Proceedings de IPMU'98

Speedup Mechanisms for Large Learning Systems

Tristan Cazenave
LIP6, case 169, UPMC, 4, place Jussieu
75252 PARIS CEDEX 05, FRANCE
Tristan.Cazenave@lip6.fr

Abstract

Eliminating combinatorics from the match in production systems is important for expert systems, real-time performance, machine learning, parallel implementation and cognitive modeling. We describe a way of managing the tradeoff between generality and efficiency in knowledge representation for large learning systems. We propose an architecture that enables to combine efficiency in problem solving to generality in learning. Our architecture combines generality and efficiency by using two problem solvers. The first one is interpreted and uses a general knowledge representation. It enables the system to learn general rules. The second one is compiled and uses a specialized knowledge representation. It enables the system to solve problems rapidly and to detect when learning can occur in order to decide to call the first problem solver. To speedup rules, we use two mechanisms which do not affect the generality of learned rules and three mechanisms that alter the learning abilities of the system and that are only used in the second problem solver. This approach has shown its efficiency in its application to the game of Go. The game of Go is the most complex two person complete information game.

1 Introduction

Solving a problem in order to learn to solve similar problems faster is different from solving a problem in order to solve it quickly. As already pointed out in [36] and in [7], generality in knowledge representation is often opposed to efficiency of learned rules. In this paper, we propose a system architecture that enables the system to combine generality of learning and efficiency of problem solving. We also propose different speed-up mechanisms that are used successfully in this architecture.

Explanation-Based Generalization [27] and Explanation-based Learning [8] are powerful learning methods for domains with an underlying theory. The use of similar methods to learn in well-defined domains can be traced back to [31] and [21]. Well-known planning and learning systems as Soar [18], Prodigy [22] and Theo [28] use these methods. Unfortunately, learned knowledge can hurt performance [22], this is known as the utility problem. Some reports showed that in some systems, learning degrades problem solving performance [10,35].

One approach to this problem is to use some form of selective learning or forgetting. [20] provides a general framework for analyzing this approach. Examples include discarding learned rules if they turn out to cause overall system slowdown [22], disabling the learning component after some desired or peak performance level has been reached [16], learning only certain types of rules (non recursive) that are expected to have low match cost [12], and employing statistical approaches to ensure that only rules that improve performance are added to the knowledge base [14,15].

Unfortunately, this approach alone is inadequate because it enforces the system to learn only a few number of rules and reduces the gain of learning. However, it can be complemented by another approach to reducing match cost, enabling the system to learn more rules before reaching its maximum. Many techniques have been developed for that. [36] and [37] prevent the formation of expensive rules that have a combinatorial match cost by restricting the representation a system uses. Prodigy reduces match cost by simplifying the conditions of learned rules using a compression module [22]. Static [11] and Dynamic [30] analyze the structure of a problem space to build simpler rules with lower match cost than Prodigy/EBL. [7] generalizes or specializes the conditions of search

control rules so as to reduce their match cost. [19] uses statistical information on program runs to dynamically unfold and reorder clauses of a logic program. Researchers in production systems have also devised methods to efficiently match production rules. [17] uses some heuristic rules to order the conditions of rules. Rete networks [13] have been enhanced by [9] to support large production systems. In [29], a production system learns parameters to be more efficient. [9] also notes that the estimation of the utility of a learning system is highly dependent on the efficiency of its matching part.

Our learning system is composed of two problem solvers. The first one is used to learn new rules, it is interpreted and composed of general rules. The second one is used to solve problems using limited time resources, and to decide when to learn new rules. It is compiled and composed of specific and efficient rules but it has the same knowledge as the first problem solver.

The generality of learning associated to the efficiency of speeded-up learned rules has lead our system to create a Go program that is better than most of the hand-coded Go programs.

