A Step toward an Artificial Artificial Intelligence Scientist

Jacques Pitrat[1]
LIP6
Université Pierre et Marie Curie

Abstract

This paper describes the system CAIA which must perform some activities of an AI scientist. It includes five agents: the most important ones are MALICE which finds elegant solutions to constraint satisfaction problems, MONITOR which monitors MALICE, and MANAGER. This last agent chooses the tasks it will perform, writes combinatorial programs for solving specific problems, generates the knowledge used by MALICE, performs experiments, explains why an elegant solution has not been found, generates new problems, finds symmetries in the formulation of a problem, gathers the information necessary to find a bug, searches for anomalies in the results, tries various methods to solve difficult problems, etc. CAIA has been working several weeks autonomously. Finally, the results of a "life" of this artificial AI scientist are given and compared with those obtained by human AI scientists.

Keywords: autonomous systems, artificial scientist, learning, constraint satisfaction problems.

1. Creating an Artificial Scientist

1.1. A goal for AI

Basically, one main goal of AI is to create an artificial AI scientist: if we succeed, AI will progress alone, this artificial scientist will develop new systems to solve new problems, and ultimately will create an artificial AI scientist even better than its artificial author. This will not certainly happen in the next few years, but it is necessary to launch less ambitious projects leading in that direction.

Such systems can solve the problems given by a user, but their main goal is to improve themselves so that they perform well on all of the possible problems in the domain they are familiar with. In order to train themselves, they make experiments, create new problems, understand the reasons for their inefficiencies, etc. Such systems have no « Stop » order, new facts can always be discovered or better understood. They must have some of the following features:

To manage to perform a meta-theoretical study of their domain so that they create new tools to solve problems.
To build a new problem solving method and to improve it constantly.
To write, if necessary, a specific program for solving one particular difficult problem.
To perform some experiments where it may find surprising events and to use what it understands from this experiment so that it learns.
To discover new useful concepts in order to improve the resolution of the problems.
To create new problems so that it can practice with different and more difficult problems.

No present system can deal with all of these aspects. However, some systems have some of

1 *E-mail address*: pitrat@lip6.fr (J. Pitrat)

them:

The first idea of such a system was probably considered in a paper by Newell, Shaw and Simon in 1960 [28]; it was an improvement of their General Problem Solver (GPS). For solving a problem, GPS finds a difference between the present state and the goal, then chooses an operator relevant to this difference and applies it. However, it is difficult to find a good set of differences for each problem domain. As GPS is a general problem solver, their idea was to use GPS to solve the meta-problem: find a good set of differences for a particular task environment. For this meta-problem, it is necessary to have another set of differences, which are meta-differences between a poor set of differences and a good one. Unfortunately, this idea was too ambitious for what was feasible at that time for them to implement: only hand simulations of the behavior of the system were tested.

From 1960 to 1966, I developed the system THEOREME [31,32,33]; its goal was to be a mathematician. It received the formulation of a mathematical theory: its axioms and its derivation rules. It had to prove theorems in this theory, but it was able to find new theorems, and also new meta-theorems, which were new derivation rules. For each theory it received, it discovered new ways to prove theorems and built a toolbox for this theory. It received six axiomatizations of the propositional logic and was able to find and prove theorems for which Lukasiewicz, one of the best experts in these axiomatics, said [23] "One must be very expert in performing such proofs".

With AM and EURISKO [7,18,19,20], Douglas Lenat built systems which could find (but not prove) new conjectures and new concepts in arithmetic for AM and in several domains for EURISKO. The success of this system mainly depended on its ability to observe what happened, to understand the reasons of its successes and failures, to perform experiments, to generate new rules so that it could continue to build better concepts. As with THEOREME, the system creates a toolbox of new rules.

Simon Colton [4,5] realized the system HR which forms concepts and makes conjectures in domains of pure mathematics. It uses a theorem prover to prove or disprove the conjectures. It has successfully been applied to three different domains: graph theory, group theory and number theory. In each domain, the system starts with a few basic concepts; using only seven general rules, which are the same for all of the domains, it builds many more sophisticated concepts. It has defined several sequences of integers which were accepted in the Encyclopedia of Integer Sequences.

Bruce Buchanan and Gary Livingston [3] have developed a framework for autonomous discovery in empirical domains; it peruses large collections of data to find interesting hypotheses. This program is able to reason on its priorities and uses domain-independent heuristics to guide the program's choice of relationships in data that are potentially interesting. A prototype, HAMB, was demonstrated successfully in the domain of protein crystallization.

Marvin Minsky, Push Singh and Aaron Sloman [26] suggest building a machine that will use a diverse array of resources, which together will deal with a great range of problems. Using a causal diversity matrix, each problem-solving method is assessed in terms of its competence in dealing with problem domains which have different causal structures. They want to define a meta-theory of AI techniques. Such a machine would have human-like common sense.

Several AI scientists [13] have implemented systems that are artificial scientists in various domains. They used historical knowledge on the data and the methods that enabled human scientists to make these discoveries many years ago. In the same way, it would be useful to know how human AI scientists are developing their systems; in some papers, for instance in the description of a solver for Sokoban problems [10], the steps that lead to a successful program are also given: several of these steps could easily be automatized. Such AI papers are very useful for our goal.

1.2. CAIA's research domain

CAIA is an acronym for "Chercheur Artificiel en Intelligence Artificielle", that is an Artificial Artificial Intelligence Scientist. It is an AI system, which is able to perform some of the activities of an AI human scientist. It is not able to work in every domain, its present domain is solving constraint satisfaction problems. This domain was chosen because there is already a powerful method for finding elegant solutions to such problems with a formalism convenient to define many kinds of such problems. It solves problems and performs experiments; it uses the results of some of these experiments for improving its problem solving abilities. The other experiments show the strengths and the weaknesses of the system which could be used for further developments by human or artificial scientists.

Jean-Louis Laurière developed ALICE [14,15], a very efficient solver for such problems, and it is an excellent starting point for CAIA's problem solving module called MALICE. ALICE does not begin with developing a tree, but it first works at the meta-level on the formulation of the problem. For instance, it generates new constraints and uses them to cut the search space. ALICE solved a great variety of problems and its results were excellent: ALICE's solutions are very elegant and can be easily understood by human beings. Its solutions are sometimes better than those of a good human problem solver because it develops a smaller tree: the reason is that it considers many methods to solve a problem. As it is sufficient to keep only the methods that succeed in order to justify a solution, the tree explaining it may be very small. Moreover, in some cases [16], it occurred that this general problem solver was able to solve a problem faster than a program written specifically for solving only this problem.

It is quite surprising that a general system would be better than a specific program, but there are three reasons for this: more knowledge, dynamic choices and use of partial information. The first reason is evident: when a system must solve a large variety of problems, it appears that a particular method is essential for one of them. Thus, we include it in the system which often uses this method successfully when it believes that it is useful, and in some cases, for problems where the human programmer does not think so. When ALICE is right, its performance is better than that of the specific program. But ALICE also dynamically determines the unknown chosen for backtracking; usually, a specific program takes them in a predetermined order. When the best choice strongly depends on the values given to the already instantiated unknowns, the specific program may perform very poorly. Finally, the third reason for the superiority of ALICE is that it is often able to prove that some constraints are false even without knowing the value of all their unknowns; it is easier to write specific programs which must know the value of all the unknowns of a constraint for assessing whether it is true or false. In that case, these programs needlessly develop a very large tree including the possible values of all the unknowns, when most of them are unnecessary. We will later see that for some problems, MALICE is actually faster than specific programs; in that case, the most efficient specific program must be identical to MALICE!

Moreover if ALICE's methods are the best for solving some of the problems, this is not always true, and for the other problems, the best method is to write a specific combinatorial program. Several experiments [17,22] have been made for generating efficient programs for solving problems defined in ALICE's formalism.

The other main reason for choosing constraint satisfaction problems is that ALICE's formalism is powerful and convenient: we can define many problems in a more pleasant way than in the first order predicate calculus. This generality is interesting because we can define not only problems, but also some meta-problems, which are problems useful for CAIA when it is performing AI research. Two kinds of meta-problems are defined and solved as constraint satisfaction problems: finding symmetries, which are variables permutations, in the formulation of a problem and generating new problems. Thus, this is a kind of a reflexivity, some meta-problems appearing while solving a problem are defined in the same formalism and solved with the same methods as those used for this problem. This reflexivity enables us to progress in a promising direction for the future of AI: bootstrapping AI. When we are bootstrapping a system, we choose the next step so that it will be helpful for solving the following steps: this is the case since two kinds of meta-problems are solved without doing no more than defining them. However, many kinds of research meta-problems have not yet been formulated and CAIA is far from performing all of the activities of a scientist: this work is only a step in that direction.

In this paper, I do not describe some parts of the system which still need to be developed. For instance, the system performs experiments, analyzes them and finds surprising results; from these results, it is possible to learn some improvements, but the learning module, which would be able to find them, is not yet written. In that case, only a part of the system has been implemented: on how to find interesting data, but still not on how to use them.

Many scientists have worked on constraint satisfaction problems and have found efficient methods for solving some of them. A human scientist begins with studying the related work and tries either to improve the best methods or to discover a new method better than all the existing ones. However, CAIA works from first principles. One reason for this choice is that realizing a system which is able to read and understand previous work is a difficult task. Moreover the ultimate goal of AI is to realize systems that can have an intelligent behavior, without depending on human intelligence; thus it is better to reduce human intervention as much as possible. Even if a system finds results that were already found by human scientists, it is useful if it knows neither these results nor the methods that were used: in another domain which has not yet been studied by human scientists, it is likely that it will find new interesting results.

Several artificial mathematicians, such AM and HR, are also starting from first principles. When describing an artificial mathematical scientist, one does not cite all the mathematicians who have made tremendous progress in the domain, but only the works which are compared with the results of the system. In the same way, I do not cite the authors whose work has not been used for CAIA even though they have found remarkable improvements for solving constraint satisfaction problems, for instance in discovering symmetries in the formulation of these problems. I only cite works that have been used in the conception of the system and works whose results are compared with those of CAIA.

1.3. Organization of the paper

I will describe the system CAIA and the results generated during one of its "lives". The goal of this system is to perform some tasks that are usually performed by human AI scientists. It is difficult to describe such a system because it includes eight thousand conditional actions. It is not possible to begin with papers describing parts of the system because their meaning only appears when we have a complete view of the system. Thus, I describe here the whole system, but most modules are presented in a very simplified form. Many aspects are not even mentioned, I only include what is necessary to have a comprehensive view of this system. It is not possible to give all the details on each of the modules: many subsections of this paper could be developed into separate papers of their own.

I begin to describe how CAIA solves a difficult problem, a kind of magic cube where the space search is huge. Then I will present another system, MACISTE, which can be considered as an operating system for CAIA. A part of CAIA's knowledge is stated as conditional actions and MACISTE translates them into efficient C programs, manages the storage, helps the debugging, etc. Moreover, the language used for describing logical expressions in MACISTE is often used for describing CAIA's knowledge when this knowledge is given (or generated by CAIA itself) in a form which is more declarative than conditional actions.

We will see the language used for defining families of problems so that MALICE can formalize all of the problems of a family as one set of constraint generators. Then we describe the general method used to choose among several candidates when reasoning is no longer possible; for each of the different kinds of choices, the user gives knowledge in a formalism which is easy to define and modify. The method then uses this knowledge to find the best candidate.

The society of mind of the system [25] includes five agents. First there is a hierarchy of four agents: MALICE, MONITOR, MANAGER, ADVISOR. Each agent is semi-autonomous, but it can ask its superior agent for advice when it seems necessary. The superior agent can also interrupt its inferior one in an authoritarian way if it considers that its subordinate would have to ask for advice or help.

At the lowest level, MALICE cleverly tries to solve a problem; this problem is given by MONITOR. It uses an ALICE like method.

The second level is MONITOR, which monitors the behavior of MALICE. When MONITOR receives a request, for instance which unknown will be chosen for backtracking, it finds and gives its choice to MALICE. Moreover, it regularly surveys whether MALICE advances toward a solution. When an anomaly is found or when MALICE is trapped in a succession of useless derivations, MONITOR takes an emergency action to correct it, for instance it forbids to use a particularly explosive derivation rule.

MANAGER is at the third level and it is the most important agent: it is the one that performs tasks similar to those achieved by a human scientist. It manages the career of the artificial AI scientist: it chooses which task takes priority, it allocates some means to this task, it possibly authorizes MONITOR, if required, to exceed them; it can also choose to perform some experiments and to analyze them, or to generate new problems.

A fourth agent, ADVISOR, is at the highest level. It assesses MANAGER's behavior and regularly checks that it is always performing interesting tasks.

A fifth agent, ZEUS, is not in the hierarchy, it is regularly called by the interrupt mechanism. It checks that the four preceding agents function adequately and do not loop. If that happens, it restarts MANAGER after storing the characteristics of its difficulties.

Finally, we will present the results of the system and consider two directions for future research on this system: solving problems given in a natural language and improving the theorem proving abilities of CAIA.

In describing such a system, the difficulty is that all of the subsystems are necessary and although we are mainly interested in MANAGER, we must also consider the other subsystems. It is mainly in sections 8 and 11 that CAIA acts as a scientist. Thus in a first lecture, it is possible to skip most of the sections 3-7, which describe important parts of the system, but are not the main objective of this realization; it would be sufficient to read the first paragraph of these sections which summarizes the function of the corresponding subpart of the system.

2. An explanatory example

Let see how CAIA works. At the beginning of a life, it knows many problems that were given to it or that it has generated in a preceding life. It begins with building a general method for solving problems. Then it will have to solve the problems given by the user; among them is a kind of magic cube: the goal is to find a bijection F between the 27 small cubes that are in a 3x3x3 large cube and the integers between 1 and 27. The sums of the planes containing 9 cubes must have the same value. There are 15 such planes, 9 horizontal and vertical planes and 6 diagonal planes. The small cubes are numbered in the following way, the first array is for the upper face and the third one is for the lower face, as shown in Fig. 1.

1	2	3
4	5	6
7	8	9

10	11	12
13	14	15
16	17	18

19	20	21
22	23	24
25	26	27

Fig. 1. Numbering of the small cubes.

The nine constraints for the horizontal and vertical planes are for instance:

$$F(1)+F(2)+F(3)+F(4)+F(5)+F(6)+F(7)+F(8)+F(9) = VAL$$

$$F(3)+F(6)+F(9)+F(12)+F(15)+F(18)+F(21)+F(24)+F(27) = VAL$$

Six other constraints are for the diagonal planes such as:

$$F(1)+F(4)+F(7)+F(11)+F(14)+F(17)+F(21)+F(24)+F(27) = VAL$$

VAL is the unknown common value of the sum for the 15 planes.

The only information received by CAIA for this problem is that it must find the value of VAL and define a bijection F between two sets, which are the integers from 1 to 27, so that the 15 constraints are satisfied.

This problem is both easy and difficult to solve. It is very easy because it is under-constrained with 15 constraints for 28 unknowns; there are millions of solutions. It is very difficult because the search space is huge, more than 28^{28}. Moreover, each constraint contains ten unknowns: in a combinatorial search, the solver must usually wait to make ten choices before it can check whether a constraint is true of false.

MANAGER begins to look for possible symmetries: among all of the different kinds of symmetries, it is only able to find permutations of the unknowns such as the set of constraints remains the same. This search has been defined as a constraint satisfaction problem and MALICE solves it using the same methods as with the main problem. In the present case, it finds 47 variable permutations; some of them are not evident, as the one given in Table 1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
9	18	27	6	15	24	3	12	21	8	17	26	5	14	23	2	11	20	7	16	25	4	13	22	1	10	19

Table 1. Example of a symmetry for the magic cube

Naturally, the image of VAL is always itself. These symmetries are used in two ways. First, the system adds new constraints, 14 for this problem, so that it does not generate symmetrical solutions. One of these new constraints is F(11)<F(17), and with this new constraint, it will not be able to generate all of the solutions which can be symmetrical with the preceding symmetry: in all of these solutions, F(11) must be less than F(17) so in their symmetrical solutions, as F(11) and F(17) are exchanged, F(11) is certainly greater than F(17) and this constraint eliminates all of them. For each new solution, MALICE can automatically generate its 47 symmetrical solutions.

MALICE also uses these symmetries for generating new constraints: when it finds a new constraint, it also generates all of the constraints where each unknown is substituted with its corresponding element. Naturally, this does not always generate 47 new constraints, different symmetries can lead to identical constraints when the initial constraint does not contain all the unknowns.

