Proceedings ECAI-98
http://www-apa.lip6.fr/~cazenave/ecai98.ps

Metaprogramming Forced Moves

Tristan Cazenave[1]

Abstract. Knowledge about forced moves enables to select a small number of moves from the set of possible moves. It is very important in complex domains where search trees have a large branching factor. Knowing forced moves drastically cuts the search trees. We propose a language and a metaprogram to create automatically the knowledge about interesting and forced moves, only given the rules about the direct effects of the moves. We describe the successful application of this metaprogram to the game of Go. It creates rules that give complete sets of forced moves.

1 INTRODUCTION

Knowledge about forced moves enables to select a small number of moves from the set of possible moves. It is very important in complex domains where search trees have a large branching factor. Knowing forced moves drastically cuts the search trees. We propose a language and a metaprogram to create automatically the knowledge about interesting and forced moves, only given the rules about the direct effects of the moves. We describe the successful application of this metaprogram to the game of Go. It creates rules that give complete sets of forced moves.

The second section describes computer Go. The third section uncovers the goal of metaprogramming. The fourth section is an introduction to the metalanguage Introspect. The fifth section shows how rules concluding on the moves to try at OR nodes are created. The sixth section describes how rules concluding on forced moves are created using metaprogramming. The last section gives the results of our computer Go system.

2 COMPUTER GO

2.1 The game of Go

Go was developed three to four millennia ago in China; it is the oldest and one of the most popular board game in the world. Like chess, it is a deterministic, perfect information, zero-sum game of strategy between two players. In spite of the simplicity of its rules, playing the game of Go is a very complex task. Robson [16] proved that Go generalized to NxN boards is *exponential in time.* More concretely, Van den Herik [19] and Allis [1] use complexity measures of different games to compare them. They define the *whole game tree complexity* A. Considering the average length of actual games L and average branching factor B, we have $A = B^L$. The *state-space complexity* of a game is defined as the number of legal game positions reachable from the initial position of the game. In Go, L≈150 and B≈250 hence the game tree complexity $A \approx 10^{360}$. Go state space complexity, bounded by $3^{361} \approx 10^{172}$, and game tree complexity are far larger than those of any other perfect-information game. Moreover, a position takes time to evaluate, on the contrary of chess where positions can be evaluated very fast. This makes Go very difficult to program. Computer Go has been recognized as a challenge for Artificial Intelligence [17].

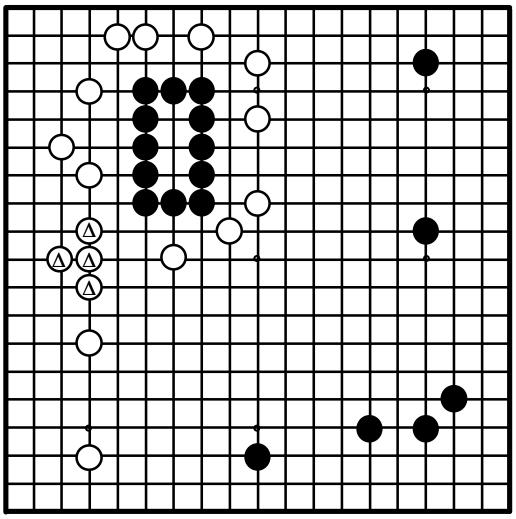

Figure 1

The board is made of 19 vertical lines and 19 horizontal lines and therefore 361 *intersections*. At the beginning the board is empty. Each player (Black or White) moves alternatively in adding one *stone* on an empty intersection. Two adjacent stones of the same color are *connected* and they are part of the same *string*. For example, the white stones of Figure 1 marked with Δ are connected and are part of the same string. Empty adjacent intersections of a string are the *liberties* of the string. The string of four marked white stones of Figure 1 has eight liberties. When a move fills the last liberty of a string, this string is removed from the board. The repetitions of positions are forbidden. According to the possibility of being captured or not, the strings may be *dead* or *alive*. A player *controls* an intersection either when he has an alive stone on it, either when the intersection is empty but adjacent to alive stones. The aim of the game is to control more intersections than the opponent. The game ends when the two players pass.

