Generating Search Knowledge in a Class of Games

Tristan Cazenave[1]

Abstract. We present the Introspect system that generates search knowledge for different games. It writes selective and efficient search programs given a logic description of a game. It can be applied to the class of games where the winning state is simply defined. We explain how the system works using the game of capturing strings in the game of Go. We begin with a description of the different kinds of metaknowledge used in Introspect. Then we analyze the consequences of different choices that can be made when designing the domain theory. Finally, we give experimental results using different limits on the amount of search and the amount of generated knowledge.

[1] Laboratoire d'Intelligence Artificielle, Département Informatique, Université Paris 8, 2 rue de la Liberté, 93526 Saint Denis, France.cazenave@ai.univ-paris8.fr

Generating Search Knowledge in a Class of Games

Abstract. We present the Introspect system that generates search knowledge for different games. It writes selective and efficient search programs given a logic description of a game. It can be applied to the class of games where the winning state is simply defined. We explain how the system works using the game of capturing strings in the game of Go. We begin with a description of the different kinds of metaknowledge used in Introspect. Then we analyze the consequences of different choices that can be made when designing the domain theory. Finally, we give experimental results using different limits on the amount of search and the amount of generated knowledge.

1 Introduction

It is very important in complex games where search trees have a large branching factor to acquire knowledge about all possible moves worth trying. Knowing the moves worth trying and the moves that can be eliminated, drastically reduces the search trees. Metaprogramming can be used to automatically create knowledge about worthwhile moves and forced moves given the rules about the direct effects of these moves [Cazenave 98].

Introspect is a metaprogramming system that writes search programs that finds all the relevant moves in search trees. It can be applied to the class of games where the winning state is simply defined. Many sub-games of the game of Go fall into this class : capturing a string, connecting two strings, making an eye and so on. Some other classical games such as Go-Moku, Renju, Chess, Abalone or Phutball are also concerned.

The Capture game is a fundamental sub-game of the game of Go. All the non-trivial computer programs dedicated to Go as well as many other sub-games of Go use it. Go is known to be very difficult to program [Herik & al. 91]. Solving problems in a complex game can be applied to other games and other planning domains especially when the approach uses general methods as is the case of our system. Introspect has been applied to other games but we will only deal with the Capture game in this paper.

The second section describes metaprogramming and especially metaprogramming in games. The third section uncovers how to represent a domain theory in order to generate efficient search programs. The fourth section gives experimental results.

2 Metaprogramming in Games

Introspect writes programs that safely cut search trees, and consequently speed up search programs. In our applications to games, metaprograms are used to create theorems that find the interesting moves to achieve a tactical goal (at OR nodes), and theorems on winning moves that stop search early. They are also used to create rules that find the complete set of forced moves that prevent the opponent to achieve a tactical goal (at AND nodes).

Metaprogramming in logic has already attracted some interest [Barklund 94]. More specifically, specialization of logic program by fold/unfold transformations can be traced back to [Tamaki & Sato 84], it has been well defined and related to Partial Evaluation in [Lloyd & Shepherdson 91], and successfully applied to different domains. The parallel between Partial Evaluation and Explanation-Based Learning [Pitrat 76, Mitchell & al. 86, Laird & al. 86, Dejong & Mooney 86, Minton & al. 89,Ram & Leake 95] is now well known [Harmelen & Bundy 88, Etzioni 93]. The ability for programs to reason on the rules of a game so as to extract useful playing knowledge is an essential problem for future AI [Pitrat 98]. It is a step toward the realization of efficient general problem solvers.

In our system, two kinds of metarules are particularly important : impossibility metarules and monovaluation metarules. Other metarules such as metarules which remove useless conditions or order metarules, are used to speed-up the generated programs. In this section, we will begin with a description of the game of capturing strings. Then we will compare our methods to usual program transformation methods. We will continue with the in-depth description of the different kinds of knowledge used in our metaprogramming system : impossibility metaknowledge, monovaluation metaknowledge, unfolding, simplification of generated rules, ordering of conditions, utility metaknowledge and compilation.

2.1 Generating search knowledge for the capturing game

We will describe the generation of search programs for the sub-game of capturing strings in the game of Go. Therefore, we give a brief description of some basic concepts of Go.

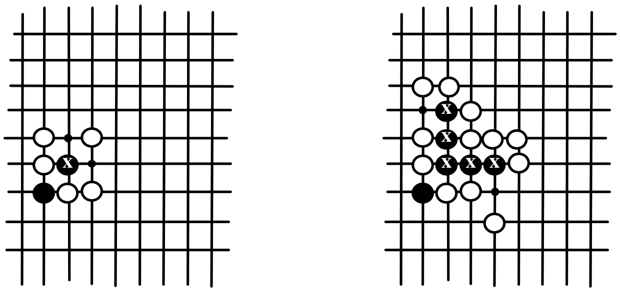


Figure 1. Examples of captured strings

A stone is directly connected to another stone of the same color if it is on the up, down, right or left intersection of the other stone. A string is a maximally connected set of stones of the same color. In the first diagram of figure 1, the string marked with a x is composed of only one stone. In the second diagram, the five stones belonging to the string are marked with x. A string is associated to a number of liberties: this is the number of empty intersections directly connected to the string. In the two diagrams, the strings marked with x have two liberties marked with small black dots. A string is in atari if it has only one liberty left. A string is adjacent to another string of the opposite color if it has stones next to the other's. The marked black string has two adjacent strings in the first diagram, and four adjacent strings in the second diagram. A string is captured if it is in atari and the opponent plays on its liberty.

Patterns are a very convenient way to represent knowledge in some search domains and in particular in the game of Go. Pattern databases have been automatically generated to solve efficiently single-agent search problems [Korf 97] as well as sub-games of the game of Go [Cazenave 00]. However patterns have shortcomings in some games. This is the case for the game of capturing strings where knowledge is better acquired through a program than through a pattern. For example, in the first diagram of figure 1, knowledge can be contained in a small pattern with external information, but in the second one it cannot easily fit in a small pattern. If a rectangular pattern is used to find the status of the marked black string in the second diagram, it will be a very specific and almost useless pattern. However the forced moves in the two diagrams can be found by the same generated logical rule. The two approaches are complementary, and on some localized problems patterns enable to save a lot of search. In this paper we focus on the generation of programs.