After finding the symmetries and adding the 14 new constraints, it tries to solve the problem. For that it generates new constraints from the old ones, using the rules of algebra. First, it finds that VAL=126 and then many other constraints. The most interesting is:

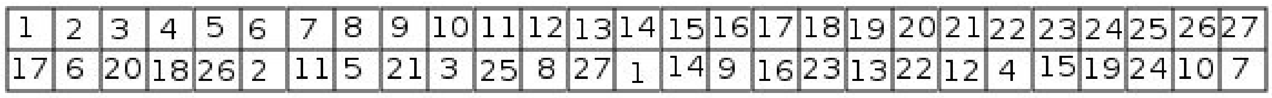
$$F(5)+F(14)+F(23) = 42$$

which includes only three unknowns. Using the symmetries, it generates two more new constraints:

$$F(11)+F(14)+F(17) = 42$$

$$F(13)+F(14)+F(15) = 42$$

Afterwards, MALICE makes no progress, it still finds many constraints but they are less interesting than those it has already found. So it asks MONITOR for help. MONITOR decides to begin backtracking, chooses an unknown and a value for this unknown, for instance 1 for F(14). From that, MALICE deduces as many consequences as it can, then when it is stuck, MONITOR chooses another unknown and so on till MALICE finds a contradiction or a solution. Unfortunately, the space search is so large that MALICE finds no solution within a short time, although it has found many new constraints with fewer unknowns than the initial ones. After a few minutes, MONITOR estimates that there is lack of hope and asks MANAGER for help. MANAGER considers what has been done and notices than only a very small part of the tree has been developed without any results. Then, it decides to take another approach: it generates a combinatorial program, and it compiles and executes it. This combinatorial program is a C program where all of the choices have been made before its generation so no time is spent in modifying the formulation of the problem and it is no longer necessary to ask MONITOR to choose the unknowns for backtracking. Indeed, for some problems, the time saved by cutting the tree does not compensate for the time spent on these activities. However, the system uses the analysis made for the initial situation and includes the constraints that are already found, such as the preceding ones, in the combinatorial program. The writing of this program by MANAGER will be described in section 8.3. Generating and compiling it requires a few seconds, and then we are showered with solutions. Table 2 gives one of them:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
17	6	20	18	26	2	11	5	21	3	25	8	27	1	14	9	16	23	13	22	12	4	15	19	24	10	7

Table 2. A solution for the magic cube.

After generating one hundred solutions (4,800 when the symmetries are included), MONITOR asks MANAGER whether it is worthwhile to continue this program. MANAGER sees that a very small part of the tree has been developed, there are probably millions of solutions, it is not interesting nor practical to generate all of them. It stops the execution of the combinatorial program and stores that, if it has no urgent task, it could be interesting to generate more solutions to have an idea of their number. Later on, it will also ask MONITOR to try and solve this problem in the ALICE way by choosing other unknowns for backtracking; however this experiment will not succeed.

Taking again the combinatorial program, and running it for 13 hours, it has found 2,000,000 solutions, that makes about 60 solutions at each second. All of these solutions have the value of 1 for F(14); however it has only considered one eighth of the tree with this value of F(14). Since there are many solutions for each possible value of F(14), this problem has an enormous number of solutions.

It is interesting to compare CAIA's performance with mine on this problem, both of us are considered as AI scientists. I had found some symmetries, but much fewer than 47. However I had found another kind of symmetry that MALICE is not able to find, where 28-V is substituted for each value V. I had found the value 126 for VAL, but I lost plenty of time trying to prove that F(14)=14. Indeed, in many magic square or cube problems, the value of the central square or cube is the mean; here, this is not the case because the problem is underconstrained and there are solutions for any value between 1 and 27 given to the central cube. Moreover, I had not found the constraints with three unknowns, but only with four unknowns such as F(5)+F(23)=F(11)+F(17); they are not as powerful as the constraints with three unknowns and a lot more time is required to find a solution. On the whole, CAIA's performance on this problem was better than mine; this is significant since it found results that I was not aware of.

We can also compare this approach with the results given by two of the best CSP programs, which are even able to use efficiently the constraints such as "all the unknowns are different". The following experiments were made on a PC with a 1 gigahertz Pentium while CAIA used a 2.4 gigahertz Pentium. ILOG Solver did not find a solution in a reasonable time while ILOG CEPLEX found one after 14 hours. Incidentally, it shows that this program is excellent because the search space is enormous when it is not possible to manipulate the formulation of the problem: it does not know the value of VAL and all of its constraints have 10 unknowns. Naturally, if the human user finds the new constraints found by CAIA and gives them to these programs, they would also discover many results within a short time. However, it would be the user and not the system who would be intelligent.

3. The system MACISTE

MACISTE [35] is not another agent, but is rather like an operating system for CAIA. For its user, CAIA is not a program, but a knowledge based system; its knowledge is given either as sets of conditional actions gathered in what we call MACISTE expertises or in a completely declarative form. For each task, there is an expertise which executes this task. An expertise is a set of conditional actions: if all of the conditions of a conditional action are true, its actions must be executed. When CAIA uses declarative knowledge, it is given in the formalism of MACISTE expressions; thus, it is easy to define MACISTE expertises which transform the declarative knowledge into new MACISTE expertises. Then MACISTE translates these last expertises into efficient C programs; this way the user does not have to write a single line of the programs that finally run on a computer, he only gives knowledge in a more convenient formalism. I will not give a full description of MACISTE, but only indicate some aspects necessary to understand how CAIA works.

MACISTE has a double role. Firstly, it translates all its expertises into C programs, including those of CAIA. Secondly, the user can ask it to start CAIA; then, when it is running, MACISTE may be called by any agent of CAIA in order to perform several general tasks such as managing the short and long term storage, printing messages to the user, writing C programs, interpreting expertises during the debugging of the system, etc.

Basically, all MACISTE knowledge is expressed as objects that are sets of couples: attribute and value of the attribute. The value of an attribute may be an integer, a character, a string, another object or a set (or a bag) of the preceding elements. The expertises, the conditional actions, the expressions are represented as objects, but the user is not obligated to know this basic representation because some functions enable him to work on expressions without actually using their internal representation.

MACISTE is a set of expertises and each expertise allows it to solve a particular task. In an expertise, no order is defined for the conditional actions or their conditions: some MACISTE expertises find how to order the execution of the conditional actions and their conditions. Using them and some other expertises, MACISTE can translate any expertise written as a set of conditional actions into an effective C program. Another expertise diagnoses anomalies in the expertises. All of the operations performed by the system are given as expertises, including the lecture and the printing of its own expertises, the garbage collector, the management of the storage on the disk, etc. The CAIA subsystem is a particularly interesting set of expertises.

A MACISTE expression may be a function with its arguments (which are also expressions), an integer, a character, an object, a variable, the attribute of an expression whose value must be an object, a set such as [X1:X2] (which is the set of integers between I1, value of expression X1, and I2, value of expression X2), etc. I will only describe the most important functions. They may be unary such as MINUS and the absolute value ABS, binary (such as $+$, $=$, $\neq$, $\in$, MOD for the modulo), and variary (in that case the argument is a bag of expressions) such as SUM, PRD (product), OR, AND. Some functions enable the system to analyze its expressions or to create new expressions; since CAIA has to examine its own knowledge and generate new knowledge, these functions are essential for defining its expertises.

MATCH(X,P) is true if the value of the expression X matches with the pattern P. This function is very useful because it enables the user to analyze an expression without knowing how it is represented internally: the language used to define a pattern is similar to the usual algebraic representation of expressions. The function MATCH can also be used to define variables when they appear in P: their value is the subexpression in X corresponding to their place in the pattern. For instance, in MATCH(X,SUM(*Z,*Z,5,**)), SUM is a variary function with a bag of arguments; naturally the order of the arguments in a bag is not used in the matching process. *Z indicates that the variable Z must be instantiated (if it appears several times, it must be instantiated to the same expression) while ** represents any number of arguments. The preceding condition is true if the value of X is the expression SUM(5,T,Y+1,T) and then the value of Z is "T"; it is also true if X is SUM(Y,3*Y+2,1,2+Y*3,Y,5) and the value of Z will be instantiated twice, by the expression "3*Y+2" (or its equivalent form 2+Y*3) and by the expression "Y": in that case, the rule including MATCH will be executed twice, once for each possible value of Z. It is false if X is SUM(T,T,4,Y) or SUM(T,5,Y,T+1).

While MATCH is used to analyze expressions, other functions are used to create new expressions. For instance SIGMA is both a unary and variary function that is used for defining SUM expressions. Its value is the sum of the bag generated in substituting in the first argument with all of the possible sets of variables defined by the conditions in the variary part. For instance

SIGMA(I*F(I); I∈[1:6], MOD(I,2)=0)

will generate the expression:

SUM(2*F(2), 4*F(4), 6*F(6))

In the same way, other functions, which are both unary and variary, are defined for generating other kinds of expressions, for instance XOR for generating OR expressions, XAND for AND, XPRD for PRD; with these functions it is easy to define the construction of new expressions, the unary argument is the model of the elements of the bag that will be generated by the variary argument.

MACISTE is a reflective system in three ways:

1. Some MACISTE expertises have been defined so that they translate every expertise (including themselves) into an efficient C program. It translates its 8,700 conditional actions into 300,000 lines of C; not a single line of MACISTE or CAIA C programs has been written by a human being. This does not take into account all of the lines of C programs written by CAIA when it tries to solve problems using a combinatorial method; in one of its "lives", it may generate 300,000 more lines in these programs. The only human programs used are the C compiler and the Linux operating system.

2. It can observe its behavior when it is running: it may generate a trace as thorough as necessary. Moreover, it can generate new expertises or new C programs and use them immediately without restarting the system.

3. It can have access to the pile of all the active subprograms and to the values of all their variables. When there is a particular incident, it can find where it happens and have a complete

overview of the situation.

All these possibilities are very useful for CAIA which can write and execute new expertises and programs, know what it has done and why it has done it, and also know in what state it is when a difficulty happens.

MACISTE is a tool essential for running CAIA. Since it is easy to understand information written in this formalism, it is possible to modify easily the expertises written over 10 years ago. Without MACISTE, I certainly would not have been able to realize CAIA. However the conditional actions used in MACISTE are partly procedural: a conditional action must be executed when all its conditions are true. Thus, when it is feasible, I prefer to define part of the knowledge used in CAIA in a more declarative form. Some expertises proceduralizes this declarative knowledge into sets of conditional actions so that MACISTE can translate them into C programs.

4. Defining a family of problems

CAIA receives the definition of a family of problems in a special language. Problems are gathered into a single family if it is possible to give one definition for all of these problems; then each problem of the family is specified by data given by the user. For instance there is one definition for the family MAGSQUARE which defines magic square problems: all the numbers in the small squares must be different and all the sums of the rows, columns or diagonals must be equal. A particular problem of this family is defined by the value of the number of small squares in a line, for instance the magic square 3x3 or the magic square 8x8. The language for defining a family of problems stems from the language of Laurière's ALICE [14,15]. The formulation of a family of problems includes four parts. The first one defines the constants, sets, arrays, matrices that specify a particular problem of this family. The second one defines relations (also called correspondences) between two sets: their departure and their arrival sets. The system must specify these correspondences in order to solve a problem of this family. The third one defines constraint generators generating the algebraic constraints that must be satisfied by the correspondences. The last one states several kinds of constraints that are not easily defined as algebraic constraints. We must distinguish the formulation of a family of problems, which usually includes constraint generators, from the formulation of a particular problem where all the constraints are completely specified.

The definition of a particular family of problems will highlight the most important features of this problem description language. To improve the readability of mathematical formulas, I will give them in the usual algebraic form rather than in MACISTE formalism: instead of SUM(F('A'), PRD(4,F{'M'))), I will write A+4*M.

To explain how a family of problems can be defined, I will take the example of the family AUTOREF, which is the generalization of a problem given in the newspaper *Le Monde*; we will see that it is also a generalization of the problem of finding magic sequences. Each problem is defined by two integers P and N; the goal is to find the values of P+2 elements in an ordered list such that the value of element I, I varying from 0 to P, is the number of elements whose value is I. The value of the last element is a given value N, which must be less than or equal to P. For instance, if P=9 and N=3, we have the problem (9,3); there is only one solution: (5,2,2,1,0,1,0,0,0,0,3). The first element (corresponding to 0) is 5 and there are five occurrences of the value 0. In the same way the second element corresponds to 1 and there are two occurrences of 1; there are also two occurrences of 2, only one occurrence of 3 and 5 and no occurrence of 4, 6, 7, 8 and 9; the value of the last element is equal to N, which is 3. In the formulation for MALICE, it will have to find a correspondence F, which is in this case a function, defined on the integers from 0 to P+1. For each I between 0 and P, the value of F(I) is the number of times the value of F(J) is equal to I, J taking all the values between 0 and P+1. The value of F(P+1) is the integer N, which is the second given value for each of the problems of the family. Because of the reflective nature of these problems, it is difficult to find all the solutions when P is large.

I will begin with the formulation of this family of problems as given to MALICE. I will give some sketchy commentaries; we will see later in a more general way how theses formulations are given to MALICE.

GIVEN CONSTANT P

GIVEN CONSTANT N

MALICE will ask the user the values of the two parameters that define each problem.

GIVEN SET ELEMENTS=[0:SUC(P)]

GIVEN SET VALUES=[0:P]

The set of elements and the set of their possible values are defined. SUC is the function successor.

FIND FUNCTION F ELEMENTS->VALUES

The goal is to find the values of a function F. Its departure and arrival sets have just been defined.

WITH F(SUC(P))=N

The value of the function for the last element is the value of parameter N. When P=3 and N=1, that gives the constraint F(4)=1.

WITH F(I)=XNBVAL(I; F(J); J∈ELEMENTS) FOR I ∈VALUES

I have defined some new convenient functions for MALICE, one of them is NBVAL: when I is an expression and B is a bag of expressions, the value of NBVAL(I;B) is the number of times the value of an element of B is I. For instance, the value of NBVAL(2; F(0), F(1), F(2), F(3)) is the number of times the value of function F is 2. When the bag B in NBVAL has many elements, it is easier to define a formula generating this bag in the same way as we use SIGMA to generate a SUM**;** this is done with the function XNBVAL(I;X;E) where I and X are expressions while E is a set of expressions. This formula defines an expression NBVAL(I,B), where B is generated by all the possible instantiations of expression X with the set of variables that can be defined by the set of expressions in E. If P=2, ELEMENTS is the set (0,1,2,3) and XNBVAL(I; F(J); J∈ELEMENTS) generates the expression NBVAL(1; F(0), F(1), F(2), F(3)) when I=1.

The last line in the preceding definition is a constraint generator and it generates as many constraints as the expressions after FOR can instantiate different sets of values for the variables. When P=2, VALUES is the set (0,1,2) so three constraints will be generated when the variable I takes these values successively. With I=0, the constraint states that F(0) times the value of function F is 0: F(0)=NBVAL(0; F(0), F(1), F(2), F(3)).

This definition of AUTOREF generalizes the problem of finding magic sequences, defined in [39] (prob019: magic squares and sequences). A magic sequence of length n+1 is a sequence of integers A0, A1, A2,..... An such as, for all i between 0 and n, the number of occurrences of integer i is exactly Ai. As the number of occurrences of the highest number n is always 0, finding the magic sequences of length n is an AUTOREF problem with P= n+1 and N=0. Problems n=3 and 5 have no solution, there is one solution for n=4 (2,1,2,0,0) and also one solution for every integer greater than 5, all of them having the same form, for instance (10,2,1,0,0,0,0,0,0,0,1,0,0,0) for n=13.

We will now see in a more general way how to define a family of problems for MALICE; a definition may include four parts, all except the second part are optional.

1. The values of the parameters defining a particular problem of the family are given by a formula beginning with GIVEN. If it is followed by CONSTANT, then by the name of the parameter, the system will ask the user the value of this parameter for each new problem of the family.

Another definition is: GIVEN SET parameter=expression. The value of the parameter will be a set defined by the expression. If this expression is [X1:X2], this set will be the set of integers between I1 (value of the expression X1) and I2 (value of the expression X2). If this expression is [V1,V2,....,Vn], where the Vi are integers or characters, the set will include all these elements. The expression may include some of the parameters appearing in other declarations; this expression will be evaluated and the value of the set it generates will be given to the parameter.

Another possibility is GIVEN ARRAY parameter E1->E2. The system will ask the user the value of the array for all the elements of set E1; their value must belong to set E2. E1 and E2 must be defined by two orders such as GIVEN SET E1=expression1 and GIVEN SET E2=expression2. If the name of the parameter is A, A(I) will be the value of the array for I.

2. A correspondence, which is a relation between two sets, is defined by:

FIND type of correspondence name of the correspondence E1->E2

E1 and E2 are the departure and arrival sets of the correspondence. The type of the correspondence may be FUNCTION, BIJECTION, INJECTION, SURJECTION, etc. In fact, MALICE does not use the name of the correspondence when it solves a problem, it only uses this type for defining the values of the minimum and maximum degrees for the departure set (DMI and DMA) and for the arrival set (AMI and AMA) of the correspondence. The maximum degree of the departure set (DMA) is the maximum number of values for each of its elements and the minimum degree (DMI) is the minimum number of values for each element. The definition is analogous for the arrival set. For a bijection, the values of the degrees are: DMI=DMA=AMI=AMA=1; for a function, DMI=DMA=1 while AMI and AMA are not defined. To simplify the explanations, we can speak later of a bijection or of an injection, but CAIA only uses the values of the degrees of the departure and the arrival sets of the correspondences.

We have seen that the definition of AUTOREF includes only one correspondence:

FIND FUNCTION F SQUARES->VALUES

For each correspondence, MALICE builds the elements of the departure and the arrival sets. It includes a POSSIBLE link between an element D of the departure set toward an element A of the arrival set if it is possible that the value of D is A. Later on, some CERTAIN links between such elements will be generated if it is certain that A is a value of D. Removing a POSSIBLE link and adding a CERTAIN link are essential steps to take toward solving the problem. When all the values of the elements of the departure set using CERTAIN links satisfy all of the constraints, the problem is solved.

3. There may be algebraic constraints; their definition begins with WITH. The correspondences appear in these constraints with their name followed between parentheses by the reference to the departure node, for instance F(4) or G('H'), where the departure nodes correspond to integer 4 or to character 'H'. If there is still not yet a certain link between these departure nodes and an arrival node, F(4) or G('H') are called "unknown". MALICE has to give a value to all of the unknowns that appear in the constraints. When there is a certain link, the unknown is substituted by its value, which is associated to the corresponding node in the arrival set. Naturally, only correspondences with DMA=1 can be found in such constraints.

