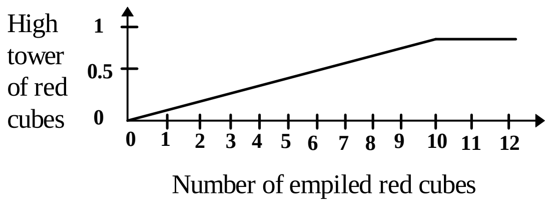

Fig 1 - A fuzzy set representing the gradual achievement of the goal 'High tower of red cubes'

consequences of its action. It is tractable for the system to solve the problem of taking a red cube in its arm, but it is intractable to directly find all the moves that will enable it to build a high tower of red cubes. Therefore, the system will be given a fuzzy definition of a high tower of red cubes, and will achieve its goals by putting the red cubes one by one, achieving gradually the overall goal, by breaking it into subgoals.

This fuzzy definition of a goal is directly expressed in a logic programming language similar to Prolog. This is a typed language which has meta facilities and which enables the use of integer and real numbers. In this article, variables are represented by a single letter or by a letter followed by a number. It has built-in functions that operate on integer and real numbers, such as add, sub, div, mult, equal, greater_than. The fuzzy definition of a goal is directly written in this language. The Fig 1 is represented with the following rule:

```
High_tower_of_red_cubes ( n f ) :- Number_of_empiled_red_cubes ( n h )
                                   equal ( f1 div ( h 12 ) )
                                   equal ( f min ( f1 1.0 ) ).
```

In this rule, n is an integer which represents the number of actions required to reach the state described. The fuzzy number associated to the achievement of the goal is f, it is between 0.0 and 1.0.

2.2 Strategy in the game of Go

Strategic knowledge in games are about long term goals. In games such as Chess and Go, the high number of possible moves makes it impossible to forecast in the long term the consequences of the moves played. A solution to this problem is to have a gradual achievement of long term goals. It enables to know if a move makes the goal easier or harder to achieve.

This is particularly true for the strategy in the game of Go. The ultimate goal of a player is to make live the more stone on the board. However, in the middle game, most of the groups of stones are in an uncertain state, and the evolution of this state cannot be precisely foreseen. It is very useful in such a case to have a fuzzy evaluation of their states and of the evolution of this state when playing different moves.

Definition 1: A group of stones is a set of stones of the same color which cannot be disconnected.

Stones of the same group have the same number in Fig 2.

Definition 2: A friend intersections of a group is an empty intersection that can be connected to the group whatever the opponent plays, moreover, this empty intersection must not be connectable to a living opponent group.

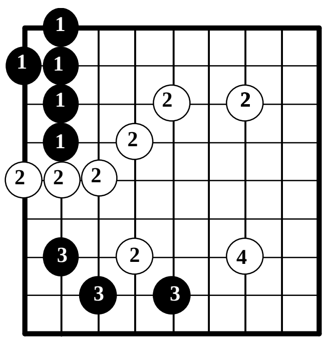

Fig 2 - A Go board with groups marked with the same numbers

In Fig 3, the white friend intersections are filled with a small white point. The black friend intersections are filled with a small black point. The intersections connectable both to a white and a black group are filled with a small gray point. Each group owns a set of friend intersections of its own color.

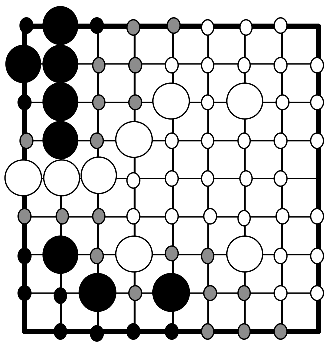
Fig 3 - Go board with empty friend intersections marked

The number of friend intersections of a group is a very good heuristic to approximate the degree of life of a group. For example, the group marked with 2 in Fig 2 has more than twelve friend intersections, it will therefore have no problems to live. Whereas the group marked with 3 in Fig 2 has only 7 friend intersections, it is not completely alive and may have some problems. Its degree of life is around 0.5. Two rules define the degree of life of a group given its number of friend intersections:

```
Degree_of_life ( n g f ) :- Number_of_friend_intersections ( n g h )
                            greater_than ( h 3 )
                            equal ( f1 div ( sub ( h 3 ) 9 ) )
                            equal ( f min ( f1 1.0 ) )

Degree_of_life ( n g f ) :- Number_of_friend_intersections ( n g h )
                            greater_than ( 4 h )
                            equal ( f 0.0 ).
```

After these rules have been fired, one rule chooses the greatest of all the degrees of life:

```
Degree_of_life ( n g f ) retract (Degree_of_life ( n g f2 ) ) :-
                            Degree_of_life ( n g f )
                            Degree_of_life ( n g f2 )
                            greater_than ( f f2 ).
```

The fuzzy degree of life is given by the real number f, the group is represented by the variable g, and the integer n is the number of moves to play to achieve this degree of life.