In spite of the simplicity of the rules, a Go player uses a lot of concepts to understand a position and to play a move. This paragraph briefly shows some intuitive definitions of these concepts. At the lower level, a player looks at the *safety* of the strings in performing look-ahead. When a string has enough liberties, the string is said to be safe. A player also checks if an intersection is controlled by one player or not. An *eye* is a small enclosed area, Figure 2 gives an example of an eye on intersection A. In this figure, B is one of the four *diagonal* intersections of A. When searching to make an eye, it is important to control diagonal intersections.

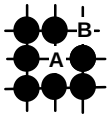

Figure 2

A virtual *connection* is a configuration that enables to connect

[1] LIP6, Université Pierre et Marie Curie, 4 Place Jussieu, 75252 Paris Cedex 05, France. Tristan.Cazenave@lip6.fr

strings whatever the opponent plays. Figure 3 gives an example of a 'Bamboo join'. If the white player plays at A, black plays at B and connects its stones. If white plays at B, then black at A connects. The four stones are virtually connected.

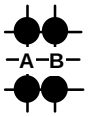

Figure 3

Using these tactical results, a Go player starts its strategic reasoning with the use of *groups*. A group is a complex concept for human players. It may be either a set of intersections that are virtually connected, either a set of intersections that gather the same properties. A group has a *status*. A status is dead or alive and it is derived from other intuitive concepts like *influence*, *fight*, *circling*, *life-base*. The reader does not need explanations of these concepts to understand the following sections.

2.2 Different levels in a Go program

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

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

Strategic reasoning is concerned with groups of stones. A group of stones is a set of stones of the same color, each stone can be connected to each other.

The tactical module uses rules to decide what moves to try in the search trees. The strategic module uses the results of the tactical module to create the groups and calculate their properties. Then it chooses the move that maximizes its territory.

3 THE GOAL OF METAPROGRAMMING

The goal of metaprogramming, in our system, is to write programs that write other programs that enable to safely cut search trees, therefore enabling great speedups. In our applications to games, metarules are used to create theorems that tell what are the interesting moves to try to achieve a tactical goal (at OR nodes). 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).

Each time our Go program tries to see the degree of achievement of a goal, it develops two AND/OR proof trees. One with the friend color playing first, and the other with the enemy color playing first.

The Figure 4 gives an example of a proof tree for the goal connect. Black is the friend color, and the goal is to connect the two black stones. The first move works (the leftmost arrow) and as it is an OR node, the other branches are cut. The moves at the OR nodes are given by rules that conclude on moves that can achieve the goal if two moves in a row by the friend color are played. This heuristic is used because these moves lead to positions containing threats to win by the friend player, and therefore forced moves for the opponent player.