A constraint generator can define one constraint or a set of constraints. Such a generator is beginning with WITH followed by a MACISTE expression followed possibly by a set of conditions. For instance:

WITH XOR(F(I)≠F(J); I∈[1:N], J∈[SUC(I):N])

If the value of N is 3, this generator will generate the following constraint:

OR(F(1)≠F(2), F(1)≠F(3), F(2)≠F(3))

XOR is a unary and variary function which generates the bag argument of an OR: this bag is created with the expression that is the first argument of XOR, its variables are substituted by all the possible values of the variables that are in the variary part of XOR.

In the preceding example, the generator generates only one constraint. If there is a set of conditions, they can define several values for the variables that are in these conditions:

WITH X FOR C1, C2,..., Cn

For the last constraint generator of AUTOREF, there is only one condition I∈VALUES; the P+1 constraints will be F(I)=XNBVAL(I; F(J); J∈ELEMENTS) where I is successively instantiated by all the elements of the set VALUES. For instance, when the value of P is 2, the constraints are:

F(0) = NBVAL(0; F(0), F(1), F(2), F(3))
F(1) = NBVAL(1; F(0), F(1), F(2), F(3))
F(2) = NBVAL(2; F(0), F(1), F(2), F(3))

since ELEMENTS is the set (0,1,2,3) and VALUES is the set (0,1,2). The conditions Ci can use all the possibilities of the language of MACISTE. The argument of a variary function such as OR and AND is a bag of expressions; its value is a bag of integers for SUM and PRD, a bag of boolean values for OR and AND.

XNBVAL and NBVAL are very convenient for defining family of problems, they were not only defined for AUTOREF, but for other problems as well. For instance, only three constraint generators define all SUDOKU problems; they generate 243 constraints for each problem, each one indicates that exactly one element on a row, column or small square has a particular value. The following constraint is generated by the first generator:

NBVAL(3; F(1), F(2), F(3), F(4), F(5), F(6), F(7), F(8), F(9)) = 1

It indicates that 3 appears exactly once in the first row. A few constraint generators can generate many constraints: the 12 generators of another family generated 16,432 constraints for one of its problems.

4. It is often useful to give some constraints in a more convenient way than algebraic expressions. This is done with constraints beginning with KNOWING. Often, at the beginning, any element of the arrival set may correspond to any element of the departure set. When this is not true, it is possible to indicate which elements can be the images of F(I):

KNOWING GRAPH F(I)=X

where X is a MACISTE expression defining the set of possible values for F(I). For instance, the Euler knight problem is defined by a correspondence between the departure square and the arrival square of a knight. Thus, the values of F(I) are the set of squares where a knight on square I can jump.

For some families of problems, it may happen that there is an incompatibility between some elements of the departure set; for instance, elements F(I) and F(J) cannot have the same value. If it is true for all elements, it is sufficient to indicate that the maximum degree of the arrival set for F (AMA) is equal to 1. But if it is only true for a subset of the arrival set, there is a possibility to define these constraints easily:

KNOWING INCOMPATIBILITY F(I)=X

where X is a MACISTE expression. Its value is the set of elements of the departure set of F which must not have the same value as F(I).

A new family of problems can be defined in many ways, and the choice made by each user can have important consequences on the performance of MALICE. Let us consider the definition of the crypt-additions such as DONALD+GERALD=ROBERT or the simpler AB+BCA=ABC. Naturally, the goal is to find an injection F between the set of letters that are in the addition and the integers from 0 to 9 (so DMI=DMA=AMA=1). A first constraint generator indicates that the first letter of each line is not 0; for the second problem that gives $A \neq 0$ and $B \neq 0$. It is possible to write a second constraint generator which states that the result is the sum of N operands, 2 for both of the preceding problems. This generator generates only one constraint; for the problem AB, this constraint is:

$$10*A+B+100*B+10*C+A = 100*A+10*B+C$$

which MALICE simplifies into:

$$9*C+91*B = 89*A$$

But the user may define the same problem in a different way: he can introduce a second correspondence, a function R whose values are the carries. Then, the second constraint generator generates as many constraints as there are columns in the addition; with problem AB, this gives three constraints:

$$B+A = C+10*R(2)$$

$$A+C+R(2) = B+10*R(1)$$

$$B+R(1) = A$$

The difficulty in solving a problem highly depends on its formulation. For the crypt-additions, the definition with carries is usually much easier to solve than the definition with only one constraint. For instance, with DONALD, MALICE develops a tree with only two leaves when it receives the formulation with the carries: its only choice is E=0 or E=9. When E=0 it finds a contradiction whereas when E=9 it finds the only solution. On the other hand, when it works with the definition with only one constraint, the tree has 11 leaves: 10 choices lead to a contradiction.

From now on, I will not give the definition of families of problems as I have done for AUTOREF because this general formalism is not alway easy to understand for someone who is not accustomed to it. For a new problem, I will often give only the constraints such as they are generated by MALICE from the definition of its family of problems.

It is not always easy to define a new family of problems and particularly to write its constraint generators. However, without them, we would have to write as many definitions as there are problems and give all the constraints successively for each problem instead of giving a general definition. For some families of problems, one single generator can generate more than one thousand constraints.

CAIA has received the definition of more than 100 families of problems which represent more than one thousand problems. I do not count here the meta-problems such as those that can create new problems or find symmetries in the formulation of a problem.

5. Defining a problem solving algorithm

This section describes MALICE, CAIA's problem solving module, which solves problems using procedural knowledge given in a declarative form. When procedural knowledge is given declaratively, it is possible to generate it, improve it, and explain why a result has been (or has not been) found. We begin with the description of this knowledge, then we show how MANAGER proceduralizes it so that it can be used efficiently. In order to solve a problem MALICE uses rules to reduce the size of the search space or to find a contradiction; this leads to a meta-combinatorial search among the rules. Then we will see the differences between this meta-combinatorial search where one considers the rules that can be executed and the combinatorial search where one considers the possible values of the unknowns. When the system has executed all the possible rules and has not found a solution or a contradiction, it then performs a step in a classical combinatorial search, choosing an unknown and considering all its possible values. After each of these instantiations it resumes its meta-combinatorial search in the situation that has just been created.

MALICE corresponds only with MONITOR which gives it a problem to solve and answers its questions. It gives MONITOR its results, asks MONITOR what it has to do when it encounters a difficulty; even when there is no apparent difficulty, it regularly asks MONITOR to see whether it is progressing smoothly toward the solution.

I will only describe the main features of MALICE. For instance I only mention that it uses 120 rewriting rules to simplify and normalize expressions such as X*1:=X; the procedural knowledge that indicates when to use them is also given in a declarative form. These rewriting rules are used for normalizing a new constraint and are completely different from the rules that we are going to describe.

5.1. The rules

MALICE uses 95 rules, which are given by the user. A rule has four parts: the declaration of the variables, the conditions, the actions and the links among the variables.

1. The declaration of the variables. The user must declare the types of the variables that will be instantiated when MALICE decides to use this rule. This is a set of couples: (type-variable name). The type may be CONSTRAINT when the variable must be a constraint of the problem, DEPARTURE if it is a departure node for a particular correspondence, ARRIVAL if it is an arrival node, CORRESPONDENCE if it is one of the correspondences.

2. The set of conditions. A condition is a MACISTE expression, as those which are in its conditional actions. Since it is a set, MALICE evaluates these conditions in the order it prefers.

3. There are six kinds of actions:
CONSTRAINT X, where X is an expression, creates a new constraint which is the value of expression X.
LINK ND NA creates a certain link between node ND from a departure set to node NA from an arrival set.
UNLINK ND NA removes a possible link between ND and NA.
VALUE ATT X V gives the value V to the attribute ATT of object X: VALUE AMI CP 1 indicates that the value of the minimum degree of the arrival set of correspondence CP is now 1.
CONTRADICTION indicates that there is a contradiction.
ELIMINATE X eliminates the constraint that is the value of expression X.

4. Let us begin with explaining why it is necessary to give the links among the variables. There may be several ways to use a rule and for each of them, the value of a different variable is known. Indeed, we will later see that the rules are triggered by events and an event instantiates the value of only one variable: MALICE must have the possibility of finding the values of the other variables which must be known in order to execute the rule. When MALICE only knows the value of one of the variables, the links among variables enable it to find the values of all the other ones.

For instance, MALICE has found a new constraint. It may be interesting to apply a particular rule R which can be executed when a constraint C and a departure node D are known. Naturally, C will be the new constraint, but there are plenty of departure nodes and it is useless to consider most of them. The links among variables enable MALICE to make a selection; they will perhaps indicate that D must be a node that corresponds to an unknown of C. Usually several unknowns are in a constraint, each one will give an instantiation for rule R, each of them with the new constraint for C and one of the unknowns of constraint C for D. Inversely, it may occur that a possible value for a departure node N has been removed; a trigger of rule R will perhaps indicate that it may be useful to use it with node N as the value given to variable D. For instantiating C, a link between the variables of rule R indicates that C must be a constraint containing N among its unknowns. Thus MALICE will execute the rule as many times as there are constraints including unknown N.

Let us give some examples of rules; for each rule I begin with a simple example of its use.

Rule R1 states that if there is at least one link starting from each node of the departure set of a correspondence (DMI=1) and at most one link arriving at each node of its arrival set (AMA=1), then there is exactly one node arriving at each node of its arrival set (AMI=1). MALICE will prove AMI=1. This happens when a crypt-addition (defined as finding an injection between its letters and the ten digits from 0 to 9) includes ten different letters; so, in that case, the correspondence is a bijection. Finding out that the correspondence is a bijection is very useful since pruning the tree is much more effective.

Declaration: CORRESPONDENCE : CP
Condition: DMI(CP)*CARD(SETDEP(CP))=AMA(CP)*CARD(SETARR(CP))
Action: VALUE AMI CP AMA(CP)

R1 means that, if for a correspondence CP, the product of the minimum degree of the departure set by the cardinality (the function CARD) of its departure set (SETDEP) is equal to the product of the maximum degree of its arrival set by the cardinality of its arrival set (SETARR), then the value of the minimum degree of its arrival set is equal to the maximum degree of the same set. The condition of this rule is true when a correspondence is an injection (DMI=DMA=AMA=1) and when the departure and arrival sets have the same number of elements.

Rule R2 finds that if there is at least one link starting from each of the ND nodes of the departure set of a correspondence and at most one link arriving at each of the NA nodes of its arrival set, then there is a contradiction if ND>NA: there are nodes of the departure set that cannot be linked to nodes of the arrival set.

R2 is similar to R1, but in the condition '=' becomes '>' and the action is CONTRADICTION. With R2, MALICE immediately finds that it is impossible to put 1,000,000 pigeons (so DMI=DMA=1) in 999,999 pigeon-holes which can contain at most one pigeon (so AMA=1).

Rule R3 is usually used when DMI=1 for correspondence F (for instance it is a function, an injection, a bijection) and there is only one possible value A for node D: one can state that F(D)=A. In a more general form (DMI may be greater than 1), we have:
Declaration : CORRESPONDENCE : CP, DEPARTURE : D
Links between variables : D∈DEPARTURE(CP), CP=FATHER(D)
Conditions:
CARD(POS(D))=DMI(CP)
A∈POS(D)
Action: LINK D A
Let D be a departure node of correspondence CP. If the number of possible links for D is equal to the minimum degree of this node, then all the possible links are certain since it is the only way to reach this minimum degree. This rule can be applied for each correspondence CP and then D is one of its departure nodes; it can also be applied for any departure node D and then CP is its correspondence, given by the attribute FATHER.

Rule R4 is particularly useful for bijections, where the set of departure nodes has the same cardinality as the set of arrival nodes, so the value of every departure node occurs exactly once in the arrival set. Thus the sum of the values of the departure nodes is equal to the sum of the elements of the arrival set. In a more general form (AMI and AMA are equal, but may be greater than 1), we have:

Declaration: CORRESPONDENCE : CP
Condition: DMI(CP)=1, DMA(CP)=1, AMI(CP)=AMA(CP)

Action:

CONSTRAINT SIGMA(VALUE(D); D∈DEPARTURE(CP)) =
AMI(CP)*SIGMA(VALUE(A); A∈ARRIVAL(CP))

VALUE(D) is the expression representing node D: for the departure node corresponding to the character 'B' of the correspondence F, VALUE(D) is F('B'); for the arrival node A associated with integer 4, VALUE(A) is the expression representing the integer 4. With this rule, MALICE has a global vision of the problem: it is very helpful for finding the common value of the lines or of the planes in a magic square or cube. In a 3x3 magic square, there is a bijection S between the integers from 1 to 9 on themselves, so DMI=DMA=AMI=AMA=1. Thus rule R4 generates the constraint:

S(1)+S(2)+S(3)+S(4)+S(5)+S(6)+S(7)+S(8)+S(9) = 45

Finally, I describe rule R5 in a simplified form without giving its MALICE formulation which is more general and further complicated. The idea is that if we know the number Ai of unknowns that have the value i, their contribution to the sum of the unknowns is i*Ai. Thus the value of the sum of all the unknowns is: Σ i*Ai. Let us assume that, among the constraints of a problem, there are the n constraints:

Ai = NBVAL(i; D1, D2,..., Dj,..., Dp)

with i∈[1: n]. The Ai may be any expression; the Dj are unknowns whose value belongs to the set [0 : n]. Integer n is the number of constraints and also the maximum value that can be taken by a Dj while integer p is the number of unknowns in these constraints. The i^{th} constraint indicates that Ai is the number of unknowns Dj whose value is integer i, so the sum of the Ai unknowns Dj that have the value i is i*Ai; we can add all these products for all the possible values of i in order to obtain the sum of all of the Dj. The value of A0, number of unknowns with value 0, is not necessary since it would be multiplied by 0. Thus rule R5 generates the following constraint:

SIGMA(Dj; j∈[1:p] = SIGMA(i*Ai); i∈[1: n])

We will again consider this rule in section 12.2. For AUTOREF with P=3, three of the constraints satisfy the conditions of R5 with n=3 (three possible positive values), p=5 (five unknowns), Ai=F(i) and Dj=F(j-1); so R5 generates the constraint:

F(0)+F(1)+F(2)+F(3)+F(4) = 1*F(1)+2*F(2)+3*F(3)

and, after simplification, as the value of F(4) is the second parameter N:

F(0)+N = F(2)+2*F(3)

For AUTOREF, the constraint generated with this rule for the large values of P drastically increases the efficiency of MALICE. If P=27, the improvement is spectacular, for instance if the value of N is 5, the rule generates the following constraint after simplification:

F(0)+5 = F(2)+2*F(3)+....+25*F(26)+26*F(27)

Each unknown has 28 possible values. As the value of the left part is at most 32, MALICE will easily find with other rules that only two values (0 and 1) are possible for F(16) to F(27), three values for F(11) to F(15), four values for F(9) and F(10), etc. Moreover, when it backtracks with F(I)=1 and I large, it immediately finds that F(J)=0 for the other large values of J.

When the unknowns are boolean, the value of n is 1. So if the Di are boolean and NBVAL(1;D1,D2,....,Dp)=X is a constraint, R5 generates the expression: D1+D2+....+Dp=X. We find a well-known result: the sum of boolean unknowns is equal to the number of unknowns with value 1.

The performance of the system is strongly depended on these rules: another new rule may significantly reduce the size of the tree and the time necessary to find the solution. However, the system must learn to use some rules cautiously, so that it does not generate a large amount of uninteresting new constraints which would lead to an increase in computing time. MALICE uses the same set of rules for all of the problems that it solves. In some cases, the consequences are surprising and fortunate: it applies a rule on a problem where human solvers do not think of using it, but which is appropriate.

Each rule is associated with one or several triggers which indicate when it is interesting to apply it. These triggers are gathered into sets linked to events. I will now describe the events and their associated sets of triggers.

5.2. The events

CAIA knows ten different kinds of events; a set of triggers is associated with each of them. When an event occurs, MALICE executes the triggers of its associated set. The events are:

1. LINK: The value of a node of the departure set is certainly a particular node of the arrival set.
2. UNLINK: A node of the arrival set has been removed as a possible value of a node of the departure set.
3. CONSTRAINT: A new constraint has been generated.

4 to 7. AMI, AMA, DMI, DMA: A value has been found for the minimum (maximum) degree of the arrival (departure) set of a correspondence.

8 to 9. MIN, MAX: The minimum (maximum) possible value for a node of the departure set has been defined or increased (decreased).

10. START: This event is only considered at the beginning of the resolution of a problem: it starts the process.

An event defines only one variable (except the START event which defines none): a constraint with CONSTRAINT, a node with LINK, UNLINK, MIN, MAX, a correspondence with DMI, DMA, AMI and AMA. This explains why it is necessary to include the links among the variables in the rules: MALICE must be able to find the values of the other variables that appear in the rules triggered by this event.

Finding a solution or a contradiction is not an event. In that case, MALICE does not have to consider applying new rules because it only stops the development of a branch of the tree.

5.3. The sets of rule triggers

A set of rule triggers is linked to each event; when this event occurs, MALICE considers all the triggers of this set. Knowledge for solving a problem, called a "method", is made up of the sets for all the events, so it is a cluster of sets of rule triggers. A method is a proceduralization of the set of rules, it indicates whether it is useful to consider some rules after a particular event, and if it agrees, it defines when it will be executed. Different methods may result in very different performance, the quality of the particular method MALICE uses is essential for its performance. If its triggers are inadequate, it wastes a lot of time executing rules unsuccessfully, delays in considering the useful ones, and perhaps does not try them at all. I defined such a method and MANAGER generated several methods. In some cases, MANAGER requires that MALICE uses a particular method because the methods do not have the same performance: some are faster while others take more time to find elegant solutions.