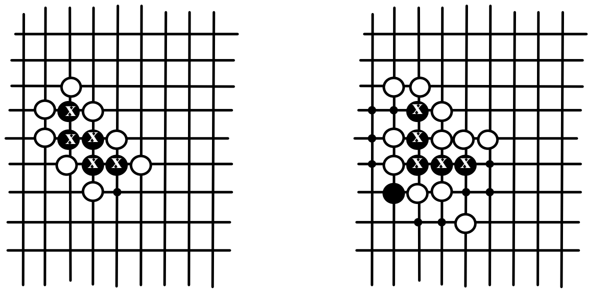


Figure 2. Examples of sets of forced moves

Figure 2 gives the results of matching two rules in order to find sets of forced moves. The small black dots give the complete sets of moves that can prevent white from capturing the marked black stones. All the other moves do not work. The goal of our system is to generate the rules that find these sets. An important feature of the generated rules is that they do not forget moves worth trying. If none of the forced moves works, then no other move works and the string is proved to be captured.

2.2 Program transformation

Program transformation is an active research domain [Pettorossi & Proietti 96], but people usually study general program transformation that can be applied to any program. We rather focus on domain specific program transformations. However, we use some traditional program transformation mechanisms. The most classical one is unfolding. It consists in replacing a condition by its definitions. For example if we unfold the condition `'liberty_after(NumberMove,Move,S)'` in the following rule:

```
winning_move_after(NumberMove,1,C1,capture,S,I):-
          color_string_after(NumberMove,S,C),
          opposite_color(C1,C),
          number_liberties_after(NumberMove,S,1),
          liberty_after(NumberMove,I,S),
          number_rule(winning1, 1).
```

and if we have the two following definitions of `liberty_after`:

```
liberty_after(NumberMove,I,S):-
          liberty_before(I,S),
          move(NumberMove,Move),
          side_to_move(C),
          opposite_color (C,C1),
          color_string_before (S,C1),
          Move=\=I,
liberty_after(NumberMove,I1,S):-
          side_to_move(C),
          move(NumberMove,Move),
```

```
           color_string_before(S,C),
           Move=\=I1,
           liberty_if_move_before(I1,S,[Move],[C]).
```

we get these two unfolded rules:

```
winning_move_after(NumberMove,1,C1,prendre,S,I):-
           color_string_after(NumberMove,S,C),
           opposite_color(C1,C),
           number_liberties_after(NumberMove,S,1),
           number_rule(winning1, 1),
           liberty_before(I,S),
           move(NumberMove,Move),
           side_to_move(C2),
           opposite_color (C2,C3),
           color_string_before (S,C3),
           Move=\=I,
winning_move_after(NumberMove,1,C1,prendre,S,I):-
           color_string_after(NumberMove,S,C),
           opposite_color(C1,C),
           number_liberties_after(NumberMove,S,1),
           number_rule(winning1, 1),
           side_to_move(C2),
           move(NumberMove,Move),
           color_string_before(S,C2),
           Move=\=I,
           liberty_if_move_before(I,S,[Move],[C2]).
```

The other important transformation we use, is the metaprogram that finds the rules about forced moves given the rules on winning moves. This is not a classical transformation rule; it is specific to games.

For example, the only move able to modify a condition 'liberty_before(Lib,S)' is to play on the liberty. In this case the system adds the variable Lib to the set of forced moves that modifies the conditions of the winning rule at hand.

Sometimes the system adds conditions to the rule at hand so as to be sure that he has the complete set of forced moves. Particularly when it creates the set of forced moves able to change the number of liberties of the string to be captured, two kinds of moves may change this number. Either the owner of the string plays on one of the liberties, or it removes an adjacent string. So the set of forced moves depends on the configuration of the adjacent strings. For example, if the string to be captured has two liberties, the only moves to consider are the liberties of the string if there are no adjacent string with one or two liberties. But if there is only one adjacent string with one liberty and no adjacent string with two liberties, then the set of forced moves is not the same: it contains the move that takes the adjacent string with one liberty. Therefore, the same condition generates multiple rules about forced moves. The designer of the metarules which will find forced moves has to be cautious about this, because if the condition leading to many rules often appears in the same rule, the number of generated forced rules quickly explodes.

The rules about forced moves that are generated in order to be used in the program are not exactly the same as the rules that are generated in order to generate other rules. For example, when looking for the complete set of forced moves that prevent to capture a string with two liberties, the system generates only one rule to be compiled which contains a predicate collecting all the moves that can take the adjacent strings with strictly less than three liberties. If the goal is to generate rules that will be used to generate other rules, the system considers different cases that will lead to small generated rules:

- no adjacent string in atari, and no adjacent string with two liberties.
- one adjacent string in atari, and no adjacent string with two liberties.
- no adjacent string in atari, and one adjacent string with two liberties.

2.3 Impossibility metaknowledge

Impossibility metarules find which rules can never be applied because of some properties of the game, or because of more general impossibilities. An example of a metarule about a general impossibility is the following one :

```
impossible(ListAtoms):-
          member(N=\=N1,ListAtoms),var(N),var(N1),N==N1.
```

This metarule detects that if a rule created by the system contains the condition `'N=\=N1'` and the metavariables N and N1 contain the same variable, then the condition can never be fulfilled. So the rule can never apply because it contains a statement impossible to verify. This metarule is particularly simple, but this is the kind of general knowledge a metasystem must have to be able to reason about rules and to create rules given the definition of a game.

Some of the impossibility metarules are more domain specific. For example the following rule is specific to the game of Go :

```
impossible(ListAtoms):-
    member(number_liberties_before(S,N),ListAtoms),
    member(number_liberties_before(S1,N1),ListAtoms),
    S==S1,nonvar(N),nonvar(N1),N=\=N1,!.
```

It states that a string cannot have two different numbers of liberties.

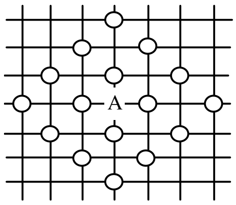


Figure 3. All the intersections at a path three of intersection A are marked.

The rule that finds all the intersections is:

```
path(X,W,3):-
          connected(X,Y),connected(Y,Z),connected(Z,W).
```

This rule is generated unfolding the goal `'path(X,W,3)'` defined below:

```
path(X,X,0).
path(X,W,N1):-N is N1-1,path(X,Z,N),connected(Z,W).
```

On a grid, all the intersections that are at a path two of X are not at a path three (figure 3 gives all the intersections at a path three of intersection A). Moreover, when unfolding definitions in more complex domains, it often happens that two variables are unified, there is only one variable left after unfolding. Unfolding can lead to generate rules similar to this one:

```
path(X,X,3):-
          connected(X,Y),connected(Y,Z),connected(Z,X).
```

This rule has been correctly generated by a correct metaprogram on a correct domain theory, but it is a rule that will never be applied, because no intersection on a grid is at a path three of itself.