The Fig 4 gives the graphical representation of the fuzzy set defined by the rules above.

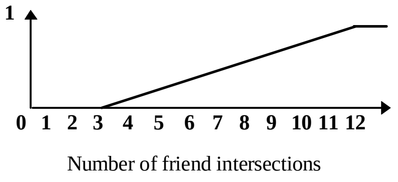

Fig 4 - The fuzzy set defining the degree of life, given the number of friend intersections

Note that the system uses a forward chaining algorithm, and that when we set the value of the degree of life, the system checks if this degree is greater than the previously established degree for the same group. This is due to the fact that there may be many rules that give a conclusion on the degree of life of a group. The convention is to create fuzzy representations of the achievement of a goal that never overestimate the degree of achievement. Thus, if according to one criterion, the goal is poorly achieved, but that according to another criterion the goal is almost achieved, the system will conclude that the goal is almost achieved. This is compatible with the disjunctive normal form of logic programs. This can be viewed as taking the max operator as the t-conorm used to make the fuzzy union between two disjunctive rules concluding on the same predicate.

There are many attributes for a group. Table 1 gives a list of the predicates used in my system to describe a group.

Each of these attributes contributes to the final goal of the game which is to make the group live. These contributions are less or more graduals. They are represented in Fig 4, 5 and 6. The vertical axis always represents the degree of life of the group, between 0 and 1.

Number_of_won_life_bases
Number_of_unsettled_life_bases
Number_of_won_eyes
Number_of_unsettled_eyes
Number_of_friend_intersections
Number_of_stones
Number_of_connections_to_living_friends

Table 1 - List of predicates used to measure the degree of life of a group

The system also uses crisp definitions of the degree of life. The crisp value is always preferred to the fuzzy value, but the most interesting strategic rules are the rules that use a fuzzy definition. The Fig 5 represents the most simple of the crisp definition of the achievement of the goal.

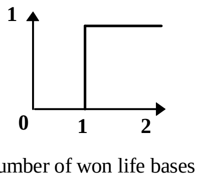

Fig 5 - Life can be given a crisp definition. But this definition cannot always be applied. That is why we need fuzzy rules.

The Fig 6 give some examples of some simple fuzzy definitions of the goal "Degree_of_life", other definitions that combine the different predicates are also used in the system. But only the simple and easily understandable rules are presented here.

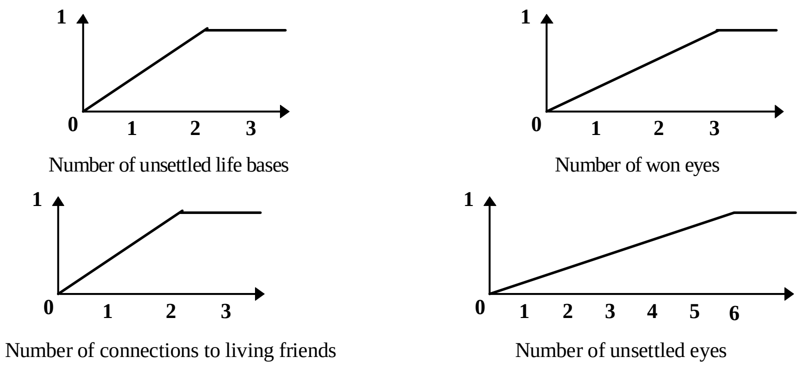

Fig 6 - Some simple fuzzy sets defining the gradual achievement of the goal Degree_of_life.

Table 2 gives an evaluation of the attributes for the four groups of Fig 2.

Attributes\Groups	1	2	3	4
Number of won life bases	0	0	0	0
Number of unsettled life bases	1	0	0	0
Number of won eyes	1	0	0	0
Number of unsettled eyes	1	0	0	0
Number of friend intersections	3	26	7	11
Number of stones	5	7	3	1
Number of connections to living friends	0	0	0	2

Table 2

Table 3 gives the degrees of life corresponding to each attribute for each group and also gives the final degree of life for the groups. This degrees of life were calculated using only the fuzzy definition of achievement of the goal.