The event START is special because it is a sequence instead of a set of rule triggers. Indeed, it is important to choose the order of the rules considered at the beginning of a resolution. For instance, if one million pigeons have to be places in 999,999 pigeon-holes, it is better to consider rule R2 before building the departure and arrival sets of this correspondence: building such sets including one million elements would waste a lot of time and saturate the memory needlessly if R2 can be applied.

There are two kinds of triggers:

1. The first kind of triggers contains only the name of one rule and the name of the variable of the rule whose value is the object tied to the event. This last piece of information is necessary, for instance if the values of two variables of a rule are constraints, MALICE must know which of them will be instantiated by the new constraint which is the event. For such triggers, it immediately executes this rule.

2. The second kind of triggers indicates that MALICE will only consider executing the rule. The task is defined as in the preceding case, but the trigger now contains a priority, which is a MACISTE expression; its value may depend on the characteristics of the object tied to the event. The rule and the value of its associated variable are stored in a waiting list; MALICE only tries it when no other task has a higher priority. It is convenient to have an expression for defining a priority; for instance, if the event is a new constraint, it is usually better to have a higher priority when few unknowns are in this constraint.

In both cases, there may be a set of conditions. The task is executed only if all of the conditions are true for the object tied to the event. The goal is to avoid wasting computer time considering rules that are certainly useless for the object tied to the trigger. For instance, among the triggers of the event CONSTRAINT, the condition may be that this new constraint is linear or that it contains exactly two unknowns; in that case, the rule will be ignored for the constraints that are not linear or contain more or less than two unknowns.

We will see that MANAGER generates such methods automatically. In section 11.2, we compare the performance of MALICE using these methods and its performance with the method that I have defined.

5.4. Backtracking

To solve a problem, MALICE propagates the consequences of the events with the rules: elimination of a possible value for an unknown, finding the value of an unknown, generating a new constraint, etc. An event triggers the execution of some rules which create new events that trigger new rules and this process carries on as long it can. However, it may occur that not a single rule can be executed; in that case, it can successively consider all of the possible values of an unknown. Thus MALICE creates a LINK event artificially and generates all of the consequences of this choice; if it is stuck again, then it chooses another unknown and considers what happens for all its possible values. In that way it generates a tree, but it tries to keep it as small as possible, using the rules as long as it can last; the system backtracks only when it cannot do anything else.

Choosing the unknown for the backtrack is difficult but very important, so MALICE asks MONITOR to perform this choice. The worst case that happened for a particular problem: the tree had 3446 leaves with the choice made by MONITOR and when MANAGER decided on some experiments to improve this result, it found another choice for the unknown that led to a tree with only 2 leaves! Fortunately, the choices made by MONITOR are rarely so poor; the experiments made by MANAGER show that they are often not the best choices, but the smallest tree is not very much smaller than the one generated using MONITOR's choice.

5.5. Combinatorial and meta-combinatorial search

When MALICE backtracks using the help of MONITOR, it leads to a combinatorial search: they choose an unknown and consider all its possible values. In this way they develop a tree; its number of leaves is equal to the sum of the number of solutions and of the number of contradictions. To justify the solutions and to show that no other solution exists, the whole of the generated tree must be given, even if it has millions of leaves: when a choice has been made in a combinatorial program, it must be given in the explanation of the solution, even when it does not lead to any results, for instance giving the value 4 to unknown F(6) leads to a contradiction. Such a program must print all of the tree so that the user can see that there are no other solutions than those already found.

We will see in section 8.3 that sometimes the system uses only the combinatorial approach. In that case it generates a purely combinatorial program: the system no longer executes rules after choosing the value of an unknown, it only checks whether all the constraints that can be evaluated are true. If a constraint is false, it returns to the preceding choice and considers another value for this unknown, but if it is true it considers one possible value for another unknown. This purely combinatorial program does not uses MONITOR nor MALICE; it is generated by MANAGER; in that case there is no meta-combinatorial search.

However, MALICE first performs a meta-combinatorial search: before considering all of the possible values of the unknowns, it executes all the rules suggested by the method that it is using. The combinatorial and the meta-combinatorial search exist simultaneously when MALICE uses what I call "the intelligent approach"; the goal of the meta-combinatorial search is to execute a lot of rules expecting some of them to succeed and reduce the size of the search space. When all the possible rules have been executed, MALICE leaves the meta-combinatorial search, MONITOR chooses an unknown, thus creating a new step in the combinatorial search; then MALICE resumes the meta-combinatorial search in the situations generated after each of these choices. In the resolution of a problem, there are long periods of meta-combinatorial search which end with a solution, a contradiction or another step in the combinatorial search.

A meta-combinatorial search can waste an enormous amount of time: the time required to find a solution may be greater than what is necessary for an efficient combinatorial program. However, the meta-combinatorial search is interesting even when it is slower because it finds more elegant solutions, which can easily be checked and explained to human beings. The justification of the solution of a problem does not include the application of all the rules that were executed: MANAGER removes those which do not reduce the size of the search space either directly or by the results that were possibly deduced from them. This occurs when MALICE already knows that F(4) is different from 3 and it generates the new constraint $F(4)\neq 3$. This also happens when a new constraint does not eliminate possible values for unknowns and all of the constraints that were generated from it are also useless: it is unnecessary to indicate the generation of this constraint in the explanation of the solution. The system is in the same situation as the mathematician who worked several years trying to prove a new theorem, wrote thousands of pages and finally published a proof which is less than one page. To justify a result, it is unnecessary to describe all of the unsuccessful attempts. If most of the rules are not useful, that means that the meta-combinatorial search is conducted poorly, but this does not lead to solutions which are difficult to explain, contrary to a poorly conducted combinatorial search.

When MANAGER explains a solution obtained by MALICE using a meta-combinatorial search, it eliminates all of the unsuccessful attempts to present elegant solutions. Moreover we will see in section 8.6 that it can also meta-explain, that is to explain why MALICE has made successful as well as unsuccessful attempts. This information could be used to learn to reduce the meta-combinatorial search in order to find the same solutions faster; however this is a difficult task for human and artificial scientists.

6. Choosing what will be done

When the system must choose among several possibilities, it can often reason, that means to execute a sequence of steps that will lead to the final choice. In that case, the necessary knowledge is given as a set of conditional actions such as the execution of an action is a step toward the goal; this succession of steps explains why a certain choice has been made. However it also occurs that the system must choose between two possible decisions and many characteristics of the situation must be taken into account in order to find the most satisfactory choice; for instance, this happens with the choice of the unknown for backtracking. In that case, it is difficult to define a good choice by a sequence of conditional actions because there are too many characteristics; therefore, one must find a good compromise between the advantages and the drawbacks of each possible choice. Here we describe the method used by MALICE, MONITOR, MANAGER, and ADVISOR to make such choices. For each kind of choice, the knowledge for choosing among several possibilities is given with a special kind of conditional actions; thus, the user can easily give and modify this knowledge. These conditional actions do not define a step toward the goal, instead they defined many micro-decisions such as:

The importance of a particular characteristic of the choice is high, but its value is low.

. Then a general algorithm makes the synthesis of all these micro-decisions and computes a degree of interest for each possible choice. In some applications, only the choice with the highest interest is chosen; in other ones, the interest is used to sequence the possible choices in order to define which one will be considered first. As the algorithm is general, it can easily be used in any context where a choice must be taken; the user only has to give the conditional actions which must be used in this context. This algorithm stems from the one defined by S. Pinson [30] in a different domain.

Nine types of choices are made using the knowledge given in this formalism.

With MALICE:

1. Deciding whether it will keep a new constraint. It is not always good to store too many constraints which can take up a part of the memory and may slow down the system with too many unsuccessful attempts. When this decision is positive, it also gives the value of the interest of this constraint, which will condition its future use.

2. Choosing the next attempt, for instance applying such a rule to such constraint. Each possible attempt receives an interest; the system begins with those with the highest interest until it finds a solution or a contradiction.

With MONITOR:

3. Choosing the unknown for the backtrack in the intelligent approach. This is done dynamically during the resolution; the sequence of choices may be different in different paths of the tree.

With MANAGER:

4. Choosing the unknown for the backtrack when MANAGER writes a combinatorial program; this program is a sequence of loops and it has to find the unknown whose values will be successively considered in the next loop. Here, the choice is made only once when it writes the program: the sequence of choices is the same in all the paths of the tree.

5. Choosing the following constraint which will be inserted in the combinatorial program generated for solving one specific problem.

6. Choosing the family of problems for which it will find new problems. MANAGER takes into account the number and the quality of the problems that are already found in this family.

7. Deciding to keep or discard a new problem that it has just created. This can also enable it to eliminate this problem later if it creates better problems.

8. Choosing the following task which it will have to execute. In that way, it sequences what it will do during its career.

With ADVISOR:

9. Assessing the progress of the system since its last assessment. Regularly, it considers if MANAGER has found interesting results in the last period of time. If there is little progress, it must either be stopped or try other directions of research.

For each of these choices a set of conditional actions is defined. Their actions are different from those used elsewhere in CAIA, which states new facts that are used by other conditional actions. Here, the execution of an action only defines a micro-decision, and the set of all these micro-decisions is given as data to a special algorithm: it considers all of them in order to define an interest for this candidate. These conditional actions have the following form:

set of conditions => a value and its importance for the candidate

The conditions are given in MACISTE's formalism; the value and the importance of a candidate are also MACISTE expressions. The actions all have the same kind of consequence: they add a micro-decision which is a couple (value-importance) to the set associated to a particular possible choice. The value of a micro-decision may be favorable or unfavorable with seven degrees in each case: VERYLOW(VL), LOW(L), RATHERLOW(RL), MEDIUM(M), RATHERHIGH(RH), HIGH(H), VERYHIGH(VH). Its importance can only be positive and has the same seven possible values; the higher the importance, the higher will be the consequences of its associated value for the final choice. This is not an "interest", which means that something attracts the attention; instead, the importance precisely indicates that it has a great influence on the result. To show why it is useful to have simultaneously the value and the importance of some characteristics of a candidate, let us consider an example given by S. Pinson where the goal is to find whether a bank will allocate a loan to a small firm. There are expertises for all the aspects of a firm, such as its commercial activities and its financial data. The quality of its managers is also essential, and one of its components is the quality of the chairman's successor. Its value can go from very high unfavorable to very high favorable. However, this value is very important if the chairman is 95 years old but less important if he is 40 years old and in good health.

When MONITOR determines if unknown U will be chosen for backtracking, a characteristic of the situation may be:

Unknown U appears in an equality constraint C including ten other unknowns

and this activates a conditional action which adds the following micro-decision:

The importance of the occurrence of U in C is RATHERHIGH and its value is LOW.

This conditional action considers that it is rather important that the chosen unknown appears in many constraints, that it is better if this constraint is an equality constraint, but that the value of this micro-decision is low when there are many other unknowns in this constraint. There can be many micro-decisions only for this conditional action because there may be hundreds of constraints and this unknown may appear in dozens of them. Moreover other conditional actions will generate many other micro-decisions.

All of these micro-decisions are gathered into a set of triples for each candidate: value-importance-number, the number indicating the number of times that a micro-decision has both this value and this importance. The algorithm associates an interest to every set of such triples and the choice is made using the interest defined by this way for every candidate. This is different from usual MACISTE expertises because the set of conditional actions do not lead to the result by a succession of steps, but only generates the set of micro-decisions which is used as a whole by the algorithm giving the result; in this way it is possible to take many factors into account simultaneously.

Let us assume that a system has to choose among several candidates for one of the nine kinds of choices. It takes all of the conditional actions tied to this choice and applies them to all of the candidates. For each candidate, this gives a set of triples: value-importance-number. It applies the general algorithm to these triples, the same for the nine types of choices that we have just mentioned. This algorithm adds the interest of each triple; the interest of a triple is the product of its number by a function of its value (negative if it is unfavorable) and by another function of its importance. This gives a number which is the interest of the candidate.

Let us consider the three following candidates with their sets of triples:

(VH-VH-2),(H-H-1),(H-RH-3),(RH-VH-3),(RH-RH-2),(RH-RL-2),(M-H-1),(RL-RH-1),(L-RH-1)

(VH-VH-1),(H-H-1),(M-RH-2),(M-RL-1),(RL-RH-3),(L-VH-1),(L-RH-1)
(VH-VH-1),(H-H-1),(L-VH-1),(L-RH-1),(VL-RL-1)

(H-RH-3) means that the value HIGH is associated three times with the importance RATHERHIGH. The interests for these candidates are respectively 644,864, 310,912 et 303,632; thus the first candidate will be preferred. In the present case, MONITOR had to choose an unknown for backtracking, but the result would have been the same if these sets of triples had been generated for any kind of choice.

It is no longer possible to explain such choices. However, it is possible to know all of the micro-decisions that were made for a particular choice, so when one choice seems inappropriate, a scientist has all the necessary information to see why this choice was selected.

7. Monitoring the search for a solution

A problem solver may waste a lot of time in dead ends. To avoid this difficulty, it is better to examine what occurs while it is solving a problem. This is the task of MONITOR which supervises the progress of MALICE toward the solution. Due to MONITOR, MALICE does not get lost in a sequence of derivations which does not lead to its goal or saturates the memory. When an incident occurs, either MONITOR solves it or provides MANAGER with enough information to make a correct decision. Finally, MONITOR stores the information necessary to explain and to learn. MALICE never asks any questions directly from MANAGER, MONITOR is a required intermediary.

MONITOR receives questions and results from MALICE while it gives MALICE the tasks which it has to perform. It answers its questions and changes the behavior of MALICE when it does not progress satisfactorily. It receives from MANAGER the tasks which must be performed, gives it the results of these tasks, and asks MANAGER to solve the difficulties that it is unable to solve.

7.1. Examining the progress toward the solution

MALICE constantly repeats the same cycle, applying a rule to an object. Every N cycle, MALICE asks MONITOR to consider the progress toward the solution. Among other things, it checks that:

1. MALICE has not exceeded the allocated time. When this occurs, it asks MANAGER if it can spend more time on this problem. MANAGER will decide according to the importance of the problem, its progress, and the importance of other awaiting tasks.

2. Enough memory is available. If not, it gets rid of many elements which are not necessary to find the solution; however, that cuts down the information the system will have on its behavior at the end. For instance, it can decide to erase the part of the tree that has already been developed; in that case, a full justification of the solutions can no longer be generated (but it is possible to store it in a file).

3. MALICE does not generate too many constraints. Often an explosive generation of constraints occurs because it does not control this mechanism appropriately: it generates new constraints from the latest ones unceasingly and this may take a long time without any progress toward the solution. MONITOR can find this happening; in that case, it eliminates some constraints that are already generated and restricts (or even forbids) the use of some prolific rules.

4. In some experiments, MALICE solves a problem that has been already solved using different backtracking choices. MONITOR regularly compares the progression of the development of the tree for these two executions. If the new execution does not develop the tree any faster than the first one, it stops this experiment.

Finally, MONITOR may change the value of N, the number of cycles after which it is consulted. It can increase it if everything goes well or decrease it if the situation is difficult to evaluate: in that case it is better to keep a close eye on MALICE.

7.2. An important step has been made

An important step is a significant advancement toward the complete resolution of the problem, such as the discovery of a contradiction or a solution; naturally, the end of the resolution is also an important step. MONITOR considers whether it would be interesting to keep the circumstances of these events.

The first thing is to see if there are not too many important steps. For instance, some problems such as the magic cube seen in the second section have millions of solutions; usually, it is not interesting to generate all of them. If the limit given by MANAGER is exceeded, MONITOR will ask MANAGER whether it can continue further. If the limit is to find 100 solutions and one third of the tree has been developed, usually MANAGER will allow it to continue so that it can get the complete number of solutions. If MALICE has developed less than one thousandth of the tree, MANAGER certainly will order it to stop.

Another important step occurs when there is nothing more to do when solving a constraint satisfaction problem: MALICE has made all of the attempts defined by all of the events that have occurred; the only possibility is to backtrack. MONITOR will choose an unknown according to the constraints, the number of possible values for each unknown, the characteristics of the correspondences, etc.

MONITOR also considers the tree while MALICE develops it. For instance, it may occur that after all the values have been given to an unknown during a backtrack, another unknown has always the same value: the value of F(7) is always 3 when the value of F(2) is 1, 2, 5, 7 and 9. This fact is given to MANAGER and perhaps later, it will ask to make an experiment in order to find if it is possible to prove that F(7)=3 before backtracking on the possible values of F(2). If it succeeds, it means that MALICE must apply more rules before backtracking; thus, MONITOR has found that there is a possibility to learn by improving the use of the rules. CAIA is not able to learn from such situations, but it can give the information to a human scientist. It may also occur that a contradiction is found shortly after giving a value to an unknown; for instance, MONITOR decides to backtrack with unknown F(4) and MALICE finds a contradiction soon after giving value 7 to F(4). Is it possible to find F(4)≠7 before backtracking?

The end of the resolution (with or without success) is a particularly important event where MONITOR assesses MALICE's performance.

7.3. Signals from the Operating System

The Operating System (here Linux), can find some anomalies and report them via a signal. For instance SIGILL indicates that the system has tried to execute an illegal instruction while SIGSEGV indicates a segment violation, probably due to an illegal pointer reference or an array bound error. Systematically, MONITOR catches all of these signals. As MACISTE is a reflective system, it finds the subroutine where this occurred, the subroutine that called the preceding one, etc. It also knows the value of all the variables of these subroutines. MONITOR keeps some of this information so MANAGER can analyze it later on, then it stops the resolution of the current problem and returns the control to MANAGER, which will find another task to perform.

7.4. Various traces

To explain, learn and choose experiments, MANAGER must know what happened when it solved a problem. For this, MONITOR generates two traces.