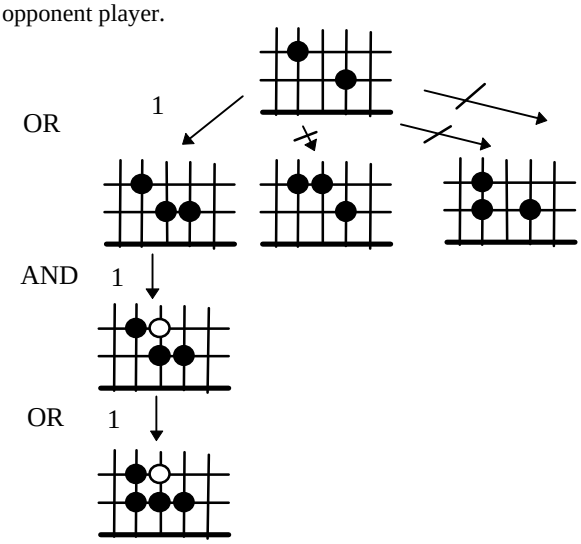

Figure 4

We give a visual and intuitive definition of the rules by selecting the part of the board corresponding to the conditions of the logical rule. The rules conclude on a move or a set of moves. This is represented using arrows that lead to positions after each move in conclusion. For example, the first visual rule of Figure 5 is represented by the following logical rule ('S1' and 'S2' are the two strings, 'I' and 'I1' the empty intersections and 'C' the color):

```
connect (S1, S2, I, C) :- color (C), color
(S1,C), color (S2,C), liberty (I, S1),
liberty (I1, S2), neighbor (I, I1).
```

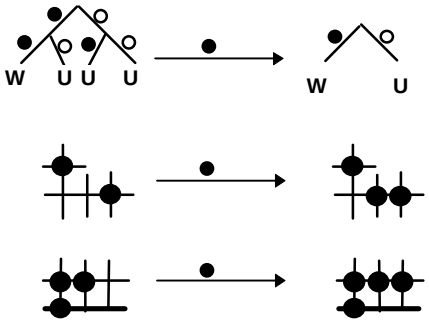

Figure 5

The rules of Figure 5 are rules that tell the moves to try at the OR levels of the proof trees. This is visually explained in the first diagram of Figure 5, where a tree is represented with the color of the moves associated to the branches. We can see that the only available information is that Black can win the goal if it plays two moves in a row (state W of the first diagram), otherwise the three other combination leads to unknown situations (state U). When Black plays the moves advised by the rule, it switches to a threatening situation represented by the tree on the right of the first diagram. Black can now win the goal if it plays one move, and therefore White has now to play to prevent Black from doing so.

The first rule of Figure 5 is used to find the upper left move of the proof tree of Figure 4. The second rule of Figure 5 is a rule advising a move to try to make an eye.

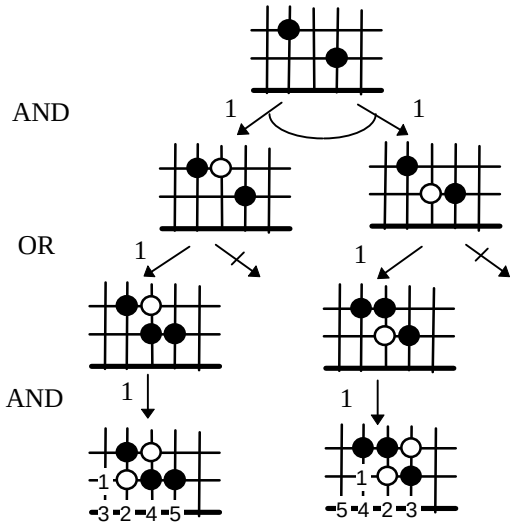

Figure 6

The Figure 6 gives the second proof tree developed by the Go program with White playing first, and the goal Connect the two black stones. In order to save place, we have numbered at the leaves of the tree the sequences of moves (odd numbered moves are Black moves and even numbered moves are White single forced moves) that lead to the winning positions for Black. The two forced white moves at the root of the proof tree are refuted by Black. So the result of the two proof trees is that Black can connect its two stones even if White plays first: the two black stones are virtually connected.

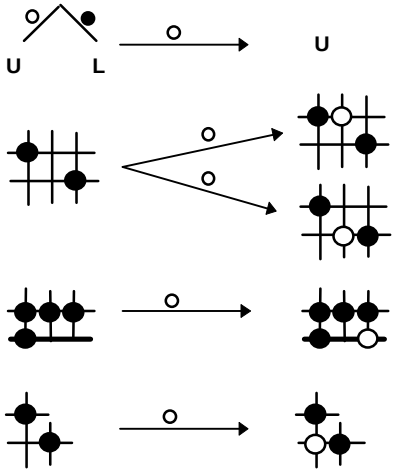

Figure 7

The last rule of Figure 7 is used to find the forced move at the lowest AND node of Figure 6. The first rule of Figure 7 is used to find the two forced moves at the root of the proof tree of Figure 6: note that these two moves are the only ones that need to be considered out of all the possible moves. The second rule is used to find the only forced move to prevent an eye. The visual definition of a forced move is given in the first diagram: a White move leads to an unknown state (state U), whereas a Black move leads to a lost state for White (a lost state L for White is a winning state W for Black: White loses if Black is connected in our example).