Attributes\Groups	1	2	3	4
Number of won life bases	0	0	0	0
Number of unsettled life bases	0.5	0	0	0
Number of won eyes	0.33	0	0	0
Number of unsettled eyes	0.16	0	0	0
Number of friend intersections	0	1	0.44	0.89
Number of connections to living friends	0	0	0	1
Degree of live of the group	0.5	1	0.44	1

Table 3

The learning system will use the simple definition of a goal to learn to forecast the consequences of its moves. It will create more complex rules that will conclude on more long term results than the rules defining the current achievement of the goal.

3 OVERVIEW OF THE LEARNING ALGORITHM

The learning algorithm is composed of six steps. The first step consists in solving a problem using a declarative logic program. After this problem solving episode, the learning system explores its own problem solving performances so as to find possible inefficiencies. This second phase is called introspection, it selects a goal which have been inefficiently deduced by the logic program. The third phase is the explanation of how this goal has been deduced, it finds the reasons why a goal can be deduced. The explanation results in a rule to deduce directly the goal, this rule contains only constants. So as to learn general rules, the next phase is generalization which consists in replacing some appropriates constants by

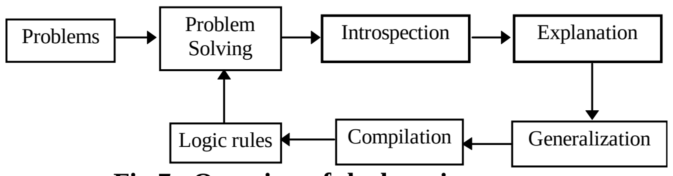

Fig 7 - Overview of the learning system

variables. The result of generalization is a rule in predicate logic, this rule may have a bad ordering of conditions, that is why it is compiled by a set of metarules which reorder the conditions so as to match the rule much faster [Cazenave 1996a]. The rules created by the system are used to learn other rules. The systems bootstraps itself using a small set of initial rules.

4 DEDUCTION

The goal of the deduction part of the learning system is to deduce the degree_of_life of a group after a move. At the beginning of the deduction, the system only has facts that describe the state of the groups at time t and a move at time t. It uses its rules about the consequences of a move to deduce the state of the groups at time t+1 after the move. For example, the rule:

```
Number_of_friend_intersections ( t g n ) :-
                        Number_of_friend_intersections ( t1 g n1 )
                        Move ( i c )
                        Color ( g c )
                        add_friend_intersections ( t1 i g n2 )
                        equal ( t add ( t1 1 ) )
                        equal ( n add ( n1 n2 ) ).
```

is instanciated by the deduction process into the rule:

```
Number_of_friend_intersections ( 1 group1 10 ) :-
                        Number_of_friend_intersections ( 0 group1 3 )
                        Move ( intersection58 Black )
                        Color ( group1 Black )
                        add_friend_intersections ( 0 intersection58 group1 7 )
                        equal ( 1 add ( 0 1 ) )
                        equal ( 10 add ( 3 7 ) ).
```

This rules gives the Number_of_friend_intersections after the move at time 1, using the Number_of_friend_intersections before the move at time 0. After deducing all the predicates describing the state of the board after the move, the system can deduce the state of achivement of its goals after the move. The rule giving the Degree_of_life using the Number_of_friend_intersections is fired and its instanciation results in the following instanciated rule:

```
Degree_of_life ( 1 group1 0.78 ) :-
                        Number_of_friend_intersections ( 1 group1 10 )
                        greater_than ( 10 3 )
                        equal ( 0.78 div ( sub ( 10 3 ) 9 ) )
                        equal ( 0.78 min ( 0.78 1.0 ) )
```

According to this rule, the degree of life of the group1 after the move (at time 1) is 0.78. A lot of other rules are used to deduce the state of the board and the degrees of life after the move, but we will mainly use this simple example to explain the learning process.

5 INTROSPECTION

The introspection module is dedicated to find inefficiencies of the deduction module. Introspection decides what is interesting to learn so as to repair observed inefficiencies.