The first trace is easy to build and many AI systems such as SOAR [12] already use it . It contains all of the attempts that have been made: what rules have been executed, which were their arguments, what unknowns have been chosen for backtracking, and what partial results were found (new constraints, removal of a possible value for an unknown, etc.). With this trace, it is easy to justify a solution: MANAGER will only keep the steps that were necessary for the solution. To explain, it does not consider the attempts that have failed, those which gave an already known result, and those that were never used for inferring an useful result. To create an explanation, it starts from the contradictions, the solutions, the decisions to remove a possible value of an unknown or to give a value to an unknown. It goes back to the attempts that lead to the preceding results, then back to the attempts that lead to the preceding attempts until it reaches the initial data of the problem. In that way, MANAGER or the human user has a rigorous justification of the solution, but it also knows when a rule is useful or useless; this will be used to build better methods.

The second trace is a meta-trace, which contains why an attempt has been made. The events that trigger a rule are associated with this rule. It also includes the decisions to eliminate rules because their interest is too low. The trace keeps track of which attempts have been made, the meta-trace stores why a certain attempt has been made. With the meta-trace, MANAGER can find out why MALICE has decided to make useful attempts, and also why a result has not been found; in this last case, it compares the meta-trace of a resolution finding the desired result with the meta-trace of the resolution failing to discover it. MANAGER only has to find the first useful result of the first meta-trace that is not in the second one: the answer is the reason that enables MALICE to try the rule that delivered this result.

Monitoring the resolution of a problem is essential for two main reasons: to avoid wasting time and to gather information that will be helpful for learning, explaining and finding interesting experiments.

8. MANAGER

MANAGER is an autonomous system that must never stop: it takes all its decisions by itself and it has been working for several weeks without any human interference. It has to supervise MONITOR when it is monitoring MALICE, but it also has to decide at each step whether it will solve awaiting tasks, deepen its understanding of the difficulties found when solving some problems, conduct experiments so that it can find interesting events, generate new problems, find symmetries in the formulation of a problem, compare the results obtained with different methods, meta-explain why a solution is more elegant with one method in comparison to another one, write a combinatorial program to solve a particular problem, understand why there was an anomaly during the execution of a task, etc.

MANAGER gives orders to MONITOR and receives the results and also the difficulties that MONITOR is unable to deal with. It regularly gives the control to ADVISER which can analyze whether it is progressing in its life as a scientist. When I say that MANAGER asks MALICE to do something, it is always via the intermediary of MONITOR.

I call a "life" of the system the sequence of steps MANAGER performs, starting from scratch, building methods, solving problems, making experiments, finding new problems, meta-explaining differences between solutions, etc. It had several lives and it starts from the beginning with each of them, the only knowledge that is transferred from one life to the following one is: new problems generated in the preceding lives, and the fact that some problems are difficult to solve. It would be possible to transfer more information, but this has not been implemented because it would be impossible to find if an improvement is due to a modification of the system or to the knowledge transfered from a preceding life. Moreover, as many changes are done between two lives, old information may be more harmful than useful in the new state of the system.

At the beginning of each new life, MANAGER must bootstrap: it generates an initial method and with this method, MALICE solves all of the problems of the learning set. When this is completed, MANAGER generates a new learned method and it enters into a loop that may last several millions of seconds.

8.1. Supervising MONITOR

MONITOR has to monitor MALICE when it solves a problem, but within the limits that were given by MANAGER, for instance it must stop when it has found 50 solutions or after 100 seconds. When this number of solutions has been found or when the time limit has been reached, MONITOR asks MANAGER whether it must carry on. Then MANAGER considers many factors such as the development of the tree, the importance of the problem, the importance of other awaiting tasks and makes its decision. If it agrees to allocate more time or to find more solutions, MONITOR will tell MALICE to carry on its resolution. But MANAGER can decide to stop this execution completely or to wait and resume it later when there will be less important tasks to perform. This decision is made as defined in section 6: several conditional actions are associated to this decision and each one defines a value and an importance for one of the possible aspects, for instance: is it an experiment, how much time has already been spent, what percentage of the tree has been developed, etc.

MANAGER will also decide when MALICE will have to solve the meta-problem to find symmetries in the formulation of a problem P. In that case, it creates the data for the meta-problem SYMMETRIES and asks MALICE to solve it. When this is completed, it stores the results and add new constraints to problem P so that it will generate only one solution for each set of symmetrical solutions. MALICE will also be able to use the symmetries for generating all the symmetrical constraints of a new constraint. In the same way, it may ask MALICE to solve another kind of meta-problem, generating a new problem in a particular problem family; for each new problem, it asks MALICE to solve it and then decides whether it will keep this problem according to its results: number of solutions, number of contradictions, time necessary to solve it, difficulty, etc.

8.2. Learning new methods

The performance of MALICE strongly depends on the quality of the method (that is the cluster of sets of rule triggers) it uses. MANAGER automatically allocates the problems in a learning set and in a testing set; some of these problems have been given by the user and others found by CAIA in one of its preceding lives. In the first step, MANAGER asks MALICE to solve all of the problems of the learning set with the initial method. With the information from this first trial, it improves this initial method into a much more efficient one.

Let us begin with the construction of the initial method. No priority is associated to its triggers, the associated rule will be executed as soon as the event occurs if its conditions are true. To decide whether a rule will be associated with an event, MANAGER considers for each rule what it is using in its conditions and what it creates in its actions. For instance, one of the conditions of rule R3 tests whether the cardinality of the set of possible values for a node of the departure set is equal to a known value. It can be useful to consider this rule for each event UNLINK that removes a possible value to a node of a departure set because that changes the cardinality of its set of possible values; thus, this rule is appended to the set attached to the event UNLINK. The conditions issued from a function MATCH (defined in section 3) are specially interesting because they usually give some necessary characteristics of the constraint, the node or the correspondence which is the value of the variable that is the first argument of this MATCH. In that case, the conditional actions of MANAGER's expertises contain functions MATCH with a function MATCH in their pattern argument. MANAGER is working at the meta-level, examining rules: it is essential that it can examine the knowledge used to solve a problem.

Building the list associated with the event START is more difficult since the triggers are ordered. For that matter, MANAGER considers the actions and the conditions of the rules: it includes the trigger of a rule R, which uses a particular object, after the triggers of another rule that may create this object. For instance, a rule with a constraint in one of its arguments is placed after the rules that generate the initial constraints of the problem.

Surprisingly enough, with this initial method MALICE performs well: it usually finds more elegant solutions (that means with fewer backtracks) than those generated using the method I have created, but it requires more time. However, since no priority is associated with each trigger, it does not sequence their usage and makes many unnecessary attempts. In some cases, it generates too many constraints, MONITOR must eliminate some of them in order to avoid a combinatorial explosion; but important results may be eliminated in that way and MALICE does not solve few of the problems that were solved with my own method.

In order to learn, MANAGER asks MALICE to solve all of the problems of the learning set with this initial method. When a problem is solved, MANAGER uses the explanation generated from the trace stored with each rule in order to count the number of times this rule was useful for justifying the solution and the number of times it was not useful. Thus, for each problem, it knows the number of useful and useless executions of this rule.

Using this information, MANAGER generates another method: the learned method. It can decide not to link a rule to an event because that was rarely useful. It can also give a low priority to a trigger because using this rule was seldom useful: the priority of each trigger is computed with the results obtained from the learning set of problems. On the contrary, it may decide that some rules will be executed as soon as they are triggered by a particular event because it was almost always useful to execute them after this event.

MANAGER can compare the performance of the three methods: the given method which I created, the initial method and the learned method which are created by MANAGER. Naturally, it only uses the testing set of problems, which were not used for generating the learned method. With this new method, these problems are solved much faster than with the initial method, and almost as fast as with my method. Curiously enough, the solutions are slightly more elegant than with the initial method, although some rules with very low chances of success have been eliminated.

Usually, MANAGER uses the learned method since it gives positive results. However, in some cases, it requires MALICE to use the initial method, for instance when it wants to make a thorough search in a situation where it seems possible to improve the results found with the learned method. If it succeeds, MANAGER compares both meta-traces and meta-explains why MALICE has not found a better solution with the learned method; we will later describe how this is done.

8.3. Writing a combinatorial program

For some problems, an elegant solution does not exist and even with the intelligent approach, MALICE cannot develop trees with fewer than several thousands of leaves. No human being can understand such a proof which also requires a whole lot of computer time. Thus, for such problems, it is better that MANAGER writes an efficient specialized combinatorial program. Rather than slowly developing a large tree with the intelligent approach, this specialized program develops a much larger tree. However, it develops it quickly because the decisions are made only once when it generates the program and no longer when it is developing the tree. In that case, the meta-combinatorial search was not very successful so the system no longer uses it, except for once in the initial situation. Here we have the advantage of compilation over interpretation. Several systems [17,22] have already been able to write such programs.

MANAGER can start from the initial formulation of the problem, but this may not lead to an efficient program if the constraints have many unknowns. With the problem of the magic cube seen in section 2, we have seen that it is very interesting to find new constraints with few unknowns. Thus, when MANAGER wants to write a combinatorial program to solve a problem, it first asks MALICE to reduce the size of the search space and to generate new constraints. It is only when MALICE can no longer advance without backtracking that MANAGER generates a program, using the constraints found by MALICE and taking into account the reduction of the search space. In that case, an intelligent study of the situation, using a meta-combinatorial search among the rules, is made only once in the initial situation and not in every choice of an unknown. MANAGER must be meta-intelligent enough to see when it is not useful to continue the intelligent approach and that the best solution is to write an efficient combinatorial program.

MANAGER uses a lot of knowledge to generate a combinatorial program, specially for choosing the order of the unknowns whose possible values will be considered first. Usually MANAGER chooses well the order of the unknowns and the combinatorial program generates a rather small tree: 176 leaves for the DONALD crypt-addition. 176 is high compared to the 2 leaves generated with the intelligent approach, but small when one considers the size of the search space, almost 10^{10}. All of this is done in a very short time: generating, compiling, loading and executing such a program requires less than one second for most of the problems, as long as the tree has fewer than 10,000,000 leaves. During one of its lives, MANAGER can generate 400,000 lines of C for hundreds of such programs, each of them solving a specific problem. A different program is generated for each problem of the same family of problems.

To explain how such a combinatorial program is generated, let us consider the magic cube of section 2. MALICE has proved several useful constraints, reduced the size of the search space, but was unable to find a solution in a reasonable time. MANAGER starts from the situation such as when MALICE was stuck and MONITOR asked it to backtrack. MANAGER first chooses one unknown, preferring those which appear in many constraints, appear in equality constraints, appear in constraints with few unknowns, have few possible values, etc. Once an unknown U has been chosen, it creates a C statement which is a "for" loop considering all the possible values of unknown U. Then it checks the bijection constraint: the value given to U has not already been given to another unknown. For each constraint where the value of all the unknowns has been instantiated, it adds a C order to check that this constraint is true. Naturally, when a constraint is false, the program returns to the last preceding loop. It may occur that the values of all the unknowns of an equality constraint except for one unknown V have been instantiated. In that case, it creates an instruction computing the value of V and resumes the preceding process with unknown V instead of U; however, there is no loop added to the program. When MANAGER has created all the C instructions that could be generated after instantiating unknown U, it chooses another unknown, adds a new loop to the program and resumes the process. When all of the unknowns has been instantiated and all of the constraints are satisfied, it has found a solution. The program generated for this magic cube has only 17 loops while there are 27 unknowns; finding the value of SUM and three equality constraints with three unknowns is essential for reducing the size of the tree; choosing the order of the unknowns is also very important and this is described in section 6.

In that way, MANAGER is able to generate efficient programs for many problems. For instance, the goal of the problem G(N,P) from the GOLOMB family is to find an ascending sequence of N numbers beginning with 0 and ending with P such that the N(N-1)/2 differences are all different. In order to discard a symmetry, the first difference must be less than the final one: F(2)-F(1)<F(N)-F(N-1). For G(4,6), there is only one solution: (0,1,4,6) and we can check that the six differences are (1,2,3,4,5,6). Such problems are very difficult when N is large. MANAGER proved in 2,992 seconds that there is no solution for G(13,105), and in 7,251 seconds, that there is only one solution for G(13,106): (0,2,5,25,37,43,59,70,85,89,98,99,106). It also proved in 43,337 seconds that there is no solution for G(14,126) while 52,190 seconds were necessary to prove there is only one solution for G(14,127) with its 91 differences:

(0,4,6,20,35,52,59,77,78,86,89,99,122,127)

The smallest number that is not a difference is 56. It is not easy to write a program that finds such results in a reasonable time.

MANAGER is not able to generate a combinatorial program for every problem; however, when it considers the formulation of a family of problems, it knows whether it can perform it. For instance, MANAGER knows that it cannot generate a program when both the maximum and minimum degrees of the departure set of a correspondence are not equal to 1; this happens when the problem is to find a circuit on a board such as the Euler's Knight. The other important impossibility occurs when the argument of a correspondence includes a correspondence in at least one constraint, for instance there is somewhere F(F(3)). This often happens in the family QGROUP, the quasi group existence problems (we will define it in section 11.4). There is no difficulty with the constraints defining a quasi-group, nor with some other additional constraints such as those demanding that the quasi-group is idempotent. But with most additional constraints, MANAGER can no longer generate a combinatorial program, for instance for any a and b, (a*b)*b=a*(a*b) must be true. However, all of these problems can be solved by MALICE using the intelligent approach.

When I am speaking of the intelligent approach in opposition to the combinatorial approach, this does not mean that the system is not intelligent when it uses such methods: it must be meta-intelligent to see that the combinatorial approach is the best one for some problems and be intelligent to write a very efficient combinatorial program. However, the combinatorial program which is generated is not in itself intelligent.

8.4. Choosing experiments

When MONITOR monitors the resolution of a problem, it can inform MANAGER if something is abnormal in order to experiment and understand it. Once MANAGER has chosen to perform an experiment, this decision is stored in a waiting list with a priority that depends on the problem, the time it may require, its type, etc. The following five examples of experiments are collected by MONITOR and decided by MANAGER to perform:

1. There is a contradiction shortly after giving value V to unknown U in a backtrack. It is natural to ask whether it would be possible to find U≠V before backtracking. If MANAGER decides to perform this experiment, it places MALICE in the situation just before backtracking and asks it to prove U≠V using the rules thoroughly. If it succeeds, it is possible in this situation to improve the method used by MALICE.

When using the rules thoroughly, MALICE is using the initial method and it can make a greater number of formal derivations on the constraints than in a normal resolution; particularly, it is less restrictive for the overuse of dangerous rules. For MANAGER, a dangerous rule is a rule which creates at least one new constraint and where at least two of its variables are constraints: this rule, applied to the old constraints and to the new constraint that it has just generated, could generate an infinity of constraints. In a thorough search, MALICE does not use most of the heuristics that limit the use of the rules, and MONITOR no longer has the right to forbid the utilization of a rule. This may require a lot of time, but it can find solutions that could otherwise be found.

2. As we have already seen, MONITOR may notice that an unknown U always has the same value V, whatever value has been assigned in a backtrack to another unknown W. Is it possible to prove that U=V before backtracking with W? As in the preceding case, MALICE is placed in the situation just before this last backtrack and it uses the rules thoroughly. If it succeeds, MANAGER knows another situation where it would be possible to improve MALICE.

3. When it is necessary to make some backtracks and when the CPU time is reasonable, MANAGER may try to solve the same problem again; but is asks MALICE and MONITOR to use the rules thoroughly during all the resolution of the problem and not only for solving a small

subproblem as in both preceding cases. For some problems, it may find a solution which develops a smaller tree and MANAGER will try to understand why MALICE did not find this more elegant solution in its first attempt.

4. When MALICE has solved a problem, it has often made some backtracks; let U be the unknown chosen at the first backtrack. MANAGER may ask MONITOR to try another choice instead of U for this first backtrack and asks MALICE to solve the same problem once again. If it succeeds by coming up with a more elegant solution, that means it is possible to improve the knowledge for choosing the unknown for the backtrack. Thus, MANAGER knows a good choice and a poor one; this information could be used by the user or by a future learning program.

In that case, MANAGER performs a meta-combinatorial search completely different from what we have seen in section 5.5. When it backtracks, it can see the outcome of alternative choices for the unknowns. When it chooses another unknown, MALICE generates another tree. In that way, MANAGER generates a sequence of trees where each of them gives a solution to the problem. Naturally, at the end it chooses the smaller tree. Thus, it may find a more elegant solution: this new tree tree may have 2 leaves instead of 3446 as we have seen in section 5.4. In that case, there is a superposition of two meta-combinatorial searches: the first among the rules, made by MALICE in order to find an elegant solution and the second made by MANAGER in order to choose among the elegant solutions generated after various backtracking choices. Even though this is very costly, it may lead to solutions justified by very small trees.

5. When MALICE has to backtrack in order to solve a problem, it may be interesting to compare this intelligent solution with the one obtained by a combinatorial program.

With type 1 to 4 experiments, a by-product may be to find a more elegant solution than the one that was found in the first attempt to solve a problem.

In the preceding cases, the problem had been solved. But it is more important to make other kinds of experiments when MALICE fails. If that occurs, MANAGER may perform some of the following experiments:

1. Asking to generate a combinatorial program. It often succeeds, but this can be applied only for the problems where MANAGER knows how to generate such a program.

2. Asking MONITOR to choose another unknown for its first backtrack. For some problems, that may result in a radical improvement, mainly for the impossible problems which have no solution and are very responsive to the choice of this unknown.