Impossibility metarules match generated rules to find subsets of impossible conditions. If an impossibility metarule succeeds, the generated rule is removed. The impossibility metarule used to detect the impossibility in our example is:

```
impossible(ListAtoms):-
          member(connected(X,Y),ListAtoms),
          var(X),var(Y),X\==Y,
          member(connected(A,Z),ListAtoms),
          A==Y,var(A),var(Z),A\==Z,
          member(connected(B,C),ListAtoms),
          B==Z,var(B),var(C),B\==C,C==X.
```

Creating new impossibility metarules

When generated rules are compiled, some of them never apply. When compiling the rule, we include a counter that records the number of times they were used.

Therefore, the system can detect generated rules that never apply and send a message about this anomaly. It often points to a missing impossibility metarule, which we then add.

Introspect automatically creates impossibility metarules by using complete sets of facts. For example, the topology of the Go board never changes and it can be completely represented by a set of facts. Therefore the system can automatically generate impossibility metarules, as it knows the set of facts is complete. If a particular set of conditions, only containing board topology conditions, cannot be matched then it is always impossible to match this set of conditions. The system will then create an impossibility metarule with this information.

2.4 Monovaluation metaknowledge

The other important metarules in Introspect are the monovaluation metarules. They apply when two variables in the same rules always share the same value. Monovaluation metarules unify such variables. They help simplifying the rules and detecting more impossible rules. An example of a monovaluation metarule is:

```
monovaluation(ListAtoms):-
          member(color_string_before(S,C),ListAtoms),
          member(color_string_before(S1,C1),ListAtoms),
          S==S1,C\==C1,C=C1.
```

It states that a string can only have one color. So if the system finds two conditions `'color_string_before'` in a rule where the variables contained in the metavariables `'S'` and `'S1'` are equal, and if the corresponding color variables contained in the metavariables `'C'` and `'C1'` are not the same, it unifies `C` and `C1` because they always contain the same value.

Most of the monovaluation metarules simply unify variables. But some monovaluation metarules are more complex like the metarules that duplicate rules. For example, when a string has only two liberties and has three different variables in three different conditions 'liberty_before', there are three possible unifications between two of the three variables. This leads to the replacement of the rule by three different rules, and each one only contains two conditions on 'liberty_before'.

Impossible and monovaluation metarules are vital rules of our metaprogramming system. They significantly reduce the number of rules created by the system, eliminating many useless rules. For example, we tested the unfolding of six rules with and without these metarules. Without the metarules, the system created 166,391,568 rules by regressing the six rules on only one move. Using basic monovaluation and impossible metarules shows that only 106 rules are valid and different from each other.

2.5 The unfolding process

The main transformation used in Introspect is unfolding. It is used to generate winning move rules, winning in two moves of the same color rules, and won rules. Each of these unfolding begins with some rules that define the goal. For example, a string is captured if all the forced moves for its safety lead to winning moves for the opponent. In the case of forced rules with a set of three forced moves, a won rule is generated when there is a winning move after each of these three forced moves, so the following rule is used for unfolding won rules:

```
won_before(N,C1,capture,S):-
    side_to_move(C),
    opposite_color(C1,C),
    color_string_before(S,C),
    move(0,I),
    color_intersection_before(I,empty),
    move(1,I1),
    color_intersection_before(I1,empty),
    move(2,I2),
    color_intersection_before(I2,empty),
    forced_move_before(N1,C,capture,S,[I,I1,I2]),
    winning_move_after(0,N2,C1,capture,S,_I4),
    winning_move_after(1,N3,C1,capture,S,_I5),
    winning_move_after(2,N4,C1,capture,S,_I6),
    N is max(N4,max(N3,max(N1,N2))).
```

Unfolded conditions are conditions ending with `'_after'`. The other conditions are never unfolded. Similarly, to find winning moves, the following rule is unfolded:

```
 winning_move_before(N,C,prendre,S,I):-
    opposite_color(C1,C),
    side_to_move(C),
    move(0,I),
    color_intersection_before(I,vide),
    color_string_before(S,C1),
    won_after(0,N1,C,capture,S),
    N is N1+1.
```

Before generating any rule, the system begins with one rule about the winning move. For the game of capturing stones in the game of Go, it is the rule of the unfolding example with the head `'winning_move_after(NumberMove, 1, C1, capture, S, I)'`. This rule is used to generate rules about complete sets of forced moves to prevent the string to be captured in one move. Then, the rules about forced moves and the rule about direct capture are used to generate rules on won games, i.e. rules that find when the string is captured in two moves even if the color of the string plays first. These rules on won games are used to generate rules on winning in three moves. Then, generation of rules continues to the next levels, until some utility metaknowledge stops it. Rules that give winning moves after a move has been played, are obtained by replacing the string '`before(`' with the string '`after(MoveNumber,`' in the winning rules.

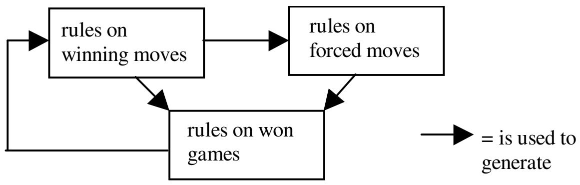


Figure 4. The cycle of rules generation.

Some optimizations are used to speed-up unfolding. For example, if the system does not care for it, it will unfold many times the same rules using the rule about won games. Each of the three unfolded winning_move_after conditions are replaced by the same set of rules, therefore the same rule will be generated with different orders of replacement of the conditions winning_move_after. In short, we do not want all the permutations but only the combinations. Therefore the system stores the different sets of three rules that have been used to unfold the winning_move_after conditions, and stops unfolding if the same combination has already been stored.

Another optimization is to choose the condition to unfold. Before unfolding a condition, the system tests all the possible unfoldings and counts the number of rules generated by each unfolding. Then it unfolds the condition that leads to the fewer number of rules. Therefore many unfoldings are shared between generated rules and the overhead of choosing the condition to unfold is less than the time gained by sharing unfolding.

2.6 Simplifying metaknowledge

The system sometimes generates some rules that contain useless conditions. Below, we give a rule that finds a path of length four to go from one intersection of a grid to another one without going twice through the same intersection. After each new binding of a variable in the conditions, the rule verifies that the bound intersection is different from any previously bound one.