To select interesting facts, the system compares the degree of achievement of the goal to learn before the move and after the move. If the degree of achievement after the move is greater than the one previously anticipated by the rules of the current knowledge base, then the fact describing the greater degree of achievement is interesting to explain so as to create a new rule which will enable to deduce it directly, avoiding a possibly long deduction process.

```
Explain ( Degree_of_life ( n g f ) ) :-
                              Anticipated_degree_of_life ( n g f1 )
                              Degree_of_life ( n g f )
                              greater_than ( f f1 )
```

This (meta)rule tells the system to explain a deduced degree of life, if it is greater than the previously anticipated degree of life.

6 EXPLANATION

The explanation consists in giving the reasons why a goal was deduced. The explanation module goes back into the problem solving trace, replacing an instanciated condition in an instanciated rule, by the conditions of the instanciated rule that has been used to deduce the replaced condition.

```
Degree_of_life ( 1 group1 0.78 ) :-
                              Number_of_friend_intersections ( 0 group1 3 )
                              Move ( intersection58 Black )
                              Color ( group1 Black )
                              add_friend_intersections ( 0 intersection58 group1 7 )
                              equal ( 1 add ( 0 1 ) )
                              equal ( 10 add ( 3 7 ) ).
                              greater_than ( 10 3 )
                              equal ( 0.78 div ( sub ( 10 3 ) 9 ) )
                              equal ( 0.78 min ( 0.78 1.0 ) )
```

In our example, the result of the explanation is the rule above. To obtain it, the module replaces the condition Number_of_friend_intersections (1 group1 10) in the last rule of section 4, by the conditions of the second rule of section 4.

Usually, the system replaces more than one condition, and conditions in the replaced rules are themselves replaced by other lists of conditions. Sometimes, there are many rules that conclude on the same condition. It leads to as many different explanations, and as many branches in the explanation tree. The explanation of the deduction of an interesting goal can lead to a lot of explanation rules.

7 GENERALIZATION

When the explanation is done, we can generalize the resulting rules to allow them to apply in many more case. The main mechanism of generalization is the replacement of instanciated variables by constants. The generalized explanation of our example rule gives:

```
Degree_of_life ( t g f ) :-  Number_of_friend_intersections ( t1 g n1 )
                             Move ( i c )
                             Color ( g c )
                             add_friend_intersections ( t1 i g n2 )
                             equal ( t add ( t1 1 ) )
                             equal ( n add ( n1 n2 ) )
                             greater_than ( n 3 )
                             equal ( f1 div ( sub ( n 3 ) 9 ) )
                             equal ( f min ( f1 1.0 ) )
```

Replacing only instantiated variables and not constants is very important. It allows to create better rules. In the example rule, it is very important to have the variable n in the condition greater_than (n 3), but it is also very important that 3 stays as a constant.

This generalized explanation gives a new strategic rule. This strategic rule is very general and can be applied in many more boards than the example board on which it was learned.

8 COMPILATION

8.1 Reordering premises

A good ordering of conditions can provide big speedups in production systems [Ishida 1988]. To reorder conditions in our learned rules, we use a very simple and efficient algorithm. It is based on the estimated number of following nodes the firing of a condition will create in the semi-unification tree. Here are two metarules used to reorder conditions of the learned rules:

```
Branching ( r connect (v v1 v2 v3 ) 1.5 ) :-  Rule ( r )
                                              Condition ( r Connect ( v v1 v2 v3 ) )
                                              Not_instantiated ( v )
                                              Not_instantiated ( v1 )
                                              Instantiated ( v2 )
                                              Not_instantiated ( v3 )
```

```
Branching ( r add_friend_intersections (v v1 v2 v3 ) 250 ) :-
                        Rule ( r )
                        Condition ( r add_friend_intersections ( v v1 v2 v3 ) )
                        Not_instantiated ( v )
                        Not_instantiated ( v1 )
                        Not_instantiated ( v2 )
                        Not_instantiated ( v3 )
```

A metarule evaluates the branching factor of a condition based on the estimated mean number of facts corresponding to the condition in the working memory. Metarules are fired each time the system has to give a branching estimation for all the conditions left to be ordered. When reordering a rule containing N conditions, the metarule will be fired N times: the first time to choose the condition to put at first in the rule, and at time number I to choose the condition to put in the Ith place. The first condition Rule (r) instanciates in the variable r all the rules of the set of learned rules to reorder. The second condition, Condition (r Connect (v v1 v2 v3)), instanciates the metavariables v, v1, v2 and v3 on all the rules which contain the condition Connect (v v1 v2 v3). The third condition Not_instantiated (v), verifies if the variable contained in v has not already been instanciated in the previous conditions of the rule r. The instanciations of the variables contained in v1 and v3 are a potential cause of branching. In conclusion, the metarule estimates the branching factor to be 1.5.