4 THE INTROSPECT LANGUAGE

Metaprogramming in logic has already attracted some interest [14] [2] [9]. More specifically, specialization of logic program can be traced back to [18], it has been well defined and related to Partial Evaluation in [11], and successfully applied [8].

Introspect [4] is a metaprogramming system based on predicate logic. On the contrary of Prolog programs, The Introspect programs are really declarative: the result of their execution does not depend on the resolution strategy. Prolog is based on the SLDNF strategy, it means that the order of clauses is important and it is harmful for the declarativity of the programs. A fixed resolution strategy is harmful when one wants to speed up a concise but inefficient logic program into an efficient but usually longer logic program by specializing it. It prevents the reordering of the atoms inside the clauses and the reordering of the clauses. A good reordering of the specialized programs can lead to large speedups [3]. For similar reasons, we do not use negation as failure. Moreover, we can have multiples atoms in the head of a clause and we use forward chaining.

Introspect uses metapredicates that enable it to manipulate its own programs. We will describe some of them in this section. They will be used to explain how to metaprogram forced moves. Constants and variables are typed. Conventionally, variables begin with a capital letter, whereas constants begin with a tiny letter. Each example of a metapredicate is followed by a definition.

rule (R) : Instantiates in the variable R all the rules of the object program.

conclusion (R, P) : Instantiates in P the atoms in the head of the rule R.

conclusion (R, Color (V1)) : Instantiates in the variable V1 all the variables and constants of the head of R that are arguments of the Color predicate.

initcut () : Affects 0 to the internal variable that test is the conclusion of a rule has been found. This affectation is done for each instantiation of each variable above this metapredicate.

cutifdeduction () : Stops backtracking if a conclusion with the current instantiations of variables has been found. It is different from the traditional cut operator of logic programming.

setofmodifyingmoves (SET, SET1, SET2) : Puts in the set variable SET the set of moves that enable to change any of the conditions contained in the set of conditions SET2. SET1 returns the set of conditions that have to be added to SET2 so that the move of SET are assured to change at least one of the condition of SET2. This is a metapredicate that calls a logic program that uses the rules of a game to build SET and SET1 using SET2.

condition (R, oppositecolors (V1, V2)) : Instantiates in the variables V1 and V2 the variables and constants of the rule R that are argument of the predicates oppositecolors.

Introspect uses numerous metapredicates. Most of them can be understood intuitively given their names. We will not define all of them here.

5 METAPROGRAMMING OR NODES MOVES

The rules used to decide the moves to try at the OR nodes of the search trees are automatically created by an Introspect metaprogram that partially evaluates the target concepts using domain knowledge. Introspect is also able to specialize the target

concepts using examples. Systems like Introspect that learn by specializing goals have been formalized as Explanation Based Learning/Generalization systems [12] [10] [6]. They have received attention more recently in [15]. The first application to games can be traced back to [13]. The relation to Partial Evaluation in logic programming has been uncovered in [20].

The formalism used to represent the rules is first order predicate logic. The rules are programmed by Introspect, only given the rules of the game in predicate logic.