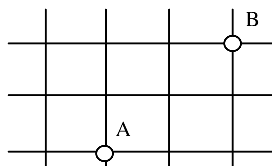

Figure 1

For the sake of simplicity, we will adopt a simplified representation of rules, so as to make it easy for the reader to understand them. Moreover, we will give examples using the grid task described in [36]. The grid task consists in finding a path of length four between point A and point B in the figure 1. This task is easier to understand than the game of Go which is our principal application. A Go board is also a grid, and all the mechanisms described in this paper also apply to the rules learned in the application of our learning system to the game of Go.

The speedup mechanisms used in our learning system can be divided into two categories. The mechanisms of the first category do not modify the generality of learned rules, and those of the second category trade generality against efficiency. In Section 2, we begin with a description of two speedup mechanisms which do not modify the generality of learned rules. In Section 3, we follow with three other mechanisms which alter generality and the ability of the system to observe itself. Then we give the architecture of our learning system, this architecture enables us to combine generality and efficiency in the same system. The Section 4 describes the main application of our system, the game of Go.

2 Speedup mechanisms not modifying Learning

In this section, we present two mechanisms used to speedup problem solving. The first one is the reordering of the conditions of a rule. The second one is the deletion of some useless conditions of the learned rules. These two mechanisms do not modify the generality of learning. They are applied to modify the rules of our two problem solvers.

2.1 Reordering conditions

In [19], statistics on several runs of a program are used to reorder and to unfold clauses of this program. [17] also dynamically uses some simple heuristics to find a good ordering of conditions for a production system. Our approach is somewhat different, it takes examples of working memories to create metarules that will be used to reorder the clauses. A metaprogram is automatically created to reorder the clauses, we do not dynamically reorder conditions of the rules. One advantage is that we can create this metaprogram independently. Moreover, once the metaprogram is created, running it to reorder learned rules is faster than dynamically optimizing the learned rules. This feature is important for systems like Introspect [4] that learn a large number of rules. The creation of the metaprogram is also fast.

We rely on the assumption that domain-dependent information can enhance problem solving [25]. This assumption is given experimental evidence in [26]. On the contrary of Minton, we do not specialize heuristics on specific problems instances, we rather create metaprograms according to specific distributions of working memories.

Reordering conditions is important for the performance of learned rules. A simple example that shows this importance is the two following clauses

that give the same results but that do not have the same efficiency :

actor (X) :- brother (X, X1), minister_of (X, DOMAIN).

actor (X) :- minister_of (X, DOMAIN), brother (X, X1).

Reordering based only on the number of instanciated variables in a predicate does not work for the above rule. In the constraint domain literature, constraints are reordered according to two heuristics concerning the variables to instanciate [26] : the range of values of the variables and the number of other variables it is linked to. These heuristics dynamically choose the order of constraints. But to do so, they have to keep the number of possible instanciations for each variable, and to lose time when dynamically choosing the variable. These lost of time is justified in the domain of constraints solving because the range of values of a variable, affects a lot the efficiency, and can change a lot from one problem to another. It is not justified in some other domains where the range of values a variable can take is more stable. We have chosen to do the choices statically by reordering once for all and not dynamically at each match because it saves more time in the domains in which we have tested our approach.

To reorder conditions in our learned rules, we use a 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. An example of metarule is given in Figure 2.

```
branching ( R, neighbor ( V, V1), 3.76 ) :-
        rule ( R ),
        condition_to_order ( R, connected ( V, V1 ) ),
        instanciated ( V ),
        not_instanciated ( V1 ).
```

Figure 2

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 T to choose the condition to put in the Tth 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_to_order (R, Connected (V, V1))', instanciates the metavariables V and V1 on two variables of type intersection. The metavariables are instantiated in all the rules that contain a condition matching 'neighbor (V, V1)', if this condition has not been ordered yet. The third condition 'instanciated (V)', verifies that the variable contained in V has been instanciated in the previous conditions of the rule R. The fourth condition 'not_instanciated (V1)', verifies that the variable contained in V1 has not been instanciated in the previous conditions of the rule R. The instanciations of the variable contained in V1 is therefore a potential cause of branching. In conclusion, the metarule estimates the branching factor to be 3.76 (this is the mean number of neighbor intersections of an intersection on a 19*19 grid, this number can vary from 2 to 4).