After each condition in the rule, we give the number of times the condition has been verified when matching the rule once on a set of facts.

```
path(X,D,4):-
         connected(X,Y),                       4
         X=\=Y,                                4
         connected(Y,Z),                      16
         Z=\=X,                               12
         Z=\=Y,                               12
         connected(Z,W),                      48
         W=\=X,                               48
         W=\=Y,                               36
```

```
            W=\=Z,                              36
            connected(W,D).                    144
```

However, in some cases, it is useless to verify that some intersections are different due to the topology of the grid. For example, two neighboring intersections are always different. We can use a metarule that removes the condition `'X=\=Y'` if the condition `'connected(X,Y)'` is also present in the rule:

```
useless(X=\=Y,ListAtoms):-
            member(connected(X,Y),ListAtoms),
            member(A=\=B,ListAtoms),A==X,B==Y.
```

Another metarule, given below, removes the condition `'X=\=Y'` when there is a path of length three between the two intersections contained in `X` and `Y`, this is a consequence of figure 3 that shows all the intersections that are at a three step path from intersection A: A is not at a three step path of itself.

```
useless(X=\=Z,ListAtoms):-
            member(connected(X,Y),ListAtoms),
            var(X),var(Y),X\==Y,
            member(connected(Y,Z),ListAtoms),
            var(Y),var(Z),Y\==Z,
            member(connected(Z,A),ListAtoms),
            var(Z),var(A),Z\==Y,A==X.
```

The initial rule produces 360 instanciations and tests. After applying the metarule of deletion on the initial rule, we obtain a rule giving the same results after only 260 instanciations and tests as shown below.

```
distance(X,D,4):-
            connected(X,Y),                      4
            connected(Y,Z),                     16
            Z=\=X,                              12
            connected(Z,W),                     48
            W=\=Y,                              36
            connected(W,D).                    144
```

General simplifications

Other kinds of simplification are also useful. For example, it happens that a generated rule contains two equivalent sets of conditions. These two sets use the same variables except for one. If the two sets are the complete sets of conditions containing the variables that differ, and if the variables that differ in the two sets are not used in the conclusion of the rule, then one of the set of conditions can be discarded. This simplification avoids useless work when matching the generated rule, and it is very useful when the generated rule is used to generate other rules, because the inherited rules would otherwise always contain the different possible specializations for the two different sets instead of only one. Therefore this simplification not only reduces the match cost of the rule at hand, but also saves generation time and lessens the matching cost of the inherited rules. Here is an example from the capturing game where the last three conditions of a rule to capture three moves ahead are removed :

```
winning_move_before(2, Color, capture, S, Move):-
  color_string_before(S, OpColor),
  opposite_color(Color, OpColor),
  number_of_liberties_before(S, 2),
  liberty_before(Move, S),
  liberty_before(Lib1, S),
  Move=\=Lib1,
  one_adjacent_string_atari_before(S, Lib1),
  nb_liberties_if_move_before(S, OpColor, [Move, Lib1],
           [Color, OpColor], 1),
  liberty_if_move_before(Lib2, S, [Move, Lib1],
          [Color, OpColor]),
  Lib2=\=Move,
  color_intersection_before(Lib2, empty),
  liberty_if_move_before (Lib3, S, [Move, Lib1],
          [Color, OpColor]),
  Lib3=\=Move,
  color_intersection_before(Lib3, empty).
```

2.7 Ordering metaknowledge

P. Laird [Laird 92] uses statistics on some runs of a program to reorder and to unfold clauses. T. Ishida [Ishida 88] also dynamically uses simple heuristics to find a good ordering of conditions for a production system. Our approach is based on a metaprogram that statically reorders the conditions of the clauses. Once the metaprogram is created, running it to reorder generated rules is faster than dynamically optimizing the learned rules. This feature is important for systems that use a large number of generated rules. The generation of the metaprogram itself is also fast.

We rely on the assumption that domain-dependent information can enhance problem solving [Minton 96a]. This assumption is given experimental evidence on constraint satisfaction problems by S. Minton [Minton 96b]. We do not specialize heuristics on specific problem instances as Minton does, but we rather create metaprograms according to statistics on many working memories.

Reordering conditions

Reordering conditions is very important for the performance of generated rules. The two following rules are simple examples that show the importance of a good order of conditions. The two rules give the same results but do not have the same efficiency when `X` is known and `Y` unknown:

```
sisterinlaw(X,Y):-brother(X,X1),married(X1,Y),woman(Y).
sisterinlaw(X,Y):-woman(Y),brother(X,X1),married(X1,Y).
```

Reordering based only on the number of free variables in a condition does not work for the example above. In the literature on constraint programming, constraints are reordered according to two heuristics concerning the variables to bind [Minton 96b] : the range of their values and the number of other variables they are linked to. These heuristics dynamically choose the order of constraints. But to do so, they have to keep the number of possible bindings for each variable, and hence lose time when dynamically choosing the variable. It is justified in the domain of constraint solving because the range of values of a variable highly affects 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 order statically conditions, and thus variables. We reorder once for all and not dynamically at each match because it saves more time in the domains in which we have tested our approach.

Reordering optimally the conditions in a given rule is a NP-complete problem. To reorder conditions in our generated rules, we use a simple and efficient algorithm. It estimates the number of different bindings generated by the matching of the free variables of a condition. Here are two metarules used to reorder conditions of generated rules in the game of Go:

```
branching(ListAtoms,ListBindVariables,
          connected(X,Y),3.76):-
          member(connected(X,Y),ListAtoms),
          member_term(X,ListBindVariables),
          non_member_term(Y,ListBindVariables).
branching(ListAtoms,ListBindVariables,
          elementstring(X,Y),94.8):-
          member(elementstring(X,Y),ListAtoms),
          non_member_term(X,ListBindVariables),
          non_member_term(Y,ListBindVariables).
```

A metarule evaluates the branching factor of a condition based on the estimated average number of facts matching the condition in the working memory. Metarules are applied 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 applied N times: the first time in order to choose the first condition of the rule, and at time T to choose the T^{th} condition. In the first reordering metarule above, the X variable is already present in some of the conditions preceding the condition to be chosen. The Y variable is not present in the preceding conditions. The condition `'connected(X,Y)'` is therefore estimated to have a branching factor of 3.76 which is the average number of bindings of Y (this is the average number of neighboring intersections on a 19*19 grid).

The branching factors of all the conditions are compared and the condition with the lowest branching factor is chosen. The algorithm is efficient and fast even for rules containing more than 100 conditions.

2.8 Utility metaknowledge

If we choose to generate all possible knowledge, the system is rapidly stopped by the utility problem: either it creates too many useless rules that fill the memory and take a lot of time to be produced, or it creates rules that very rarely match but that are time consuming.

We do not want to keep complicated rules that very rarely apply and that can take a very large amount of memory. Other authors [Minton 90, Markovitch & Scott 93] have dealt with this problem by computing statistics on the use of learned rules. But to get statistics about rules, the rules need to be generated and applied. If we do not restrain the number of rules when they are generated, it will very quickly explode.