The target concepts are the tactical subgoals of the game of Go : Remove a string, Make a string alive, Connect two strings, Disconnect two strings, Make an eye and Remove an eye. Each of these target concepts is defined using rules in predicate logic. For example the target concept for the tactical goal removestring is defined using this rule:

```
removestring (S, I, friend) :- color (S, enemy), move (I, enemy), numberoflibertiesbeforemove (S, 1), liberty (I, S), legalmove (I, enemy).
```

Thousands of rules are created by using the rules of the game to specialize the tactical goals. Example of a simple created rule used to find connections between strings of stones :

```
connect (S1, S2, I, C) :- color (C), color (S1, C), color (S2, C), oppositecolors (C, C1), liberty (I, S1), liberty (I, S2).
```

This rule tells that if an intersection I is a liberty of strings S1 and S2 that have the same color C, playing a stone of color C at I enables to achieve the goal Connect between the two strings.

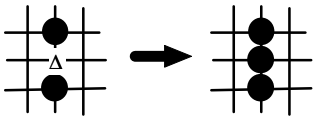

Figure 8

The created rule previously given as example applies to the move marked Δ that connects the two black strings in the Figure 8. The initial target concept defining the connect goal is:

```
connectedaftermove (S1, S2) :- color (C), color (S1, C), color (S2, C), oppositecolors (C, C1), elementofaftermove (I, S1), elementofaftermove (I, S2).
```

The rules used to specialize the target concept in this example are:

```
elementofaftermove (I, S) :- liberty (I, S), color (S, C), move (I, C).
```

```
connect (S1, S2, I, C) :- move (I, C), connectedaftermove (S1, S2).
```

There are different predicates to describe the board after the move and the board before the move. This is to prevent side effects to happen, and to avoid incomplete explanations. The predicate 'move (I, C)' is retracted from the created rule because it is redundant.

The specialized rules used at OR nodes of the proof tree give moves that achieve the goal if they are followed by another move of the same color, as the rules in Figure 5.

6 METAPROGRAMMING FORCED MOVES

Metaprograms can be used to automatically create programs. We give in this section an explanation of the metarule that enables to create rules concluding on forced moves, given rules concluding on winning moves.

```
addconclusion (R1, forcedmovetolive (V4, V2, SET)), addrule (R1) :-
  rule (R),
  conclusion (R, movetotake (V1, V2, V3)),
  condition (R, color (V1)),
  condition (R, oppositecolors (V1, V4)),
  setofconditions (R, SET2),
  setofmodifyingmoves (SET, SET1, SET2),
  length (SET, N),
  greaterthan (5, N),
  newrule (R1),
  addconditiontoset (R1, SET1),
  addconditiontoset (R1, SET2).
```

The rules concluding on winning moves are also created by Introspect as described in the previous section.

The first condition of this metarule, 'rule (R)', selects all the rules and instantiate them in the variable R. The second condition selects among the rules those that conclude on a winning move for color V1 (V1 contains either a constant or a variable representing a color, V2 contains a variable representing a string to Take, and V3 contains a variable representing an intersection where to play the winning move), with V4 containing the color opposite to V1. Note that if there is a winning move for color V1, then the forced moves to prevent V1 to win have the color V4. The next condition 'setofconditions (R, SET2)' puts into SET2 the set of conditions of the rule R selected. Then the condition 'setofmodifyingmoves (SET, SET1, SET2)' calls a metaprogram specific to the rules of the game that fills SET with the forced move to prevent V1 to make the winning move, and that fills SET1 with a set of conditions that ensure that the moves in SET are the complete set of forced moves. After that, the metarule verifies that there are less than 5 forced moves, creates a new rule R1 and adds SET1 and SET2 to the set of conditions of R1. Eventually, the metarule adds in the conclusion of the rule R1 the predicate containing the set of forced moves.

We enforce rules to select less than five forced moves. This has two justifications: having simple rules with a small number of conditions (less than 200), and not searching too many forced moves. Forced moves are used to develop the AND nodes of the AND/OR search tree. So all the branches of an AND node have to be proved before the node can be set to 1. Therefore it is reasonable to try to keep the number of branches at an AND node low. Otherwise the search will be less successful, wasting time proving unnecessary things, and will solve less problems.