The branching factors of all the conditions to reorder are compared and the condition with the lowest branching factor is chosen. The algorithm is very efficient, it orders rules better than humans do and it runs fast even for rules containing more than 200 conditions. More examples of conditions reordering by hand-coded metaprograms are given in [2].

```
preferpath ( X, Y ) :-
        currentstate ( X )        1
        color ( X, + )            1
        color ( Y, + )           81
        connected ( X, Y )        4
        desired (Y ).
```

Figure 3

```
preferpath ( X, Y ) :-
        currentstate ( X )        1
        color ( X, + )            1
        connected ( X, Y )        4
        color ( Y, + )            4
        desired (Y ).
```

Figure 4

Figure 3 and 4 give an example of the difference in the number of instanciations and tests between a bad ordered rule and a well ordered rule. The rule ordered with naive constraints on the number of variables makes 87 instanciations and tests whereas the reordered rule only makes 10 instanciations and tests. For large rules (some of our learned rules for the game of Go contain more than 200 conditions),

the right ordering of conditions by metarules leads to much greater speedups.

2.2 Deletion of useless conditions

The system sometimes learns some rules which contains useless conditions. The figure 5 gives a rule that finds a path of length four to go from one point of a grid to another one without passing twice on the same point. After each new instanciation of a variable in the conditions, the rule verifies that the instanciated point is different from any previously instanciated one.

```
preferpath ( X, Y ) :-
        currentstate ( X ),             1
        connected ( X, Y ),             4
        different ( X, Y ),             4
        connected ( Y, Z ),            16
        different ( Z, X ),            12
        different ( Z, Y ),            12
        connected ( Z, W ),            48
        different ( W, X ),            48
        different ( W, Y ),            36
        different ( W, Z ),            36
        connected ( W, D ),           144
        desired ( D ).
```

Figure 5

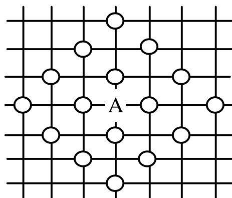

Figure 6

```
preferpath ( X, Y ) :-
        currentstate ( X ),             1
        connected ( X, Y ),             4
        connected ( Y, Z ),            16
        different ( Z, X ),            12
        connected ( Z, W ),            48
        different ( W, Y ),            36
        connected ( W, D ),           144
        desired ( D ).
```

Figure 7

However, in some cases, it is useless to verify that some points are different due to the topology of the grid. For example, two connected points are always different. We can use a metarule that tells to remove the condition 'different (V, V1)' if the condition 'connected (V, V1)' is present in the rule. Such a metarule is given in figure 8. Another metarule given in figure 9 removes the same condition when there is a path of length three between two points, this is a consequence of figure 6 that shows all the points that are at a three step path from point A, they are all different from point A. The initial rule of figure 5 makes 361 instanciations and tests. After firing the metarule of deletion on the initial rule, we obtain the rule of figure 7 which makes only 261 instanciations or tests with the same results.

```
removecondition ( R, different ( V, V1 ) ) :-
        rule ( R ),
        condition ( R, connected ( V, V1 ) ),
        condition ( R, different ( V, V1 ) ).
```

Figure 8

```
removecondition ( R, different ( V, V3 ) ) :-
        rule ( R ),
        condition ( R, connected ( V, V1 ) ),
        condition ( R, connected ( V1, V2 ) ),
        condition ( R, connected ( V2, V3 ) ),
        condition ( R, different ( V, V3 ) ).
```

Figure 9

Figure 8 and 9 give the metarules used to remove the unnecessary conditions of the rule in figure 5.