Therefore, we need to control the unfolding with some metaknowledge that can estimates the interest of a generated rule without using it. We have used some simple metarules to discard useless rules. The first simple heuristic consists in calculating the number of conditions of a rule, and to set a threshold for this number. This is a simple and efficient way to keep only the simplest rules, and the rules that have usually the best chance of being applied as they have the lowest number of conditions to be fulfilled.

Another estimation of the utility of a rule is its branching factor given by the reordering knowledge. Once a rule is ordered, we know the average number of branching of each condition and the cost of the test of the fully bound conditions. Therefore we can compute an estimate of the overall cost of the rule. A utility metarule detects rules that have a computed cost higher than a given threshold and discards them immediately.

The other utility metarule is domain dependent and optional. It discards all the rules on the capture of a string that has three liberties.

2.9 Compilation metaknowledge

Another source of inefficiency is the interpretation of generated rules. When an interpreted problem solver binds a variable, it has to find the matched facts in the working memory, create a linked list of the variable bindings and go through this linked list. Binding a variable or making a test requires a lot of instructions at assembly language level. If a rule is compiled into a C program, tests are represented only by one instruction and multiple bindings by a simple loop.

Before compilation, the condition lists of the rules are collected into a single tree of conditions. The conditions at the beginning of the rules, which are often the same, can thus be shared by all other rules. Condition trees speeds up the generated programs by a factor two. This improvement is much higher if the cache of conditions is not used in the list representation of the C programs. Moreover the size of the generated program is smaller with the tree representation, and for some games the size of the generated program is the critical resource. For example, when compiling the rules on forced moves about string of two liberties five moves ahead, the list representation gives a 4203 line C program taking 70.22 seconds to solve a problem set using caching, whereas the tree representation gives a 3221 line C program taking 33.22 seconds to solve the same set of problems.

3 Representing the Domain Theory

The design of the domain theory has a very strong influence on the results of metaprogramming. Two theories that are strictly semantically equivalent, i.e. that find the same moves when knowledge is generated, can have very different efficiency depending on how the knowledge is represented. In this section we will compare different predicates for different concepts of the game of Go. These predicates are sometimes specific to Go, however they give hints and ideas of implementations used in other games. Moreover, the domain theory of Go is the one that has the most subtle implementation over all the domain theories implemented in Introspect for different games. Therefore, analyzing it in depth is a good way to introduce the tricks and traps of designing domain theories in general.
For example, the neighbors of an intersection can be represented either with a predicate neighbor or with four different predicates: left, right, up and down [Tambe & al. 90, Tambe & al. 94]. With the first representation, finding a liberty of an intersection costs one rule, whereas it costs four rules with the second representation. For example, a generated rule that contains five conditions on neighboring intersections is equivalent to 4^5=1024 rules with the predicates up, down, left and right. If we choose the four predicate representation, the generated program will be a little faster because some impossible cases are detected by metarules at generation time instead of being dynamically detected by useless match each time the generated program is run. However, the gain in speed is not very important compared to the problems due to the size of the generated program. It often happens that the program does not fit in memory. Moreover, what takes days to generate with the four predicate representation, only takes a few seconds with the neighbor representation.
Another trick in representing the rules of Go for metaprogramming is to use the condition `'min_number_of_liberties(S,2)'` instead of the two conditions `'number_of_liberties(S,N), N>=2'`. The unfolding of the predicate number_of_liberties uses rules exactly counting the number of liberties of the string before the move. Therefore, it leads to more generated rules than the predicate min_number_of_liberties which only verifies the number is over a given threshold. However in some cases, the generated search program has to know the exact number of liberties of a string, so it must unfold the predicate number_of_liberties. There is only one way to reduce the number of liberties of a string when trying to capture it: playing a legal move on one of its liberties. There are many ways to increase the number of liberties of a string when trying to save it:

- playing one of its liberties, with one, two or three neighboring empty intersections that are not yet liberties of the string.
- capturing an adjacent string in atari, and counting the number of adjacent stones of the captured string.

- connecting the string to one, two or three other strings of the same color.
- and a lot of combinations of the three first ways (for example, connecting to another string which has two liberties, one liberty in common with the string at hand, and only one empty intersection neighboring the move that is not yet liberty of the string and not yet a liberty of the connected string: this adds two liberties).

If we describe all these cases in the domain theory, the number of rules of the domain theory is quite large, a condition on the number of liberties leads to a large number of generated rules, and the system is stopped after two or three unfolding of a move because of the time used to generate the rules, of the number of rules and of the size of the generated program that cannot fit in memory. Moreover, the generated program is quite slow because for each different case, it creates a branch in the condition tree of the generated programs, but the conditions below each branch of the tree are often the same (and need the bindings of the variables above to run, so they cannot be shared), therefore the generated program computes many times the same portions of code in the different branches of the program tree.
A solution is to use the predicate number_of_liberties_if_moves in the domain theory. It hides the complexity of calculating the different cases of liberty counting in a predicate that can be directly computed when the generated program is run. Instead of having a lot of specialized rules, we have four simple rules. The first one is used when a move of the opposite color reduces its number of liberties by playing on a liberty:

```
number_liberties_after(NumberMove,S,N2):-
  side_to_move(C),
  opposite_color(C,C1),
  move(NumberMove,Move),
  liberty_before(Move,S),
  color_string_before(S,C1),
  N1 is N2+1,
  number_liberties_before(S,N1).
```

the other case of a move of the opposite color is a move that is not on a liberty:

```
number_liberties_after(NumberMove,S,N):-
  side_to_move(C),
  opposite_color(C,C1),
  move(NumberMove,Move),
  non_liberty_before(Move,S),
  color_string_before(S,C1),
  number_liberties_before(S,N).
```

We have now all the unfolding cases for the color that captures strings. For the opposite color, we only need two rules for all the cases (LI is a list of intersections and LC a list of colors):

```
number_liberties_after(NumberMove,S,N):-
  side_to_move(C),
  move(NumberMove,Move),
  color_string_before(S,C),
  nb_liberties_if_move_before(S,C,[Move],[C],N).
nb_liberties_if_move_after(NumMove,S,C1,LI,LC,N):-
  side_to_move(C),
  move(NumMove,Move),
  nb_liberties_if_move_before(S,C1,[Move|LI],[C|LC],N).
```

However, it is not always good to hide all the information. For example, it is better to keep the two rules for the player that tries to capture, and not to use the predicate number_of_liberties_if_moves in this case, because these rules are only used to generate rules on winning moves (due to the color of the move) and either the winning move is defined by the predicate liberty_before in the first rule, or it is defined elsewhere and the second rule is used. Using the predicate number_of_liberties_if_moves would then lose essential information on the move and would generate very slow programs (the only information on the move would be that the intersection is empty, inducing a branching factor of approximately 250 for the condition that finds the move).