The set of forced moves is complete, because all the possible moves to destroy each condition of the rule R, are added to SET. The completeness ensures that all the moves that are not considered at an AND node, do not need to be considered. Therefore, if all the move to prevent to achieve a goal at an AND node are refuted, no other move can refute the goal, and the goal can be achieved for every move played to try to prevent it. Eventually, it ensures that the results of the proof trees returning 1

are theorems of the position: they are always true, the goal can always be achieved whatever the opponent plays.

The rule created with the example rule of the previous section, using a metarule similar to the previous one, is:

```
forcedmovetodisconnect (S1, S2, I, C1) :-
color (C), color (S1, C), color (S2, C),
oppositecolors (C, C1), liberty (I, S1),
liberty (I, S2).
```

It means that I is the only move to prevent S1 and S2 from being connected. This is because no moves can change the conditions 'color (C)', 'color (S1, C)', 'color (S2, C)' and 'oppositecolors (C, C1)' of the example rule, and only a move on I can change the condition 'liberty (I, S1)' and the condition 'liberty (I, S2)'.

This created rule gives the last rule of Figure 7. It is even more general because it also applies to Figure 8, with only one forced White move on Δ to prevent the two Black strings to connect.

Note that rules about achieved goals are used to create rules about winning moves, that are themselves used to create rules about forced moves, that are in turn used to create rule about achieved goals (when no forced move works). This enables the system to incrementally create more and more complex rules, until no more rules can be created. The actual limits are set to five forced moves and less than 200 conditions in the created rules.

7 RESULTS

The rules resulting of metaprogramming enable to safely consider only between 1 and 5 moves out of the 250 possible moves on a board. They strongly decrease the size of the brute force search tree and the time to develop proof search trees. This enables our Go program to look as far as 60 moves ahead in some tactical positions.

The Go program 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 has learned thousands of tactical rules. All the learned rules are compiled into a 1 000 000 lines C++ program.

Gogol competed in the international computer Go tournament held during IJCAI97 together with 40 other participants. It finished 6 out of 40 participants [7]. The five first programs are commercial programs that have required a lot of person*years of work. It has outperformed other commercial systems that have required more than 10 person*years of work.

8 CONCLUSION

We have shown that metaprogramming forced moves enables to drastically reduce the number of nodes of the search trees and the time to compute the search trees. It is an improvement on the search algorithms used in Go playing programs that rely on many hand-coded rule to heuristically cut search trees. Our approach has two advantages over the traditional approach : we automatically create the rules that otherwise take a lot of time to create, and the results of our search trees 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 the metaprogramming has good results in international competitions (6 out of 40, best non-commercial program). We are currently applying Introspect to other domains [5]. Our approach is particularly suited to automatically create complex, efficient and reliable programs in domains that are complex enough to require a lot of knowledge to cut search trees.

9 REFERENCES

[1] Allis, L. V. 1994. Searching for Solutions in Games an Artificial Intelligence. Ph.D. diss., Vrije Universitat Amsterdam, Maastricht.

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

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

[4] Cazenave, T. 1996b. Système d'Apprentissage par Auto-Observation. Application au Jeu de Go. Ph.D. diss., Université Paris 6.

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

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

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

[8] Gallagher J. 1993. *Specialization of Logic Programs.* Proceedings of the ACM SIGPLAN Symposium on PEPM'93, Ed. David Schmidt, ACM Press, Copenhagen, Danemark, 1993.

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

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

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

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

[13] Pitrat J. 1976. Realization of a Program Learning to Find Combinations at Chess. Computer Oriented Learning Processes, J. C. Simon editor. NATO Advanced Study Institutes Series. Series E: Applied Science - N° 14. Noordhoff, Leyden, 1976.

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

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

[16] Robson, J. M. 1983. The Complexity of Go. In Proceedings IFIP, 413-417.

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

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

[19] Van den Herik, H. J.; Allis, L. V.; Herschberg, I. S. 1991. 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.

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