3. Asking MALICE to inhibit some rules. Indeed, MALICE may fail because some rules can create many new constraints which lead to the execution of many more rules which in turn can create more constraints, etc. MONITOR can already find this occurring and can also inhibit some dangerous rules, but it is often too late. Thus, to avoid this combinatorial explosion, MANAGER asks to inhibit these rules from the start.

MANAGER assesses its experiments. It notices the particularly interesting results and informs them to the user. Unfortunately, it is unable to use most of these results by itself. For instance, if it has tried to backtrack with some unknowns other than the one chosen by MONITOR, it will only write a message with the following information:

MONITOR chose to backtrack first with F(16) and there were 2,061 contradictions. The best choice is to backtrack first with F(11) and there are only 552 contradictions. Four other choices are better: F(2) with 1,494 contradictions, F(4) with 831 contradictions, F(7) with 638 contradictions, and F(10) with 1,153 contradictions. The three choices: F(12), F(1) and F(13) are worse than F(16).

8.5. Understanding anomalies

MONITOR often detects anomalies and informs MANAGER, which then stores them. It usually stops the resolution of the current problem so that it can later understand what happened.

Some anomalies come from a bug that leads to a Linux SIG message or an infinite loop (which is detected by ZEUS as we will see later). MONITOR catches the signal and informs MANAGER about the situation of the problem and the subroutine that was running when the signal or the loop occurred. Usually, MANAGER demands to stop working on this problem, but later it will try to understand what happened. In that case, it asks to solve the problem in exactly the same way in order to be sure that it is not due to a transient problem coming from the operating system. If this error happens again, MONITOR or ZEUS indicates that the difficulty occurs in the translation of a particular expertise. MANAGER tries to find exactly where it comes from: it recompiles this expertise, but after each instruction, it inserts another instruction which changes the value of a new variable: when the anomaly reoccurs, it knows exactly the C instruction that is responsible from the value of this variable. Since it wrote this subroutine, it can relate this instruction to a condition of a particular conditional action whose translation contains this instruction. However, it is unable to correct this condition and the expertise that includes it, but it gives a lot of useful information to the user.

Some other anomalies are not seen by the operating system but are detected by MANAGER, when it analyzes the results found by MALICE. Some years ago, there were still many bugs in the system and a rather frequent anomaly occurred: MALICE found a different number of solutions when it solved the same problem with different methods or when a combinatorial program was generated. In that case, MANAGER asks MALICE to solve this problem again, but it must check that each solution satisfies all of the initial constraints. If this verification fails for one solution, it means that it has found too many solutions and it knows one of them. If this does not occur, it means that it has missed one solution in one resolution; when comparing the solutions found with both resolutions, it knows which one it has missed. Therefore, it always finds the missing or erroneous solutions. The user has to complete the debugging of the system.

8.6. Explaining and meta-explaining

MALICE generates a trace including all of the rules it tries to apply. Thus, the system explains its solution starting from the events LINK and UNLINK as well as from the contradictions and solutions it has found; from the rules that create these events, it knows the events that trigger these rules, and then the rules that create these events, and so on until it comes to the initial constraints of the problem. All of the trials that are not found in this process are unnecessary to explain the solution. This method generates an explanation in the same way as an Explanation Based Learning system [27,34].

This explanation is sufficient enough to convince the user, but it is difficult to follow when the tree contains more than ten leaves: MALICE applies many rules and dozens of them may be applied in any path from the root to a solution or a contradiction. Although it is possible to check such a proof, it is a lengthy and difficult process. For instance, the tree for the solution of the crypt-addition DONALD has only two leaves and it is not so easy to check it.

I defined a heuristic concept, the "difficulty" of the explanation of a problem. It does not evaluate the difficulty in finding a solution, but the difficulty in understanding the solution found by MALICE. Its value is the number of "important" rules that appear in the explanation. This definition depends on the rules defined as important and this choice is made by the user. I consider a rule as important if using it is not evident for a human being; for instance, I did not classify the rules as important when they remove possible values for an unknown using the maximum degree of a node.

With this definition, the difficulty of the solution of the crypt-addition DONALD is 101 when the tree has only two leaves. If the system is forbidden to use some of the rules, it may find the solution while generating a tree with 5 leaves, which is also what a good human problem solver finds. Although the tree is larger, the difficulty of this new proof is only 91. Indeed it occurs that solutions with a small tree are a bit more difficult to understand than solutions with a larger tree; the reason for this is that it is necessary to apply many rules in order to reduce the size of the tree. However, I preferred to consider the smallest tree as the best solution: the concept of the number of leaves in a tree is objective while my concept of difficulty is subjective since it depends on the classification of rules being easy or difficult to understand. With this new solution, MALICE generates fewer constraints using algebraic derivations; these constraints enable the system to reduce backtracking, but increase the difficulty in understanding the solution. In practice, it is not easy for a human being to understand a solution when its difficulty is greater than 100. However, this is not true for MANAGER, which can efficiently use explanations for solutions that have a difficulty much greater than 1,000.

The explanation of a solution is used for learning. MANAGER counts as useful the number of times a rule is used in the explanation and as useless the number of times it was triggered and was not kept in the explanation. In this way, for all of the problems of the learning set and for each possible way of applying a rule, it knows the number of its useful and useless applications. With this information, MANAGER decides whether to keep or eliminate this possible trigger: if it decides to keep it, it defines its priority.

We have just seen why it is interesting to generate a trace to explain solutions and to learn how to use the rules. The meta-trace is useful for answering "why not?" questions that MANAGER asks itself. There are several kinds of "why not?" questions; for instance, at a bridge game, we can ask why 4 hearts have not been declared. Here, I consider another kind of "why not?" question: MANAGER asks itself why MALICE has not generated in one resolution the smaller tree that was found in another resolution. Let us assume that MALICE has used method M1 for the best resolution and M2 for the worst one. First, the reason for the difference in the tree sizes may be that the first unknown chosen for backtracking is not the same in both executions, one choice C1 may be better than the other choice. When this occurs, it orders MONITOR to choose C1 for the first backtrack. If, with this choice, both trees have the same size, MANAGER has meta-explained why the solution was not elegant with the second method: the choice of the unknown for backtracking was poor. If the solution is always less elegant, it will compare the solutions with methods M1 and M2 using the same choice C1 for this first backtrack.

Thus, when the first unknown chosen for backtracking is the same and the solution with method M2 is always less elegant, there must be another meta-explanation. Then, MANAGER compares both explanations in order to find the first useful result that is in the explanation when using M1 but not when using M2. With the meta-trace, it knows why this result was considered in the first place so that it can answer its "why not?" question.

Let us explain this with an example. The crypt-addition:

$$\text{THIS} + \text{ISA} + \text{GREAT} + \text{TIME} = \text{WASTER}$$

has only one solution. We consider here the formulation with the carries; using method M2 (which was my method), MALICE found a solution with two contradictions (so the tree had 3 leaves), while using another method M1 (which was the learned method) it found a solution with only one contradiction. It compared both traces and in both instances it found that it had generated the following constraint:

$$T+10*M+11*S+10*I = 9*E+100*R(4)+R$$

where R(4) is the carry of the second column from the right. Unfortunately, it discovers from the meta-trace that when it generated the solution using M2, it has decided to eliminate this constraint because it found that its interest was too low. So now the "why not?" question has been answered. Furthering this study, it finds that when it kept this constraint, the system is able to generate from this constraint a new constraint: R(4)≥1. With the possible values of the variables at this stage of the solution, it is easy to see that the left member of the equality is larger than 9*E+R, which is the value of the right member if R(4)=0. From this inequality, it finds 1 as the only possible value for R(4); now it is no longer necessary to backtrack with unknown R(4), as it occurred in the resolution using M2. Thus the best tree has only two leaves.

Although MANAGER generates useful information, it does not use it: the information is only given to the user who has to make the improvement. In the preceding example, this is difficult because the reason for eliminating the useful constraint is probably that MALICE generated too many constraints using method M2. One can decide either to keep more constraints or to modify the evaluation of the interest of a constraint so that the interest of the useful constraint is higher. Unfortunately, when one makes a modification that is good for a problem, one can also be very wrong for another problem.

In many other situations, MALICE missed the best solution not because it had eliminated a useful constraint but because it has not considered a sequence of derivations leading to an interesting result. When this occurs, MANAGER may generate a more interesting meta-explanation for the reason of the failure than in the preceding example. For instance, in a problem it was required to find 5 bijections A, B, C, D and E, each bijection is from a set of 5 elements on itself. I will give the description found by MANAGER of the reason why an elegant solution was not found by method M2 although it was found by method M1. Here is the useful sequence of steps found by M1 and not by M2 for the same problem:

Constraint 14=A(1)+A(2)+A(4)+B(1) found applying rule R8 to the constraints A(3)=B(1) and 14=A(4)+A(3)+A(2)+A(1)
Constraint 14=B(1)+A(4)+A(2)+D(2) found applying rule R8 to the constraints A(1)=D(2) and 14=A(1)+A(2)+A(4)+B(1)
MALICE backtracks, giving the value 5 to the unknown D(2)
Constraint 7=B(1)+A(2) found applying rule R79 (at that time in both executions MALICE knows the value of A(4) and of D(2) to the constraint 14=B(1)+A(4)+A(2)+D(2)
Constraint 7=B(1)+E(1) found applying rule R8 to the constraints A(2)=E(1) and 7=B(1)+A(2)
Constraint E(4)=E(1) found applying rule R8 to the constraints 7=B(1)+E(1) and 7=B(1)+E(4)

Rule R8 substitutes an unknown found in an equality constraint into another constraint; for the last constraint, the unknown is B(1) and is substituted in the second constraint by 7-E(1). Rule R79 normalizes a constraint; in particular, it replaces the unknowns whose value is known by this value. We can see why it is interesting to prove the last constraint E(4)=E(1): when a bijection gives the same value for two different elements, there is a contradiction since two different elements must have a different value (the maximum degree of the arrival set is 1). In the poor solution, MALICE had not generated this constraint, so it had to backtrack with unknown E(1), and it had naturally found a contradiction for each of its possible values: the tree included one more leaf because E(1) had two possible values. MANAGER knows the steps that is missing in the solution with M2: R8 was not applied to constraints A(3)=B(1) and 14=A(4)+A(3)+A(2)+A(1). In this case, I had wrongly put in my method a condition which forbade to consider R8 in this context. Naturally I used the results found by CAIA and modified my method so that R8 could be considered in such situations.

MANAGER only gives the steps that were not found in the poor resolution. For instance, both methods M1 and M2 found A(4)=2 that leads to a simplification with R79 and the constraint 7=B(1)+E(4) which is used in the last step. It is unnecessary to put the derivations that lead to these constraints in this "why not?" meta-explanation.

In other cases, MANAGER finds that the reason for a poor solution is that a trigger includes a condition that prevents MALICE from considering a useful rule. It occurred several times with my method because I had included too many conditions in some triggers. This explains why my method is often faster than the learned method: some rules are applied less frequently. In the contrary, its solutions are often less elegant since it rejects interesting applications for some rules. More examples are given in [36].

Sometimes the system can also find in the meta-trace that a useful rule has been wrongly delayed, wasting a lot of computer time. These are the useful rules with a very low priority; at the end of each execution, MANAGER lists all of the occurrences of such events. The user can use this information to improve the priority in the trigger of its rules. Moreover, in the explanation MANAGER highlights the useful rules with a low priority: it is likely that they are difficult yet important steps in the resolution.

With a meta-explanation, MANAGER can know why the execution of a useful rule has been delayed or has not been made at all, what are the difficult steps in a resolution and which triggers must be modified. It is better to meta-explain the resolution of simple problems: not only is it easy for MANAGER to generate the meta-explanation but it is also easy for the human user to understand it. In the same way, a human scientist prefers to analyze the resolution of simple problems when he may want to understand why his system is inefficient. However, when a reason has been found, the method can be improved, which is useful for all of the problems, even for the most complex ones. Although MANAGER can make interesting meta-explanations, it is unable to use them to improve its methods.

8.7. Generating new problems

MANAGER can find new problems in a family when it receives the definition of the constraints that must be satisfied by any problem of this family. However, it is unable to define new families of problems. It is important that MANAGER can generate new problems. Firstly it increases the number of problems so that the system will be well trained. Secondly, the problems found in the journals or in the books are often too easy because they are to be solved by human beings. In some cases, we can make them even more difficult to solve. For example, the problem DOMINOES given in the journal *Paris Match*: 8 dominoes are given and the goal is to place them on a 4x4 square so that it is magic, where the sum of the lines, columns and diagonals have the same value. Such as it is given, this problem is quite easy to solve because 3 dominoes are already placed on the board; the problem is further difficult when none of the dominoes is on the board at the beginning, but there may be many solutions. The problems of this family are given to MALICE without initially placing any dominoes on the board.

To generate a new problem, it is sometimes sufficient to change the value of some of the parameters that define a familiar problem. For instance, the Euler's Knight problem may be generalized into the problem of finding a circuit for a hopper (P,Q) on a board NxN. A hopper (P,Q) on square (X,Y) can move to eight squares (X+P,Y+Q), (X+P,Y-Q), (X-P,Y+Q), (X-P,Y-Q), (X+Q,Y+P), (X+Q,Y-P), (X-Q,Y+P), (X-Q,Y-P) as long as they are on the board. Euler's Knight is a special case with P=1, Q=2 and N=8. For these new problems, N must be even if the hopper must return to the starting point and P+Q must be odd so that the hopper can move on the white and the black squares. For instance, a new problem is to find whether the hopper (1,4), called giraffe, can complete a circuit on a board 8x8 (MALICE found that it is impossible) or 10x10 where there are many solutions. In order to generate a new problem for the family AUTOREF, it is sufficient to find one value for N and another for P with the meta-constraint N<P.

The user does not need any help in order to generate such new problems. However, for many problems it is not sufficient to modify the value of one or two parameters. For some of these problems, MALICE receives the formulation of a meta-problem with constraints, which could be called meta-constraints because they define the constraints that must be satisfied by each problem of this family. In this way, MANAGER can ask MALICE to find new problems, but only for the families where the meta-problem of finding new problems has been stated, it is unable to define new families of problems. There are 15 formulations of such meta-problems, for instance the first one generates new crypt-additions, the second one generates crypt-multiplications, the third one covers a square with dominoes so that it is magic, the fourth one defines cubes to superpose such as for the "instant insanity" problem, the fifth one generates Kakuro problems, etc. Some meta-constraints of the meta-problem check if the problem is correct, for instance that the result of a crypt-addition is equal to the sum of its operands. Some other meta-constraints prevent the system from generating too many impossible problems or from exploring areas where it is unlikely to find an interesting problem. The function NBVAL is very useful to express such heuristic meta-constraints.

When MANAGER has created a new problem, it asks MALICE to solve it. Then it considers whether it is interesting to keep this new problem. This interest depends on the resolution: one prefers problems with few solutions, specially those with only one solution. MANAGER also considers the characteristics of the other problems of the family: an impossible problem is more interesting when it is the first one in this family. It prefers problems that can only be solved after developing a very large tree because they are difficult. It also tries to keep some problems with only one solution which are solved without finding a contradiction, their tree has only one leaf: these problems are interesting for human beings because they are not too difficult. However, that does not mean that we can solve them easily: it is not alway easy to find a solution as elegant as those found by MALICE and we often have to backtrack in order to solve problems that MALICE directly solves.

Thus, MANAGER knows many difficult problems which are far more difficult than those found in the journals. One example of crypt-addition that it has created:

ABC + DEFG + ECHHGB + BFIHFC = IJECAD

The difficulty of this problem is 1,674; in comparison, DONALD with a difficulty of 101 is very easy. There is only one solution and MALICE never found a tree with fewer than 100 leaves. Another difficult problem in the DOMINOES family requires to build a magic square with the following dominoes: (0-5), (0-6), (1-5), (1-6), (2-3), (2-5), (3-4), and (4-5). The common value of the sums is 13 and there is only one solution if one takes the symmetries into account. MALICE must generate a very large tree to find the solution and prove that it is unique.

Let us give an example of a problem from the family LOGIGRAPHE [38]. The goal of these problems is to blacken some squares in a rectangular board so that the sequences of consecutive black squares agree with the sequence of numbers given for each row and column. For instance, if a particular row has the sequence (2, 4) there must be two consecutive black squares, at least one white square, and then four consecutive black squares. There may be any number of white squares (including zero square) at the beginning and at the end of the row. The left side of Fig. 2 gives a problem found by the system for a 10x10 board while its unique solution is on the right. The board is rather small, for instance some problems are for a 55x40 board. However this is a difficult problem, MANAGER rated 535 as the difficulty in understanding the solution found by MALICE. It is easy to see that, on the second row, we must blacken the seventh square: this square is always black for the ten possible ways to blacken 2 squares and then 4 squares on a row of 10 squares. In the same way, the four middle squares of the third row, where there is only a sequence of 7 black squares, must also be blackened.