3 Speedup mechanisms modifying Learning

In this section, we present three mechanisms used to speedup problem solving. The first one is the insertion of cuts in the unification graph. The second one is the specialization of some multi-attributes predicates. The third one is the compilation of the learned rules into C++ programs. These three mechanisms modify the generality and the ability of learning. They are only applied to create the rules of our efficient problem solver. We finish this section by showing how these mechanisms can be used in a learning architecture without altering the generality of learning.

3.1 Cuts in the unification graph

A mechanism is used so as not to deduce many times the same conclusion using different paths in the semi-unification graph. It consists in verifying that the conclusion has not been already deduced when instanciating new variables. This is done by inserting cuts after conditions instanciating variables

with multiples values. A priority is given to the instanciation of the variables present in conclusion in order to instanciate them as soon as possible in the semi-unification of the rule. The sooner they are instanciated in the rules, the more cuts are possible and the more savings are done. In our application to the game of Go, the insertion of cuts approximately doubles the speed of the semi-unification.
The cuts in the unification graph are represented in the rule of figure 10 by '!'. We use a depth first semi-unification strategy. When a variable has multiple instanciations, like Z in the third condition of the rule, we continue to fire the following conditions with the first instanciation of Z. And it is only when all the semi-unification tree following the instanciation of Z has been traversed that we continue with the second instanciation of Z. The cuts after the third condition have worked six times as we can see by comparing the number of instanciations with the rule in figure 7. These savings are done because for these six values of Z, it was unnecessary to develop the tree further as the corresponding conclusion had already been deduced.

```
preferpath ( X, Y ) :-
        currentstate ( X ),          1
        connected ( X, Y ),          4
        connected ( Y, Z ), !        10
        different ( Z, X ),          8
        connected ( Z, W ), !        20
        different ( W, Y ),          16
        connected ( W, D ), !        40
        desired ( D ).
```

Figure 10

This speedup mechanism modifies the generality of learning because it does not deduce the same fact in various ways. Therefore, when explaining the deduction of a fact, the explanation module only produces one explanation. However, it is sometimes useful to produce several explanations of a fact because some explanations of different deductions can be shared and other cannot be shared. If the system has several explanations, it can choose the explanation which makes the learned rules contain the less number of conditions. Moreover, the system can learn several rules from the same example using different explanations for each rule. Cutting the semi-unification graph leads to less explanations and longer learned rules. It prevents from learning multiple rules and makes learned rules contain unnecessary facts and therefore be less general than rules learned without cuts.

3.2 Specialization of some predicates

In order to show the originality of our approach, we will compare it to the description of [36] also using the grid task.

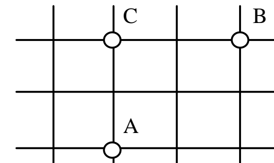

Figure 11

```
preferpath ( X, Y ) :-
        currentstate ( X ),
        upconnected ( X, Y ),
        rightconnected ( Y, Z ),
        upconnected ( Z, W ),
        rightconnected ( W, D ),
        desired ( D ).
```

Figure 12

Tambe [36] compares the influence of the knowledge representation on the generality and the efficiency of learned rules. The number of unique-attribute chunks required for the same level of generality as the multi-attributes chunk for a path of length p is $(p+1)^2$. However, $(p+1)^2$ is the number of points that can be reached with a path of length p. The number of paths to go to this point is greater than one. In some cases, it is necessary to have a chunk for each different path. For example, when you want the system to reach multiple goals with the same move. This is very important in some complex applications like Go, where a move achieving multiple goals is preferred to a move achieving only one goal. Another example, if when there are many paths of length four to go from point A to point B, but that only some paths enables to pass to point C which contains something to pick up, like in figure 11. It is better for a robot to know all the paths so as to be able to choose the one which pass through point C and pursue two goals in one move. Therefore, the number of chunks required for the same level of generality is much higher than $(p+1)^2$ in more complex applications like the game of Go or the achievement of multiple goals in the grid task.
However, the unique attribute representation is faster than the multi-attribute representation, but for other reasons than those given in [36]. The reason