The exclusivity between rules of the domain theory is another important property. A rule is exclusive from another one if the sets of positions verified by the two rules are disjoint. For example a rule that contains the condition 'liberty_before(Move,S)' is exclusive of a rule that contains the condition 'non_liberty_before(Move,S)'. There are many ways for an intersection to become a liberty of a string. So we will use the same representation as for the number of liberties. The rules about the liberties after the move are:

```
liberty_after(NumberMove,I,S):-
  side_to_move(C),
  opposite_color(C,C1),
  move(NumberMove,Move),
  liberty_before(I,S),
  color_string_before(S,C1),
  Move=\=I.
liberty_after(NumeroCoup,I,B):-
  side_to_move(C),
  opposite_color(C,C1),
  move(NumberMove,Move),
  color_string_before(S,C),
  Move=\=I,
  liberty_if_move_before(I,S,[Move],[C]).
liberty_if_move_after(NumeroCoup,I,S,LI,LC):-
  side_to_move(C),
  move(NumberMove,Move),
  liberty_if_move_before(I,S,[Move|LI],[C|LC]).
```

The conditions on the color of the string are unnecessary in the first rule. However they make the first rule exclusive of the second one. Using these three unnecessary conditions divides the number of generated rules by ten when generating rules for string captured four moves ahead. Furthermore, as the number of generated rules tends to be exponential with the number of moves ahead, these 'unnecessary' conditions are quite useful. However, exclusivity is not a sufficient property. The real parameter that needs attention is the maximum number of generated rules created by unfolding a condition. A lot of exclusive rules can be written about the predicate liberty_after, but the best representation is the three rules above because they do not increase the number of generated rules.

4 Experimental Results

This section gives the results and the analysis of some experiments in generating programs for the sub-game of capturing strings in Go. We begin with some examples of the matching of a generated rule. Then we explain how generated rules are used to develop search trees. And we give experimental results on a standard test set.

4.1 Examples of generated rules

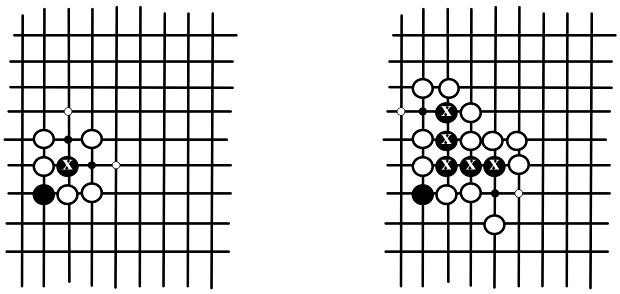


Figure 5. Examples of forced moves found by generated rules

In the two examples of figure 5, the system finds two forced moves. The forced moves are found by applying twice the same rule in the two examples. Each time, the rule about forced moves finds a set composed of the two intersections marked with a small black dot, and of an intersection marked with a small white dot. In each case the system finds two complete sets of forced moves, makes the intersections of the sets of forced moves, and hence finds that the set of forced moves is the two intersections marked with small black dots. The rule that finds the sets of forced moves is the following rule generated for finding forced moves three moves ahead:

```
forced_move_before(2, C, capture, S, [I1, I2, I3]):-
   color_string_before(S,C),
   opposite_color(C1,C),
```

```
number_liberties_string_before(S, 2),
no_adjacent_string_atari_before(S),
no_adjacent_string_two_liberties_before (S),
liberty_before(I2,S),
no_adjacent_string_atari_if_move_before(S,[I2],[C1]),
min_liberties_if_move_before(I2,C1,[],[],3),
no_capture_if_move_before([I2],[C1]),
liberty_before(I3, S),
I3=\=I2,
nb_liberties_if_move_before(S,C,[I2,I3],[C1,C],1),
liberty_if_move_before(I1,S,[I2,I3],[C1,C]),
I1=\=I3.
```

4.2 Use of generated rules to improve search

The search consists in the development of an AND-OR proof tree. In the program two searches are performed, one where the opposite color of the string to capture moves first, and tries to capture the string, and one where the color of the string moves first and tries to save the string. For each tree, the program tries to find as much winning moves as it can. So it continues to develop the search tree until no more interesting worthwhile moves are found, or until the maximum number of nodes for the search is reached.

The most interesting rules to reduce the search effort are the rulse about forced moves. They enable to select a small number of moves at the AND nodes of the search tree (the nodes where the player tries to save the string). Other useful generated rules are the rules about the winning moves: they are used at the OR nodes of the proof tree to detect that the string is captured some moves ahead, therefore saving search effort, and replacing it with the match of winning rules. In these experiments about capturing strings in the game of Go, we do not use the rules which find the moves that work if another move of the same color is played after. For the capture of strings, there is a simple function that finds the moves worth trying. However in many other games, there is not such a simple function, so we use these rules to find the worthwhile moves at the OR nodes of the search tree. The moves that work if another move of the same color is played after, always lead to a position where rules on winning and forced moves apply. That is why these rules are used at the OR nodes of the search tree: they give the moves that lead to positions with forced moves.

Figure 6 gives an example of a proof tree developed with rules about forced and winning moves five moves ahead.

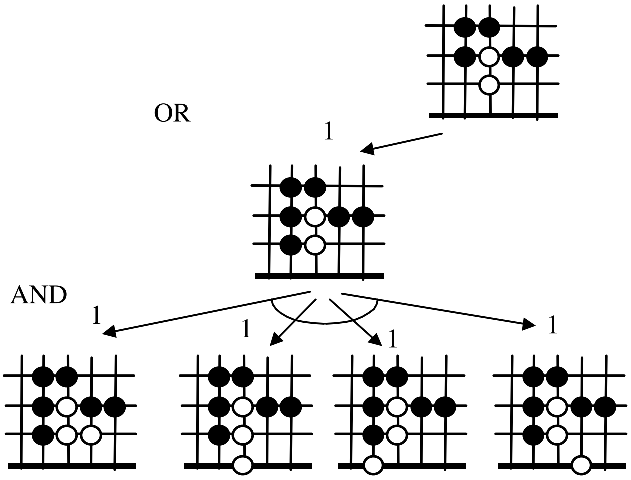


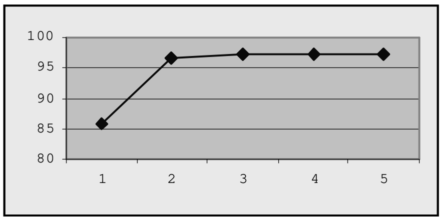
Figure 6. A proof tree.