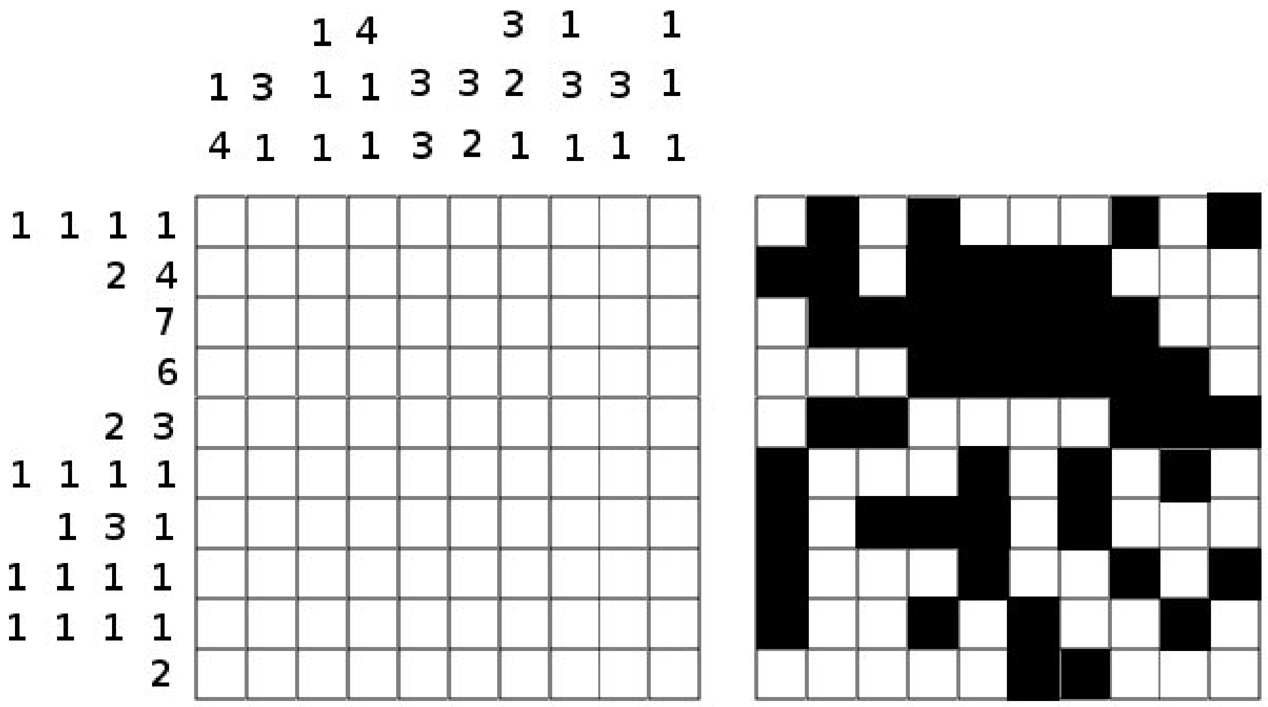
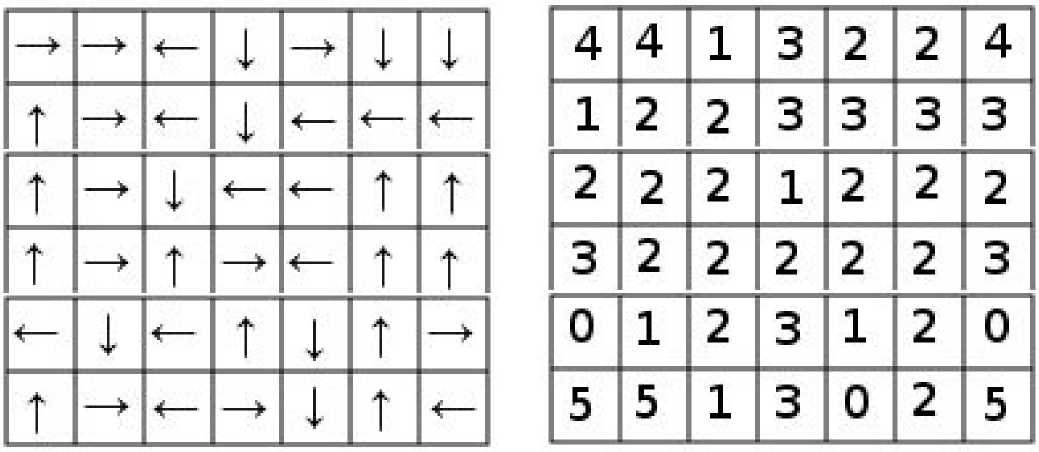


Fig. 2. A LOGIGRAPHE problem found by MANAGER and its solution.

The left board in Fig. 3 describes a third problem generated by MANAGER; it derives from the family ARROW. On a board NxP, each square contains an arrow with four possible directions: right, left, up and down. The goal is to find an integer for each square so that it is equal to the cardinality of the set of the integers of the squares pointed by the arrow: if the arrow points to the right, this is the number of different values which are at the right of the arrow. For instance, in figure 3, the top left square has a right arrow. Its value must be the cardinality of the set of numbers of the six squares on its right, their ordinate is 6 and the abscissa 2, 3, 4, 5, 6, and 7. The size of the search space is very high since there are 42 squares with up to 7 possible values.

The right board in Fig. 3 presents the only solution. It is easy to check this solution, for instance the number in square (1,6) is 4, the value of the squares on its right are 4, 1, 3, 2, 2, 4 , the corresponding set is (1,2,3,4), whose cardinality is 4. It is really difficult to find this solution and to prove that it is unique. With the intelligent approach, the smaller tree has 93 leaves and the difficulty of this proof is 2457. When MANAGER generated a combinatorial program, it developed a tree with more than 3 billions leaves.

The idea to consider this family of problems came from a problem published in the newspaper *Le Monde*, where MALICE found the only solution without backtracking. MANAGER decided to keep three of the easy problems that it generated and about 40 more difficult problems on boards of various sizes.



→	→	←	↓	→	↓	↓
↑	→	←	↓	←	←	←
↑	→	↓	←	←	↑	↑
↑	→	↑	→	←	↑	↑
←	↓	←	↑	↓	↑	→
↑	→	←	→	↓	↑	←

4	4	1	3	2	2	4
1	2	2	3	3	3	3
2	2	2	1	2	2	2
3	2	2	2	2	2	3
0	1	2	3	1	2	0
5	5	1	3	0	2	5

Fig. 3. An ARROW problem found by MANAGER and its solution.

8.8. Finding symmetries

Since Gelernter [7], we have been trying to realize programs that are able to find symmetries in the formulation of a problem. This is useful to prove new theorems easily or to reduce the size of the search space. Geometrical symmetries are a special case, but many problems have other kinds of symmetries: there is a symmetry when one or several other solutions can be automatically generated for each new solution. If that occurs, it is possible to add new constraints so that only the first solution can be found; then, the search space can be reduced, in some cases significantly. The main goal here is not to exploit symmetries [24], but to find them. A lot of research has been done in finding symmetries for constraint satisfaction problems; however, as we have seen at the beginning of this paper, the system tries to find solutions from first principles, and if possible, it does not use research conducted by human scientists. This application is interesting because, in order to solve a meta-problem, it uses the same methods than those used for solving problems; this reflexivity makes the development of such systems much easier.

MANAGER is only able to find symmetries in the formulation of a problem P that are a permutation of the unknowns; for that it defines a family of meta-problems whose goal is to find a bijection between the equality constraints that lead to a bijection between the unknowns of P. For each new meta-problem, it tries to find whether there are such bijections. This meta-problem is defined as a constraint satisfaction problem, but its data are generated by MANAGER from the formulation of problem P. When it has found such bijections, it uses them to generate new constraints from each constraint it creates while solving P as we have seen it with the magic cube. Moreover, for each symmetry, it checks whether the set of constraints of P, including even those that are not equality constraints, is converted into the same set by this symmetry. If that occurs, a symmetry has been found for problem P: from each solution A, another solution B can be automatically generated where each unknown has the value of its symmetrical unknown in A.

Naturally, MALICE must not waste its time generating solutions symmetrical to those that have been already found; thus, MANAGER adds some constraints to the formulation of the problem so the symmetrical solutions are no longer generated. As Puget [37] shows, when the value of the unknowns are all different, if there is a symmetry between the unknowns (V1,V2,...,Vn) and (W1,W2,...,Wn), all of the symmetrical solutions are eliminated if the constraint V1<W1 is added. Since the same constraint may be added to remove different symmetries, the number of these constraints is usually less than the number of symmetries. For the problems where the variables are not all different, Crawford et al. [6] show that one must add more complicated constraints to eliminate all of the symmetrical solutions; for instance, if there is a symmetry

between the unknowns (U,V,W) and (X,Y,Z), the associated constraint is:

$$U<X \vee (U=X\wedge V<Y) \vee (U=X\wedge V=Y\wedge W<Z) \vee (U=X\wedge V=Y\wedge W=Z)$$

These constraints are cumbersome when there are many unknowns; therefore, MANAGER also adds the weaker constraint $U\leq X$ which MALICE easily uses in order to remove most of the symmetrical solutions. Only the few remaining symmetrical solutions are eliminated by the stronger constraints, which are kept only if they are not too complicated.

Many symmetries are transformations such as geometrical symmetries or rotations, as we have seen with the magic cube, but some symmetries are not geometrical. For instance, the crypt-addition:

$$ABCDE + FGHAFC = FFIGIA$$

has a symmetry where every unknown corresponds to itself, except B which corresponds to H. Thus, MANAGER adds the constraint H<B so that the symmetrical solution cannot be generated.

MANAGER must be careful that MALICE does not generate the same symmetry twice when it solves the meta-problem: it is looking for a bijection between the constraints and a bijection between the unknowns. I was surprised to find that two different bijections on the constraints were often associated with the same bijection on the unknowns; for the meta-problem, these are different solutions. However, we are only interested in the bijection on the unknowns, so MANAGER must check, before storing a bijection on the unknowns, that it has not already been found associated with a different bijection on the constraints.

There are other kinds of symmetries that MANAGER is not able to find for the following two main reasons. Firstly, it has not received the definition of the meta-problem that could find this new kind of symmetry. Secondly, it only performs the search for symmetries on the initial formulation of a problem. For instance, with the magic cube, MANAGER has found 47 symmetries, but there is another symmetry which it has not found: a new solution is obtained when the value V of each unknown is replaced by 28-V. In order to find this symmetry, it is necessary to know when to apply this kind of substitution. This would be a new kind of symmetry, easy to implement; moreover, it also appears in many problems. But this symmetry cannot be found in the initial formulation of the problem: in order to prove that a constraint does not change when each unknown V is replaced by 28-V, it is necessary to know that 126 is the value of VAL because 28=(126*2)/9, 9 being the number of the remaining unknowns in a constraint. The constraint $F(14)\leq 14$ eliminates the symmetrical solution: it divides the size of the search space by almost 2. So, it would be necessary to give MANAGER the possibility to search for symmetries while MALICE is solving a problem.

MANAGER is not able to find other kinds of symmetries. A particularly interesting family of problems is DOMINOES, as seen in the preceding section, where a magic square is built with dominoes. Naturally, there are the geometrical symmetries of the square, but there are also two other kinds of symmetries. Firstly, there may be one or more double dominoes among the given ones, where both numbers are the same. It would be clumsy to count two solutions as different when the only difference is that one of these double dominoes has been rotated. Secondly, it is possible that the same domino comes up twice and exchanging them does not give an interesting new solution. For instance, MANAGER has generated the following problem with the dominoes (1,6), (2,3), (3,5), (3,5), (3,6), (3,6), (4,4), and (5,5). There are many symmetries since (3,5) and (3,6) come up twice and there are also two doubles: (4,4) and (5,5). These symmetries multiply the number of solutions by 16 even when the geometrical symmetries are removed. For this family of problems, I had to give two more constraint generators so that such symmetrical solutions are no longer generated; with these constraints, MALICE finds only one solution for the preceding problem.

Finding symmetries is a very complex meta-problem and MANAGER is not able to find all of them; moreover, there are many problems where there are no symmetries at all. It has actually found symmetries for 67 problems; for some of them the number of symmetrical solutions is enormous. For instance, the family MAGICYL is a variant of the magic square where the board is cylindrical; in that case, there are 2*N diagonals instead of 2, all the squares are equivalent because each unknown occurs in exactly four constraints and there are no longer center or corner squares; consequently, there are plenty of symmetries. For N=3, there are 431 symmetries, but no solution; for N=4 there are 127 symmetries and only 3 basic solutions. There are 799 symmetries for N=5 and only 36 solutions when symmetries are taken into account; without using any symmetries there would be 28,800 solutions!

8.9. Choosing which tasks to perform

For a human being, managing a large set of possible tasks can be described with the attraction exerted on these tasks. The attraction of a task depends on three factors: the cost, the importance and the presence of cognitive attractors [11]. A cognitive attractor is an element that is presented to the perception of the subject more or less strongly: for us a telephone ring, an order from a superior, a file on our desk are powerful cognitive attractors. MANAGER finds its next task using these three factors; they are determined in the conditional actions evaluating the triples (value-importance-number) for each candidate task. Human beings (including scientists) are not the best to make this kind of choice, they favor tasks with a low cost and with powerful cognitive attractors: naturally, at the end of the day, they feel that they wasted their time on uninteresting tasks. With its knowledge for choosing the next task, MANAGER chooses its tasks satisfactorily during an execution that lasts several weeks, it does not procrastinate as so many human beings do.

Thus, at the beginning of each loop, MANAGER chooses its next task. That may be any activity we have already seen: solving a problem which has not yet been solved with a particular method, conducting an experiment so that it can find interesting events, generating new problems, comparing the results obtained with different methods, meta-explaining why a resolution is more elegant with a particular method, writing a combinatorial program to solve a problem, understanding why there was an anomaly during the resolution of a problem, learning new methods, etc. In order to choose its next task, it uses the algorithm as described in section 6 with a set of conditional actions that enables it to compute the interest of the possible tasks, then it chooses the most interesting one.

To illustrate this point, Fig. 4 presents a problem found by MANAGER after running for about 2 millions of seconds. In the ROOK family of problems, the goal is to find a circuit for a rook on a board where some squares are forbidden. The rook must go once and only once on each square that is not forbidden and return to its starting square. MANAGER generated this problem for a human being so that it is neither too easy nor too difficult; its difficulty is 47 which is fairly good for this goal. There is only one solution. Naturally, it generated more difficult problems in this family, specially with larger boards, but only for its own use.

Although CAIA can run for several months, I stop it after 2 or 3 millions of seconds. I do not kill it but I freeze it so that it is possible to restart. However, this is no longer very interesting because, while it was running, I watched what it was doing, which in turn gave me many ideas to improve its behavior in a new version. Thus, I prefer to start the life of a new and improved CAIA.

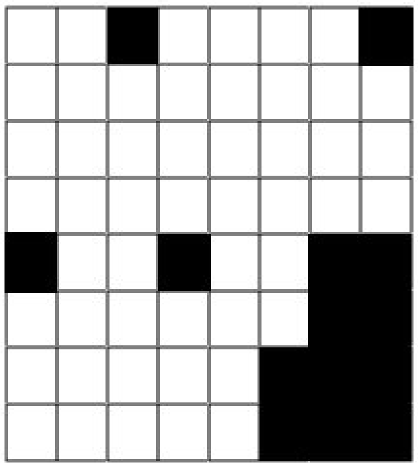


Fig. 4. A ROOK problem found by MANAGER.
The goal is to find a circuit for a rook

9. ADVISOR

Regularly a fourth agent, ADVISOR, assesses the behavior of MANAGER. It considers what problems have not yet been solved, analyzes the experiments, compares the results obtained with the three methods, compares the results of the combinatorial programs with those found in the intelligent approach, and also considers the new problems MANAGER has generated. Finally, it synthesizes these analyzes in a report for the user. In this report, it describes the important events and gives a summary of its experiments and their results. ADVISOR only communicates with MANAGER.

ADVISOR selects problems with interesting solutions, for instance a crypt-addition formulated without carries was one of the most difficult resolutions to understand:

ABCDE+FFGHFI=FDICCC

There are 36 solutions and the difficulty is 7,011. The difficulty of:

ABC + DAEDDF + BEDB + GGDCHA = IJFGAI

is only 4,004, but ADVISOR notices it because there is only one solution. It gathers useful information to improve the system such as the problems where the choice of the unknown for backtracking was particularly poor and the families of problems where this happened frequently.

ADVISOR also considers MANAGER's advancement since the preceding assessment: it has succeeded in solving a very difficult problem or it has discovered very interesting problems. If there are few or no such results, perhaps the time has come to freeze the life of this CAIA.

These analyzes are useful for MANAGER. For instance, ADVISOR can decide to eliminate some problems because even more interesting problems have been found in their family. It may also prompt MANAGER to give a special effort to a very difficult problem that it has not yet been able to solve.

ADVISOR guides MANAGER the same way as MONITOR guides MALICE.

10. ZEUS

MONITOR is able to see that MALICE is always applying the same rules with the same arguments or that it is always generating the same constraints. It can stop such loops, for instance it may inhibit or restrict the use of some rules. In the same way, MANAGER can see that it is always performing the same experiment. But there are loops that MANAGER or MONITOR cannot detect in order to regain the control. This may occur for instance when if is a loop when MALICE is executing a rule.

ZEUS is acting independently of the other agents. It can examine some of their results and their recent actions. From the value of this observation, it may change the value of some parameters so that the various agents will modify their behavior, it may even stop them and restart MANAGER after storing some indications of the serious anomaly that it has just seen.

Interrupts by the alarm clock are used to stop these loops: the system calls the agent ZEUS for every S seconds no matter what ADVISOR, MANAGER, MONITOR or MALICE is doing. ZEUS examines whether CAIA has advanced since its preceding interruption. If it is always at the same point, ZEUS gives CAIA a second chance; but at the next interruption, if it is still at the same point, ZEUS assumes that the system is in an infinite loop. In that case, it stores some information on the current task, particularly the subroutine that is looping. Then it indicates MANAGER to be cautious if it tries this task again, and automatically restarts it: CAIA does not stop and it does not forget the information on its past. Later on, MANAGER will try to understand what happened, for instance after recompiling the dubious subroutine so that it can trace each instruction.

ZEUS is also necessary when MANAGER solves a problem with a combinatorial program that it has generated. When the allocated time is exhausted, it orders the program to stop, but if it does not comply then it is interrupted.