that the unique attribute representation is faster is that it enables the system to evaluate some conditions at compile time. Thus, some of the computations which where done each time the learned rule was fired are now done only once at compile time. If we replace the general rule of figure 7 by its specialized rules, we obtain 144 rules, each of one containing 5 instanciations as shown in figure 12. Therefore we have 5*144=720 instanciations when matching all the rules. To avoid that specialization makes matching slower, we have to share the conditions between rules. If we share the conditions of the partially specialized rules into a tree of conditions, we now have 1+4+12+36+144=197 instanciations for the same result. This is now less than the 261 instanciations and tests of the general rules. What we have done is the removal of the tests and of some instanciations, we still have 197 out of the 261 instanciations of the general rule but we have no more the 36+12=48 tests of the general rule and the 16 useless instanciations which were cut by the tests. Sometimes, the specialization module creates a rule more than once, the unification between rules enables the system to remove redundant rules.

3.3 Compilation in C++

Another source of inefficiency is the interpretation of production rules. When an interpreted problem solver instanciates a variable, it has to go through trees representing the working memory, to create a linked list of the instanciations of the variable and to go through this linked list. Instanciating a variable or making a test requires a lot of instructions at the assembly language level. If a rule is compiled into a C++ program, tests are represented by only one instruction and multiple instanciations by a simple loop.

```
x=current_state;
y=up_connected [x];
z=right_connected [y];
w=up_connected [y];
d=right_connected [w];
if (d==desired) {
        prefer_path (x,y);}
```

Figure 13

```
x=current_state;
for (_y=1; _y<number_of_connections [x]; _y++) {
y=connected [x] [_y];
for (_z=1; _z<number_of_connections [y]; _z++) {
z=connected [y] [_z];
if (z!=x) {
for (_w=1; _w<number_of_connections [z]; _w++) {
w=connected [z] [_w];
if (w!=y) {
for (_d=1; _d<number_of_connections [w]; _d++) {
d=connected [w] [_d];
if (d==desired) {
        prefer_path (x,y);}}}}}}}
```

Figure 14

Our system transforms its learned production rules into C++ programs so as to match them efficiently. Figure 13 gives the program corresponding to the compilation of the rule in figure 12. The system is also able to compile rules containing multi-attribute predicates as shown in figure 14 which represents the compilation of the rule in figure 7. The compilation of interpreted rules into C++ programs gives a factor sixty in the speed of matching the rules.

3.4 How to use these speedup mechanisms in a learning system

Despite the fact that efficient and compiled rules are not used in our interpreted learning system, they are of great use in the overall architecture of the whole learning system. Our learning architecture is composed of two problem solvers. One is interpreted and is used to learn new rules, and the other one is compiled and is used to solve problems quickly and to detect when learning can occur. The

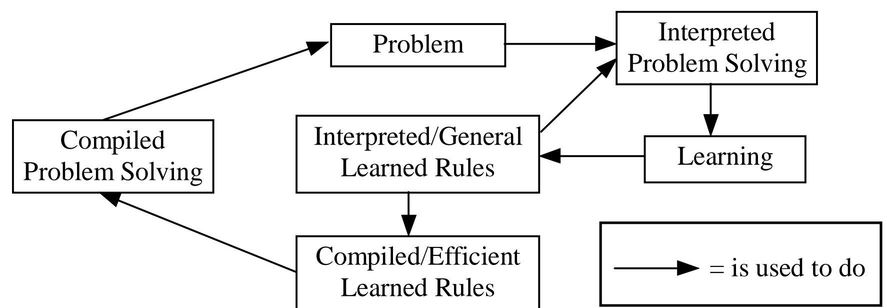

Figure 15

interpreted problem solver uses a general representation so as to learn general rules. Learning

general rules is more efficient than learning a lot of specific rules. For example, to learn the rule of figure 5, a system using a general representation needs only one run. A system using an efficient but specialized representation needs 144 different runs. The use of the general representation requires less time and less examples than the specialized representation for the same results. The compiled version of the problem-solver can be used to detect that learning can occur. When the compiled problem-solver detects that learning can occur, it creates a new problem and gives it to the interpreted problem-solver which learns new rules and integrate them in the two problem solvers.
The architecture of the whole learning system is given in figure 15. Learning and efficient problem solving are two different activities that can be done in parallel.