At the AND node, three rules about forced moves are matched. The intersection of the three sets of moves given by these three rules gives the set of four moves that are played. For each of the following positions, a rule on a wining move five moves ahead is matched. So the search is stopped and returns 1. Each of the four moves at the AND node returns 1, so the AND node also returns 1. Therefore the move tried at the OR node is a winning move: it captures the white string.

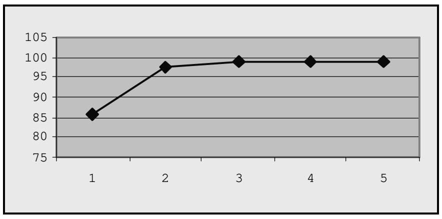
4.3 Results on a test set for the game of Go

The problem solver uses a simple and efficient hand-coded function to find the moves worth trying at the OR nodes of the search tree. The generated rules are used to find the complete set of forced moves at the AND nodes of the search tree. Proof-Number search [Allis & al. 94] is used to develop the search trees. It has a threshold on the number of nodes that can be developed. It stops its search as soon as this threshold is reached. A benefit of Proof-Number search is that it develops less nodes than Alpha-Beta, but the drawback is that it takes too much memory and time to keep incremental information about the board, so at each node the system has to compute the liberties of the interesting strings again, and although it uses a lazy evaluation and a cache to speed-up this computation, it still takes time. A comparison of the respective merits of Alpha-Beta and Proof-Number search for tactical Go problem solving is outside the scope of this paper.

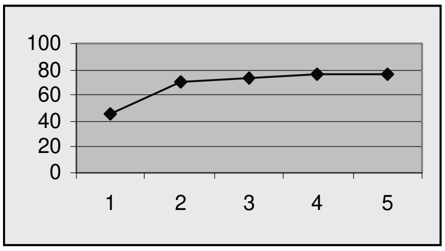
In our experiments, we used five different sets of generated rules. The first set consists of the rule on forced moves one move ahead (i.e. the string can be captured in one move if the forced moves are not played). The second set consists of the rules on forced moves one and three moves ahead. The third set consists of the rules on forced moves one, three and five moves ahead, without the rules on the strings that have three liberties. The fourth set contains the rules of the third set, and the set of rules on winning moves one, three and five moves ahead; it does not contain the winning rules about strings that have three liberties. The fifth set contains all the winning rules and all the rules on forced moves to capture one, three and five moves ahead, including the rules on the strings that have three liberties. The test set is composed of capturing problems for beginner's (Vol. 1) [Kano 85a] and advanced beginner's (Vol. 2) [Kano 85b], the problems about *semeai* (race to capture between two strings) are included in the test set.



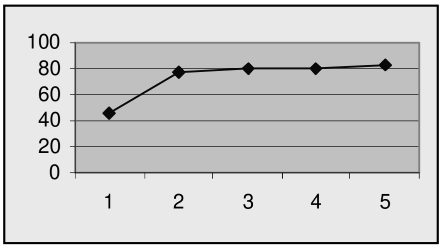
Vol. 1 – 500 nodes



Vol. 1 – 4,000 nodes



Vol. 2 – 500 nodes



Vol. 2 – 4,000 nodes

Figure 7. Results of different search knowledge on the test sets

On a given a problem, the problem solver tries to find all the moves that solve the problem, it does not stop after the first winning move is found. Therefore, if more nodes and more rules on moves worth trying are given, it will take more time to find other moves that work even if the problem is already solved according to our test set. The reason for this feature is that the problem solver is used in a playing program. Therefore, it is interesting to know all the moves that solve a problem, because some moves will also solve other problems, while other moves only solve one problem: for instance, it is better to make a move that connects two groups and that capture ten stones, than a move that only connects two groups.

Here is an example of a *semeai* problem it cannot solve yet, even with all the rules 3 moves ahead and 20,000 nodes:

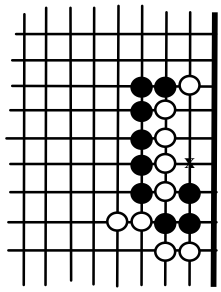


Figure 8. A race to capture that the system does not solve due to a lack of search.

The search knowledge is sufficient to solve this problem, but more than 20,000 nodes are needed. There are different optimizations that can be combined to solve this problem. One way is to make the C programs faster by optimizing the base functions corresponding to concepts, and by caching the results of these functions. Another possible improvement is to work on the splitting of the overall goal into independent sub-goals: for example it is sometimes possible to separate the search on capturing an adjacent string, and the search on saving the string at hand by playing its liberties. Presently, moves are ordered using an heuristic on the number of liberties of the string containing the intersection of the move after it has been played. More work could be done on the ordering of moves, like having a dynamic ordering that depends on statistics on the results of the moves already played in the search tree. The last improvement detects equivalent moves, for example in case of a race to capture: playing anyone of the external liberties is sometimes equivalent, so trying only one of them gives the same result in much less time.

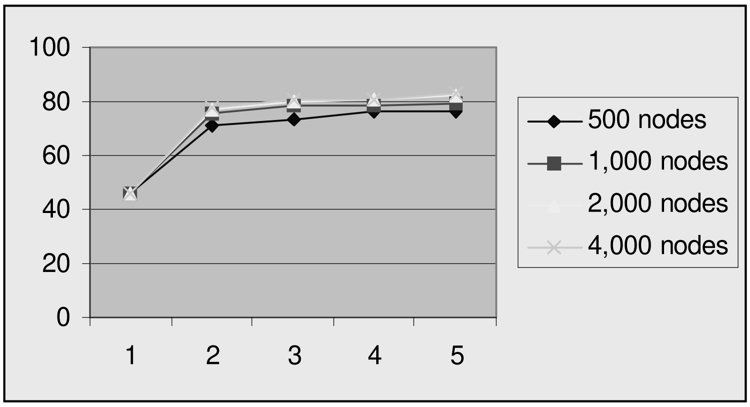


Figure 9. Compared results with different search thresholds.

Figure 9 compares the percentage of problems solved for Volume 2, with different search knowledge and different search threshold. It appears that there is a balance between adding more search or adding more generated rules to the problem solver. The merit of our approach is that it becomes as easy to tune the knowledge used by the problem solver as it is to tune the search threshold. Given processing power and memory limitation, the best configuration can be chosen easily by making tests as the ones in the appendix.