The branching factors of all the reordering conditions are compared and the condition chosen is the one with the lowest branching factor. The algorithm is very efficient, it orders rules better than humans do and it runs in less than one minute even for rules containing more than 200 conditions.

The two following rules gives an example of the difference in the number of instanciations and tests between a bad ordered and a well ordered rule. Each condition is followed by the number of instanciations it has required. For big rules (some of our learned rules for the game of Go contain more than 200 conditions), the ordering of conditions can lead to 14.000 times less instanciations and tests than for non ordered rules [Cazenave 1996a]. In our example, the bad ordered rule has a cost (13570) 68 times higher than the cost (200) of rule ordered by the system using the metarules of compilation.

```
Degree_of_life ( t g f ) :-  Color ( g1 c )                                   10
                             add_friend_intersections ( t1 i g n )          2500
                             Number_of_friend_intersections ( t1 g n1 )     2500
                             Number_of_friend_intersections ( t1 g1 n2 )    2500
                             Color ( g1 c )                                 1500
                             equal ( t add ( t1 1 ) )                       1500
                             equal ( n add ( n1 add ( n n2 ) ) )            1500
                             greater_than ( n 3 )                           1500
                             Connect ( t1 i g g1 )                            20
                             equal ( f1 div ( sub ( n 3 ) 9 ) )               20
                             equal ( f min ( f1 1.0 ) )                       20
```

```
Degree_of_life ( t g f ) :-  Connect ( t1 i g g1 )                              20
                             add_friend_intersections ( t1 i g n )               20
                             Number_of_friend_intersections ( t1 g n1 )          20
                             Number_of_friend_intersections ( t1 g1 n2 )         20
                             Move ( i c ) Color ( g c ) Color ( g1 c )           20
                             equal ( t add ( t1 1 ) )                            20
                             equal ( n add ( n1 add ( n n2 ) ) )                 20
                             greater_than ( n 3 )                                20
                             equal ( f1 div ( sub ( n 3 ) 9 ) )                  20
                             equal ( f min ( f1 1.0 ) )                          20
```

8.2 Ordering rules.

The system always chooses the rule which concludes on the highest degree of achievement. Therefore, we can order the firing of the rules so as to stop firing rules as soon as a conclusion has been deduced. The system begins with rules concluding on the highest degree of achievement of the goal, and decrease until the rule concluding on the lowest one.

9 APPLICATION TO THE GAME OF GO

This section describes the application of the strategic learning system to the game of Go. It explains why it is the most complex game. It briefly describes how are made actual Go program and stresses the interest of the game of Go for machine learning. The architecture of the Go playing system using fuzzy definitions of its goals is given.

9.1 Complexity of Go

Go was developed three to four millennia ago in China; it is the oldest and one of the most popular board game in the world. Like chess, it is a deterministic, perfect information, zero-sum game of strategy between two players. The board includes 19 vertical lines and 19 horizontal lines which give 361 *intersections*. At the beginning the board is empty. Each player (Black or White) adds in turn one *stone* on an empty intersection. Two adjacent stones of the same color are *connected* and they are part of the same *string*. Empty adjacent intersections of a string are the *liberties* of the string. When a move fills the last liberty of a string, this string is removed from the board. Repetitions of positions are forbidden. According to the possibility of being captured or not, the strings may be *dead* or *alive*. A player *controls* an intersection either when he has an alive stone on it, or when the intersection is empty but adjacent to alive stones. The aim of the game is to control more intersections than the opponent. The game ends when both players pass.