4 Application to a learning Go system

4.1 Computer 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 deterministic, perfect information, zero-sum game of strategy between two players. The game of Go is the most complex two-person complete information game [1]. Robson [33] proved that Go generalized to NxN boards is exponential in time. Making a good Go program is recognized as a challenge for AI [34]. Today, the best computer Go program is Handtalk. It has the strength of an advanced beginner. This is not due to a lack of work in the computer Go field, the best top programs are the result of more than 10 years of work. But it is rather due to the intractability of search in the domain (250 moves per position, up to 60 moves to look-ahead) and to the huge amount of knowledge necessary to play the game well. The best Go programs are based on knowledge intensive approaches. But there is too much Go knowledge to put in a program to create a good Go program in a reasonable time. That is why large learning techniques are of great interest for the computerization of the game of Go.

4.2 Representation of knowledge in computer Go

A Go board is a grid, therefore the speedup mechanisms used for the grid task are also used for the game of Go. However, in the game of Go, some predicates cannot be specialized. An example of such a predicate is the Liberty predicate. A string of stones can have a number of liberties ranging from 1 to 266. On the contrary of the number and of the location of the intersections connected to a given intersection, the number and the location of liberties is variable. Learned rules in the game of Go mix unique-attribute and multi-attribute predicates.

4.3 Results obtained by our program

Introspect has been used to write the tactical and most important part of a Go playing program named Gogol, it plays a move in 10 seconds on a Pentium 133 MHz. For each move it proves about 450 tactical theorems, each theorem requires between 4 and 600 nodes in a search tree to be proved, at each node of each tree, the rules learned by Introspect are called to find the useful moves to try. Introspect discovered these rules by itself only given the rules of the game. Gogol competed in the international computer Go tournament held during IJCAI97. It finished 6 out of 40 participants. The five first programs are commercial programs that have required a lot of man*years of work. It has outperformed other commercial systems that have required more than 10 man*years of work.

5 Conclusion

We have described a way of managing the tradeoff between generality and efficiency in knowledge representation for large learning systems. We have proposed an architecture that enables to combine efficiency in problem solving to generality in learning. Our architecture combines generality and efficiency by using two problem solvers. The first one is interpreted and uses a general knowledge representation. It enables to learn general rules. The second one is compiled and uses a specialized knowledge representation. It enables to solve problems rapidly and to detect when learning can occur in order to decide to call the first problem solver. We have described speedup mechanisms that allow to transform a general representation into a specialized and efficient one. This approach has shown its success in its application to the game of Go. It is a general approach that has also been applied successfully to other domains [3,4,5].

References

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

[2] T. Cazenave, *Automatic Ordering of Predicates by Metarules*. Proceedings of the 4th International Workshop on Metareasonning and Metaprogramming in Logic, Bonn, 1996.

[3] - T. Cazenave. *Learning to Manage a Firm*. First International Conference on Industrial Engineering Application and Practice, USA, 1996.

[4] - T. Cazenave. *Système d'Apprentissage par Auto-Observation. Application au Jeu de Go*. Thèse de l'Université Paris 6, Décembre 1996.

[5] - T. Cazenave. *Automatically Improving Agents Behaviors in an Urban Simulation*. Second International Conference on Industrial Engineering Application and Practice, USA, 1997.