Stopping the search one to five moves ahead by using the rules on the winning moves does not speed-up the search as much as was expected. The number of nodes is approximately 2/3 of what it is without the rules, but the time is only 9/10. This is mainly because the time spent on searching cut down nodes is now spent on the match of winning rules. But, these results also highly depend on the search algorithm.

5 Conclusion

Metaprogramming in games is considered as an interesting challenge for AI [Pell 94, Pitrat 98]. Moreover it has advantages over traditional approaches: metaprograms automatically create the rules that otherwise take a lot of time to create, and the results of the search trees developed using the generated programs are more reliable than the results of the search trees developed using traditional heuristic and hand-coded rules.

The Go program that uses the rules resulting of metaprogramming has good results in international competitions [Fotland & Yoshikawa 97], (ranking between 5 and 12 out of more than forty different Go programs). The metaprogramming methods presented here can be applied in many games and in other domains than games. They have been applied to other games like Abalone, Go-Moku, Phutball, Robotic soccer, and to planning problems. Metaprogramming is particularly suited to automatically create complex, efficient and reliable programs in domains that are complex enough to require a lot of search knowledge.

6 References

Allis L. V., van der Meulen M., van den Herik H. J. *Proof-number search*. Artificial Intelligence 66, pp. 91-124, 1994.

Barklund J. : *Metaprogramming in Logic*. UPMAIL Technical Report N° 80, Uppsala, Sweden, 1994.

Cazenave T.: *Generation of Patterns with External Conditions for the Game of Go.* Advance in Computer Games 9, 2000

Cazenave T.: *Metaprogramming Forced Moves*. Proceedings ECAI98 pp. 645-649, Brigthon, 1998.

Dejong, G. and Mooney, R.: *Explanation Based Learning : an alternative view*. Machine Learning 1 (2), 1986.

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

Fotland D. and Yoshikawa A.: *The 3rd Fost-cup world-open computer-go championship*. ICCA Journal 20 (4):276-278, 1997.

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

van den Herik, H. J.; Allis, L. V.; Herschberg, I. S.: *Which Games Will Survive ?* Heuristic Programming in Artificial Intelligence 2, the Second Computer Olympiad (eds. D. N. L. Levy and D. F. Beal), pp. 232-243. Ellis Horwood. ISBN 0-13-382615-5. 1991.

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

Kano Y.: *Graded Go Problems For Beginners. Volume One.* The Nihon Ki-in. ISBN 4-8182-0228-2 C2376. 1985.

Kano Y.: *Graded Go Problems For Beginners. Volume Two.* The Nihon Ki-in. ISBN 4-906574-47-5. 1985.

Korf, R.: *Finding optimal solutions to Rubik's Cube using pattern databases*. AAAI-97, pp. 700-705, 1997.

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

Laird P.: *Dynamic Optimization*. ICML-92, pp. 263-272, 1992.

Lloyd J. W. and Shepherdson J. C.: *Partial Evaluation in Logic Programming*. J. Logic Programming, 11 :217-242., 1991.

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

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

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

Minton S.: *Is There Any Need for Domain-Dependent Control Information : A Reply*. AAAI-96, 1996.

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

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

Pell B.: *A Strategic Metagame Player for General Chess-Like Games*. Proceedings of AAAI'94, pp. 1378-1385, 1994. ISBN 0-262-61102-3.

Pettorossi, A. and Proietti, M.: *A Comparative Revisitation of Some Program Transformation Techniques*. Partial Evaluation, International Seminar, Dagstuhl Castle, Germany LNCS 1110, pp. 355-385, Springer 1996.

Pitrat J.: *A program for learning to play chess*. Chen ed. Pattern recognition and artificial intelligence, Academic press, pp. 399-419, 1976.

Pitrat, J.: *Games: The Next Challenge*. ICCA journal, vol. 21, No. 3, September 1998, pp.147-156, 1998.

Ram, A. and Leake, D.: *Goal-Driven Learning*. Cambridge, MA, MIT Press/Bradford Books, 1995.

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

Tamaki H. and Sato T.: *Unfold/Fold Transformations of Logic Programs*. Proc. 2nd Intl. Logic Programming Conf., Uppsala Univ., 1984.

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

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

Appendix

Number of forced rules	1	4	50	50	215
Lines of C for forced rules	28	126	3346	3346	22034
Number of winning rules	0	0	0	50	210
500 nodes, Vol.1					
% solved problems	85,96	96,49	97,37	97,37	97,37
Total time	37,6	44,52	45,42	39,11	46,6
Total Number of Nodes	8223	20107	21709	8360	7759
Time/Node	0,46	0,22	0,21	0,47	0,60
500 nodes, Vol.2					
% solved problems	45,83	70,83	73,61	76,39	76,39
Total time	50,12	68,03	73,92	70,31	94,69
Total Number of Nodes	10005	35772	40293	31860	31591
Time/Node	0,50	0,19	0,18	0,22	0,30
1,000 nodes, Vol.1					
% solved problems	85,96	96,49	98,25	99,12	99,12
Total time	36,82	43,26	47,2	38,9	57,38
Total Number of Nodes	9003	30128	34112	13131	12669
Time/Node	0,41	0,14	0,14	0,30	0,45
1,000 nodes, Vol.2					
% solved problems	45,83	75,69	78,47	78,47	79,17
Total time	50,02	72,85	82,72	78,25	116,2
Total Number of Nodes	11531	52846	63513	49690	50966
Time/Node	0,43	0,14	0,13	0,16	0,23
2,000 nodes, Vol.1					
% solved problems	85,96	97,37	99,12	99,12	99,12
Total time	37,24	45,05	50,38	40,31	54,12
Total Number of Nodes	10431	46317	53966	20257	19553
Time/Node	0,36	0,10	0,09	0,20	0,28
2,000 nodes, Vol.2					
% solved problems	45,83	77,08	79,86	80,56	81,94
Total time	49,73	77,19	90,3	86,36	147,9
Total Number of Nodes	11489	73164	93401	71523	79484
Time/Node	0,43	0,11	0,10	0,12	0,19
4,000 nodes, Vol.1					
% solved problems	85,96	97,37	99,12	99,12	99,12
Total time	44,08	46,8	54,85	43,32	66,45
Total Number of Nodes	10750	68957	82154	32232	31772
Time/Node	0,41	0,07	0,07	0,13	0,21
4,000 nodes, Vol.2					
% solved problems	45,83	77,78	80,56	80,56	82,64
Total time	50,46	79,34	97,85	95,33	187,6
Total Number of Nodes	11488	97252	130163	99213	113652
Time/Node	0,44	0,08	0,08	0,10	0,17