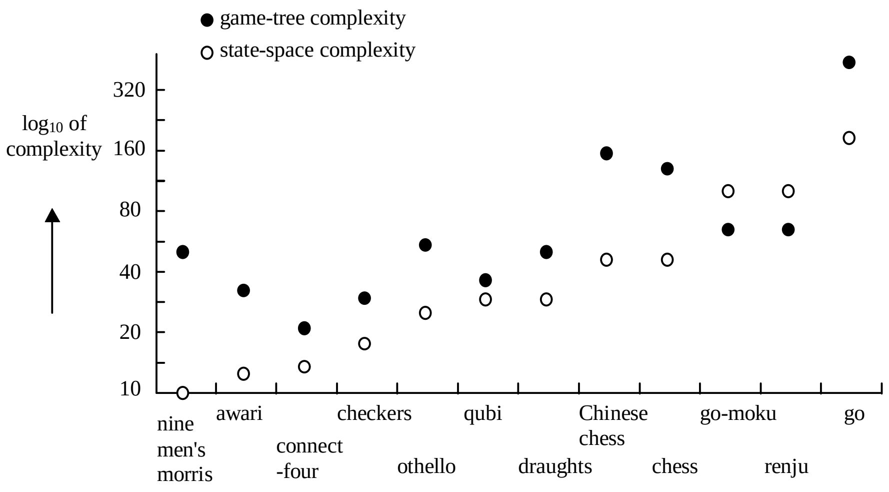

Fig 8 - Relative complexities of the games of the Olympic list [Allis 1994]

In spite of the simplicity of its rules, playing the game of Go is a very complex task. [Robson 1983] proved that Go generalized to NxN boards is *exponential in time*. More concretely, [Allis 1994] defines the *whole game tree complexity* A. Considering the average length of actual games L and average branching factor B, we have $A = B^L$. The *state-space complexity* of a game is defined as the number of *legal* game positions reachable from the initial position of the game. In Go, $L \approx 150$ and $B \approx 250$ hence the game tree complexity $A \approx 10^{360}$. Go state space complexity, bounded by $3^{361} \approx 10^{172}$, is far larger than that of any other perfect-information game of the Olympic list. Fig 8 resumes the information on the estimated complexities for the perfect information games of the Olympic list. A specificity of Go is that the end of a game is decided by mutual agreement, there is no rule defining the end of the game, knowing the game has ended requires expert knowledge. Moreover, a position is very difficult to judge, on the contrary of chess where a good heuristic for evaluating a position is the material balance. This makes Go the most complex perfect information game.

The best Go playing program in the world is Handtalk. Its level may be the one of a low-ranked Go club player, about 8 or 10 *kyu.* A complete novice is about 30 kyu, a beginner quickly reaches 20 kyu, a strong player is 1 kyu and then 1 dan until 9 dan for the strongest players in the world.

9.2 Methods for programming Go

As it is impossible to search the entire tree for the game of Go, the best Go playing programs rely on a knowledge intensive approach. They are generally divided in two modules:

- a tactical module that develops narrow and deep search trees. Each tree is related to the achievement of a subgoal of the game of Go.
- a strategic module which chooses the move to play according to the results of the tactical module.

A Go expert uses a great number of rules. Go programmers usually try to enter by hand these rules in a Go program. Creating this large number of rules requires a high level of expertise, a lot of time and a long process of trial and errors. Moreover, even the people who are expert in Go and in programming find it difficult to design these rules. This phenomenon can be explained by the high level of specialization of these rules: once the expert has acquired them, they become unconscious and it is hard and painful for the expert to explain why he has chosen to consider a move rather than another one. Moreover, even when the work of extracting some rules has been done, it results in thousands of specific expert rules. Thus, it is difficult to describe them in a synthetic way.

9.3 Computer Go and Machine Learning can benefit from each other

The difficulty of encoding Go knowledge is the consequence of a well known difficulty of expert system development: the knowledge engineering bottleneck. The goal of machine learning is to avoid this bottleneck by replacing the knowledge extraction process with an automated construction of knowledge based on examples of problem solving. Machine learning techniques enable Go programmers to get rid of the painful expert knowledge acquisition. Thus, computer Go is an ideal domain to test the efficiency of the various machine learning techniques.

9.4 Using the learned rules in a Go program

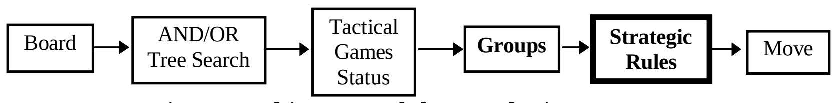

Fig 9 - Architecture of the Go playing program

The tactical part of the Go playing program develops AND/OR tree searches to calculate the states of tactical games. Each tactical game corresponds to a simple crisp subgoal of the game of Go. The tactical games status are used to create the groups and to fill the predicates used by the strategic module. Our Go program develops approximately 1000 proof tree searches on a position. These proof trees contains between 2 and 600 nodes. Then the program fires the learned strategic fuzzy rules that give it the degree of life of each group and its evolution after each interesting move. This information is used to choose the best move. The best move is chosen by evaluating the difference of the board value after and before each move. The best move is the move that has the highest difference.