[6] - M. P. Chase, M. Zweben, R. L. Piazza, J. D. Burger, P. P. Maglio, H. Hirsh. *Approximating learned search control knowledge*. Proceedings of the sixth International Workshop on Machine Learning, pp. 218-220, 1997.

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

[8] - G. Dejong, R. Mooney. *Explanation Based Learning: an alternative view.* Machine Learning 2, 1986.

[9] - R. B. Doorenbos. *Production Matching for Large Learning Systems*. Ph. D. Thesis, Carnegie Mellon University, Pittsburgh, January 1995.

[10] - O. Etzioni. *Why PRODIGY/EBL works.* AAAI-90, pp. 915-922, 1990.

[11] - O. Etzioni. *A Structural Theory of Search Control.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990.

[12] - O. Etzioni. *A structural theory of explanation-based learning.* Artificial Intelligence 60(1) :93-139, 1993.

[13] - C.L. Forgy, *RETE : A Fast Algorithm for the Many Pattern / Many Object Pattern Matching Problem*, Artificial Intelligence vol. 19, pp 17-37, 1982.

[14] - J. Gratch, G. Dejong. *COMPOSER : A probabilistic solution to the utility problem in speed-up learning*, AAAI-92, pp 235-240, 1992.

[15] - R. Greiner, I. Jurisica. *A statistical approach to solving the EBL utility problem*, AAAI-92, pp 241-248, 1992.

[16] - L. B. Holder. *Empirical Analysis of the general utility problem in machine learning*, AAAI-92, pp 249-254, 1992.

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

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

[19] - P. Laird. *Dynamic Optimization*. ICML-92, pp. 263-272, 1992.

[20] - S. Markovitch, P. D. Scott, *Information Filtering: Selection Mechanisms in Learning Systems*, Machine Learning 10, pp. 113-151, 1993.

[21] - Minton S. *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.

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

[23] - S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, Y. Gil. *Explanation-Based Learning : A Problem Solving Perspective*. Artificial Intelligence 40, 1989.

[24] - S. Minton. *Quantitative Results Concerning the Utility of Explanation-Based Learning*. Artificial Intelligence 42, 1990.

[25] - S. Minton. *Is There Any Need for Domain-Dependent Control Information : A Reply*. AAAI-96, 1990.

[26] - S. Minton. *Automatically Configuring Constraints Satisfaction Programs : A Case Study*. Constraints, Volume 1, Number 1, 1996.

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

[28] - T. M. Mitchell et al. *Theo : A Framework for Self-Improving Systems*. In Architecture for Intelligence, K. VanLehn, Ed., Erlbaum,1991.

[29] - Y. Parchemal. *SEPIAR : un système à base de connaissances qui apprend à utiliser efficacement une expertise.* Thèse de l'Université Paris 6, 1988.

[30] - M. A. Pérez, O. Etzioni. *DYNAMIC : A new role for training problems in EBL.* ICML-92 pp 367-372, 1992.

[31] - J. Pitrat. *Realization of a Program Learning to Find Combinations at Chess*. Computer Oriented Learning Processes, Simon J. Ed., Noordhoff, 1976.

[32] - J. Pitrat, *Métaconnaissance - Futur de l'Intelligence Artificielle*, Hermès, Paris, 1990.

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

[34] - B. Selman, R. A. Brooks, T. Dean, E. Horvitz, T. M. Mitchell, N. J. Nilsson. *Challenge Problems for Artificial Intelligence*, AAAI-96, pp. 1340-1345, 1996.

[35] - D. Subramanian, R. Feldman. *The utility of EBL in recursive domains*, AAAI-90, pp. 942-949, 1990.

[36] - M. Tambe, A. Newell, P. S. Rosenbloom, *The problem of expensive chunks and its solution by restricting expressiveness*. Machine Learning 5 (3) (1990), pp. 299-348, 1990.

[37] - M. Tambe, P. S. Rosenbloom, *Investigating production system representations for non-combinatorial match*. Artificial Intelligence 68 (1994), pp. 155-199, 1994.