To evaluate the value of the board, the system has to evaluate the degree of life and the importance of each group. The importance of the example groups are given in Table 3. The importance of a group is the evaluation of the difference of points at the end of the game between the life of the group and its death. It is calculated using the following rule:

```
Importance ( g n ) :-    Number_of_stone ( g n1 )
                         Number_of_friend_intersections ( g n2 )
                         Number_of_shared_friend_intersections ( g n3 )
                         equal ( n add ( add ( add ( n1 n1 ) n2 ) n3 ) )
```

Groups	1	2	3	4
Importance of the group	24	80	32	31

When the importances and the degrees of life of the groups have been computed, the system can evaluate a Go board:

$$\text{Evaluation} = \sum_i (\text{Degree}_i * \text{Importance}_i) - \sum_j (\text{Degree}_j * \text{Importance}_j)$$

with $i \in$ Friends Groups and $j \in$ Opponent Groups.

In the example of Fig 2, if black is the friend color, the evaluation of the position gives:

Evaluation=0.5*23+0.44*32-1.0*80-1.0*31=-85.4

This evaluation means that black is probably going to lose the game by 43 points. This analysis is compatible with the analysis of Go expert players. This evaluation function has been tested on numerous Go boards and it gives a good approximation of the evaluation of a position.

The two moves we are examining in the board of Fig 10 are the black moves in i28 and i59. Table 4 gives the outcomes of the black move in i28 and Table 5 gives the outcomes of the black move in i59.

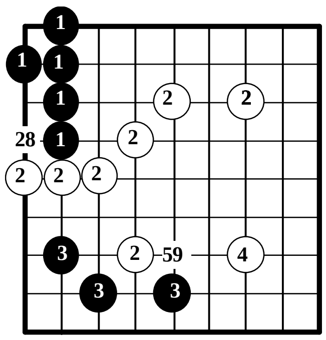

Fig 10 - The two best moves found by the system

Attributes\Groups	1	2	3	4
Number of won life bases	+1	0	0	0
Number of unsettled life bases	-1	0	0	0
Number of won eyes	+1	0	0	0
Number of unsettled eyes	-1	0	0	0
Number of friend intersections	0	0	0	0
Number of connections to living friends	0	0	0	0

Table 4

Attributes\Groups	1	2	3	4
Number of won life bases	0	0	0	0
Number of unsettled life bases	0	0	0	0
Number of won eyes	0	0	0	0
Number of unsettled eyes	0	0	+1	0
Number of friend intersections	0	-4	0	-1
Number of connections to living friends	0	0	0	-1

Table 5

If the board is evaluated after the two black moves, there is a variation of +12 points for the black move in i28 and a variation of +11 points for the black move in i59. The system will choose the black move in i28.

9.5 Results in international competition

Our learning system has been trained using one hundred beginners problems. It has learned 1000 general rules on these problems. The resulting Go program plays a move in 10 seconds on a Pentium 133 MHz, it is one of the fastest programs. It has beaten the best Japanese program in the 1996 FOST cup (the 1997 FOST cup will be held during IJCAI97). It is in the group of programs following the best four commercial programs. Moreover, it is the best symbolic learning Go program.

10 APPLICATION TO THE MANAGEMENT OF A FIRM

This learning method has been applied to the learning of the management of a firm [Cazenave 1996b], using the formal analysis of a firm given in [Alia 1992]. This model has four hierarchical levels represented in Fig 11. Each level is related to a goal. My system learns to achieve a goal for each level.

On the physical level, it learns to buy and to produce according to the expected sales. I give below some rules of the domain theory of the Physical level (MP stands for Manufactured Products). The system learns to set the value of the variables n and n1 in the predicates Quantity_Products_Bought (t n) and Quantity_Work (t n1).

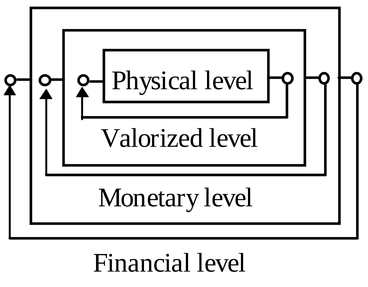

Fig 11 - A simple hierarchical model of the goals used to manage a firm

Stock_MP_Before_Sale (t1 n5) :- Stock_MP_After_Sale (t n)

```
                              Stock_Products ( t n1 )
                              Quantity_Products_Bought ( t n2 )
                              equal ( n3 sum ( n1 n2 ) )
                              Quantity_Work ( t n4 )
                              greater_than ( n3 n4 )
                              equal ( n5 sum ( [ n n3 n4 ] )
                              equal ( t1 sum ( t 1 ) )

Stock_MP_After_Sale ( t n2 ) :-   Stock_MP_Before_Sale ( t n )
                              Quantity_Sold ( t n1 )
                              greater_than ( n n1 )
                              equal ( n2 sub ( n n1 ) )
```

On the valorized level, it learns to calculate the price the product should be sold. The system learns to set the value of p in the predicate Sell_Price (t p).

On the monetary level, it learns how to have a positive cash. This is a crisp goal.

On the financial level, it learns how to have a good return on investment. This is a fuzzy goal. It is represented by the fuzzy set of Fig 12. This is the level which is the closest in spirit to the strategic level of the Go program.

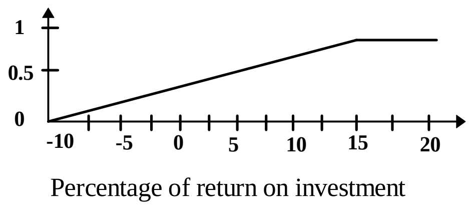

Fig 12 - A fuzzy set describing the goal good return on investment.

11 Conclusion

I have described a method to automatically create strategic fuzzy rules in the game of Go and in the management of a firm. This method can be used to bootstrap a large base of fuzzy rules beginning with a small set of rules. It creates a large set of valid, useful and general rules using only the simple definition of the strategic goals of the system. The system uses strategic fuzzy rules and plays the game of Go to an international level [Pettersen 1994]. This learning algorithm can be applied to other domains than the game of Go. An example of its application to the learning of the management of a firm has been given. It is adapted to very complex domains where the important goals are better represented using gradual knowledge. In domains where it is impossible to compute directly if a goal is achievable because of the combinatorial explosion of the search.

References

[Alia 1992] - C. Alia. *Conception et réalisation d’un modèle didactique d’enseignement de la gestion en milieu professionnel.* Ph.D. Thesis, Montpellier II University, 1992.

[Allis 1994] - L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*, Ph.D. Thesis, Vrije Universitat Amsterdam, Maastricht, September 1994.

[Cazenave 1996a] T. Cazenave, *Automatic Ordering of Predicates by Metarules*. Proceedings of the 5th International Workshop on Metareasonning and Metaprogramming in Logic, Bonn, 1996.

[Cazenave 1996b] T. Cazenave, *Learning to Manage a Firm*. International Conference on Industrial Engineering Applications and Practice, Houston, 1996.

[Cazenave 1996c] - T. Cazenave. *Système d'Apprentissage par Auto-Observation. Application au Jeu de Go.* Ph.D. Thesis, Université Pierre et Marie Curie, Paris 6, 1996.

[Cheng 1995] - J. Cheng. *Management of Speedup Mechanisms in Learning Architectures*. Ph. D. Thesis, Carnegie Mellon University, Pittsburgh, January 1995.

[Hill 1994] - P. M. Hill, J. W. Lloyd. *The Gödel Programming Language.* MIT Press, Cambridge, Mass., 1994.

[Ishida 1988] - T. Ishida. *Optimizing Rules in Production System Programs*. AAAI-88, pp. 699-704, 1988.

[Junghanns 1995] - A. Junghanns. *Search with Fuzzy Numbers.* 4th IEEE International Conference on Fuzzy Systems, Yokohama, Japan, 1995.

[Laird 1986] - J. Laird, P. Rosenbloom, A. Newell. *Chunking in SOAR : An Anatomy of a General Learning Mechanism*. Machine Learning 1 (1), 1986.

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

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

[Pitrat 1990] - J. Pitrat. *Métaconnaissances.* Hermès, France, 1990.

[Robson 1983] - J. M. Robson. *The Complexity of Go* - Proceedings IFIP - pp. 413-417 - 1983.