ZEUS functions well: when it believes there is a loop, it is correct, it stores all the necessary information to find where it occurred and then gives the control to MANAGER. Such a module is essential for a system that has to function for weeks without stopping or looping needlessly.

11. Results

MANAGER compared the results for three methods: G is the given method that I generated myself, I is the initial method it generated, and L is the learned method. In the final step of its third life, MALICE has more than 1,000 problems from more than 100 families. In these results the meta-problems are not included; there are 38 meta-problems for generating new problems and they belong to 15 families of meta-problems; another meta-problem finds symmetries in the formulation of a problem. The problems were taken from various newspapers and journals, several interesting and difficult problems were found in the CSPLib [39]. When there are many solutions, several millions or even billions could be found in some problems, MANAGER does not try to find all of them, instead it stops at 100 except if it has already developed a large part of the tree. In that case, MANAGER may allow MALICE to find all of the solutions.

A PC with a 2.4 gigahertz Pentium 4 was used by CAIA.

11.1. General results

We will compare the results obtained with the three methods: the given method (G), the initial method (I), and the learned method (L). I do not take into account the results for the meta-problems as well as those for the problems created by MANAGER but which were not kept because their interests were too low: these problems were only solved with the learned method. This learned method has been generated using only the solutions of the problems in the learning set. I do not include 17 problems which were not completely solved because there were too many solutions, more than one thousand. Indeed, the time necessary to find a small subset of a large number of solutions does not have any meaning because in an execution the system may be in a search space area where there are many solutions, while in another execution it may be in an area where they are scarce. Now it has 1,053 problems to consider. Moreover I do not keep 20 of them since they require a whole lot of time, more than 1,000 seconds: the CPU time for solving one of them is so high that it is as lengthy as solving many other problems. For these few problems, a small difference in the choice of the unknown to backtrack may considerably increase the time and the number of contradictions necessary to solve the problem so the results would have no meaning at all. The results describe what occurred for the 1,033 remaining problems.

Table 3 presents the results for the 512 problems of the testing set which have 4,862 solutions. MANAGER defines the problems which will be in this set; they are randomly chosen, but if a family includes two or more problems, at least one of these problems must be in the testing set and one of the others must be in the learning set. The number of leaves in a tree is the sum of the number of its solutions and of its contradictions. "Nodes" is the total number of nodes of all the trees in the solutions of this set of problems.

	contra-dictions	***seconds***	***nodes***
Method G	149045	11352	829295
Method I	118668	33676	643504
Method L	117833	16436	712706

Table 3. Results for the 512 problems of the testing set.

Table 4 shows the results for all of the problems, including those of the learning set, there are 1,033 problems and 10,027 solutions.

	contra-dictions	***seconds***	***nodes***
Method G	225745	16044	1514205
Method I	190136	77673	1241779
Method L	187196	24617	1394486

Table 4. Results for all of the problems.

The explanations given by the three methods are completely different, even when they develop exactly the same tree. Each method has preferred rules and it is natural that they more frequently occur in the final explanation of the solution. Two solutions given by two methods are as different as two solutions given by two human problem solvers.

Let us now compare the results with methods G and L. With G, the one I generated, the solutions are usually less elegant because the trees are larger, but MALICE is faster with G than with L: the tree generated with G is larger than with L for 296 problems while it is smaller for only 127 problems; both trees have the same size for the other problems. Comparing both versions I and L made by MANAGER, the initial version I is very slow, which is normal, but it generates trees with a few more leaves than L even for the problems of the testing set. However, the number of nodes for I is lower: L develops more nodes after a backtrack before finding out that there is a contradiction. The methods generated by MANAGER stands in comparison with the method that I created. In all fairness, the comparison would be more in favor of MANAGER because I often modified my own method by taking account of my mistakes, which I found from the analyzes made by MANAGER.

It is interesting to understand why version L usually gives more elegant solutions than version G. The main reasons are that I have made some mistakes and was too lazy. In some cases, I thought that some rules could not be useful in certain contexts, thus I added conditions so that MALICE would not also use them. This speeds up the system and it is one of the reasons why G is usually faster; however, if it is sometimes better to use this rule then MALICE cannot conclude faster with version G, thus the tree is too large. Moreover, generating and modifying a method is a difficult process. It is tempting not to add a rule to the set of rules which would be considered after an event when we think that it would be not likely be useful. In that case, version G does not use the set of rules in the best possible way. On the contrary, MANAGER is never lazy because it generates a new method in less than one second.

11.2. Failures

Naturally, there are many problems that MANAGER is unable to solve. Several families of problems depend on a parameter and the difficulty of the problem grows with the value of the parameter. For instance, if N is the size of a magic square, which includes NxN squares, it is rather easy to solve it when N=3, but it is quite difficult for MALICE to solve it when N is greater than 11. In the same way, the difficulty of AUTOREF grows fast with the value of P; among its problems there is one with P=27, but it would certainly fail with larger values of P.

For two of the problems given to MALICE, the value of one of the parameters was too high and it failed; it is interesting to give such problems to see how MANAGER reacts when it finds a problem too difficult to solve. It writes a combinatorial program, backtracks with other unknowns, inhibits some rules, but these attempts were not sufficient enough to solve these problems. It would be necessary to write a more powerful combinatorial program and find new methods to restrict the growth of the search tree.

The third problem it failed to solve is the only problem in its family: it is kind of a magic 5x5x5 cube. All of the lines with 5 small cubes, including the diagonals of the faces and those of the cube, must have the same value. The goal is to find a bijection from the set of numbers 1 to 125 on itself and also the value of the common sum. There are 109 constraints and each one contains 6 unknowns; each unknown may have 125 values, thus the search space is enormous. MANAGER has found 191 symmetries, so it created 42 new inequality constraints to remove symmetrical solutions. Finding these symmetries is already a difficult problem since its definition includes 8,838 constraints. When it tries to solve the main problem, it quickly finds 315 as the common value of the sums and also 63 as the value of the center square. This last result is certainly difficult to find, it was only discovered in 1972 by Schroeppel. Now most of the constraints have 5 unknowns, and some of them only have 4 unknowns. The system finds some new constraints, but all of them have at least 4 unknowns, which is not enough to restrict the search.

Since the system was not able to find any solutions with the intelligent approach, it generated a combinatorial program. This program contains 44 loops, and for each of them the variable can have more than 100 values, thus the program has not found any solutions. This problem is very difficult: its first solution has just been found in 2004.

11.3. The intelligent approach

It is interesting to compare the performance of the intelligent approach with the results obtained after generating a combinatorial program. When the time for finding a solution with the intelligent approach is short, at most a few seconds, the times are almost the same: writing, compiling, loading the new program already takes some time and even if the running time is almost nil, the whole process takes at least one second.

However, when the solution with the intelligent approach takes at least 10 seconds, in most cases the solution with the combinatorial program is much faster. For instance, in most crypt-arithmetic problems, there are at most 10 unknowns which can take at most 10 values each. When the combinatorial program is written cleverly, it only backtracks with the unknowns that appear in the operands, not with those that only appear in the result of the operation. Moreover, as MANAGER begins with the intelligent approach, it has already removed many possible values for the unknowns. Thus, for most crypt-arithmetic problems, the tree has fewer than 100,000 leaves and the combinatorial program solves the problem in less than one second.

Not all of the problems were solved using these both approaches. The first reason is that the combinatorial approach is tested only for problems where the intelligent approach had to backtrack. Moreover, for several problems MANAGER knows that it does not know how to write a combinatorial program: this is the case for 9 families of problems, which include 106 problems. For these problems, the intelligent approach is the only way to find a solution. Among the problems that were simultaneously solved by the intelligent and the combinatorial approach, there were 337 problems where the combinatorial program was faster whereas for 299 problems the times were equivalent. Finally, there are 19 problems where the intelligent approach is better than the combinatorial program. For 13 of them, the intelligent approach was much faster, but for the last 6 problems the combinatorial program was not even able to find a solution while the intelligent approach was able to find all of them. On the other hand, 4 problems were solved by the combinatorial program but were not solved by the intelligent approach because it required too much time. In order to solve them, it would require at least half a day of CPU; MANAGER estimated that too much time would be wasted for problems which had already been solved using the combinatorial approach.

When the intelligent approach is the best one, the problems usually contain a kind of reflexivity such as AUTOREF or ARROWS. For the problem of ARROWS given in 8.7, the intelligent approach was able to solve the problem in 29 seconds while 952 seconds were necessary for the combinatorial program. For the AUTOREF problems, the advantage of the intelligent approach increases with P: for P=27 and N=5, it was able to find the two solutions in 379 seconds while the combinatorial program had developed only 7 thousands of the tree after 40,000 seconds. Let us explain the reason for this failure of the combinatorial program; we have seen that for AUTOREF problems, MALICE finds a new constraint with the rule R5. For P=27 and N=5, this constraint is:

$$F(0)+5 = F(2)+2*F(3)+\dots +25*F(26)+26*F(27)$$

As MALICE knows that $F(0)\leq 27$, it immediately finds that if it gives the value 1 to two of the F(I) where I is large, the preceding constraint is FALSE because their coefficient in this constraint is large so the right part is greater than F(0)+5. Thus, when the value 1 is given to one of the F(I)'s where I is large, it immediately finds that 0 is the value of all the other F(I)s where I is large. The combinatorial program does not find this and it checks the constraint only when it knows the value of all the F(I)s; for almost all of these combinations of values, the constraint is false because the right part is much larger than the left one when the value of two F(I)s is 1. When there are many unknowns, the specific combinatorial program unnecessarily develops a large tree and it takes too much time. However, the combinatorial program is reasonably efficient because it uses the fact found by MALICE, that for large I, the F(I)s have only two possible values, 0 and 1, instead of 27 possible values for each unknown. Naturally, for I small, both the intelligent and the combinatorial programs have to try more values for each F(I), but this is feasible.

Finding Langford's numbers is particularly interesting because the intelligent approach is obviously the best one. For the Langford problem L(K,N), the goal is to arrange K sets of numbers 1 to N so that each appearance of the number m is m numbers on from the last: two successive appearances of number m are separated by m numbers. For example, L(2,4) has only one solution 41312432. Naturally, the symmetrical solution obtained by reversing a number is not considered. For L(3,9), MALICE finds the 3 solutions in 31 seconds and proves that there is no other solution. One of them is:

347839453674852962752816191

We can check that every number from 1 to 9 occurs exactly three times and that there are 7 numbers between the first and the second occurrence of 7 and between the second and the third. On the other hand, the combinatorial program generated for solving this problem had only developed a tiny part of the tree after 10 hours. The reason for the success of the intelligent approach is that for such problems it is not adequate to choose the order of the unknowns for backtracking a priori: the best choice strongly depends on the values taken by the unknowns that were already defined. On different paths from the root to the leaves, the best choices of the unknowns are completely different.

MANAGER can write a combinatorial program from the start, without having an intelligent phase. For most problems, they take the same amount of time; for instance, with the crypt-addition DONALD the tree has 1,296 leaves instead of 176 leaves when the program is generated after some intelligent deductions (and only 2 with the intelligent approach). Most of the time is spent on writing, compiling and loading the program, the running time for such small trees is completely negligible, but it becomes large with trees that have at least one hundred million leaves. However, for some problems it is essential to begin with an intelligent phase before writing the program. For instance, for the AUTOREF problem with P=23 and N=5, both solutions are found in 15 seconds with the intelligent approach, and in 48,154 seconds with the combinatorial approach after an intelligent phase. The program was stopped after 40,000 seconds when it was generated without an intelligent phase; it had developed less than one thousandth of the tree. The reason for this is that it is not aware of facts that were found in the intelligent stage: for large values of I, F(I) can take only two values 0 and 1, and there is also a restriction for the smaller values of I. Thus it tries the 24 values between 0 and 23 for each of the 24 unknowns, which is over the limit even for a fast combinatorial program. On the contrary, it is much faster to generate the combinatorial program without an intelligent stage in the case where it wastes a lot of time without significantly decreasing the size of the search space. This occurs for the problems of the GOLOMB family as defined in section 8.3.

11.4. Quasi-groups

An order N quasi-group is defined by a multiplication table of size NxN in which each element occurs exactly once in every row and column. It is idempotent if a*a=a for every element a. It is interesting to determine the existence (or non existence) of quasi-groups of a given size with additional properties. Fujita, Slaney and Benett [9] have made many experiments with a general theorem prover called MGTP; this system solved the problem of the existence of several quasi-groups. For the quality of its results, this paper won the award for the best paper at the IJCAI'93. MALICE considered problems with one of the following properties which must be true for every couple of elements a and b:

QG3: (a*b)*(b*a)=a

QG4: (b*a)*(a*b)=a

QG5: ((b*a)*b)*b=a

QG6: (a*b)*b=a*(a*b)

QG7: (b*a)*b=a*(b*a)

It is useful to add some symmetry breaking constraints. In his experiments Fujita included, in the definition of this family of problems, the constraints a*N≥a-1 for a∈[1:N-1] where N is the order of the quasi-group. This symmetry was not found by MANAGER but these constraints were included in the definition of the family so the problems were exactly the same for both systems; in that way we can compare their performance.

The definition of this family includes the symmetry breaking constraint, the five properties QG3-7 and possibly the idempotent constraint; for these last constraints, the value of a parameter indicates which ones must be considered for a particular problem of this family. The quasi-group constraint is given by a KNOWING INCOMPATIBILITY constraint.

Table 5 presents the results of MGTP and those of MALICE for some of these problems. QG7-8 indicates that the problem is the existence of quasi-groups of size 8 with property QG7. It is followed by N if the idempotent constraint is not included as with QG5-10N. For this family of problems, MALICE is unable to generate a combinatorial program, so these results are those of the intelligent approach. Table 5 summarizes those results. "Leaves" indicate the number of leaves of the tree generated for these experiments while "CPU" is the computer time in seconds.

Problem	Leaves MGTP	Leaves MALICE	CPU MGTP	CPU MALICE
QG3-7	183	102	6	6
QG3-8	3875	2101	28	132
QG3-9	312321	113507	1022	9465
QG4-7	123	85	6	5
QG4-8	3516	1781	23	125
QG4-9	314925	145122	1127	13149
QG5-7	9	5	3	1
QG5-8	34	15	7	2
QG5-9	239	32	12	4
QG5-10	7026	275	66	34
QG5-11	51904	625	224	121
QG5-12	2749676	7956	13715	1938
QG5-10N	4474508	193359	13101	20912
QG6-7	7	5	2	1
QG6-8	20	11	6	3
QG6-9	160	54	14	6
QG6-10	2881	329	43	26
QG6-11	50888	4931	248	359
QG6-12	2420467	54880	8300	5154
QG7-7	182	38	4	2
QG7-8	160	244	5	10

Problem	Leaves MGTP	Leaves MALICE	CPU MGTP	CPU MALICE
QG7-9	37026	2165	90	88
QG7-10	1451992	33187	2809	1467

Table 5. Results for the quasi-groups.

Naturally, the number of solutions is always the same in both experiments, for instance 178 solutions for QG4-9 and no solution for QG4-8 or for QG5-12. The sizes of the trees are always smaller with MALICE. The CPU times are roughly the same, but it is difficult to compare the computers: MGTP ran in 1993 on a computer with 256 processors, the parallelism nature of their method enabled them to have an almost linear speed-up with the number of processors. With Moore's law, 256 correspond to 8*1.5 years, that is 12 years which correspond roughly to my 2004 PC. Thus the performance of MALICE for this family of problems is, for the CPU time, equivalent to the performance of MGTP and even better when we consider the size of the tree. MALICE also solved QG5-13, which was not solved by MGTP: no solution and 530,746 leaves in 55,844 seconds.

11.5. Experiments

The experiments made by MANAGER are very useful. Some of them, such as helping to find bugs, were interesting at the beginning of the implementation of the system because they really helped me to find some bugs. For some other experiments, MANAGER is not able to use them to improve its behavior although it has all the information necessary to do so. However, this is an important step in the collaboration between an artificial and a human scientist: MANAGER performs and analyzes experiments more thoroughly than human beings so we can use its results to create an even better problem solver. Each part is doing what he/it does the best, although using the results of these experiments to improve the system can be very difficult even for human beings.

One particularly useful experiment is conducted when MANAGER asks MALICE once again to solve a problem choosing another unknown for its first backtrack; often it finds more elegant solutions than the one found with its initial choice. MANAGER tried this experiment for 673 problems and for 514 of them MALICE found a solution with a smaller tree, with fewer leaves. On the whole, there were 8,495 attempts to solve a problem using another choice than the one selected by MONITOR and for 2,676 attempts, that is 31%, a tree with fewer leaves was generated. If we consider the problems instead of the attempts, this experiment was made for 673 problems, and for 519 of them, that is 77%, at least one choice leads to a smaller tree. Often the improvement is only one or two leaves, but for 155 problems, the new tree had fewer than one half of the leaves of the tree generated with the first choice. For instance, MANAGER created the crypt-addition ABAC+DBEADB=DAGFBH formulated without the carries (only one constraint expresses that it is an addition). This problem has 18 solutions and when MONITOR first chose to backtrack with H, MALICE found 1,216 contradictions. In the experiments, it found only 2 contradictions when it first backtracked with B. Such differences are rare when a problem has at least one solution, but it is very frequent for impossible problems; for such problems, it often occurs that it is possible to find a contradiction locally. If the unknown is chosen in that area A, MALICE quickly finds that there is a contradiction, but if it is in another area, it develops a large tree before if goes to the area A and finds the contradiction. It is not always easy to find places strongly constrained, even for human beings. Most of the problems where the choice of the unknown was very poor were created by MANAGER; usually this choice is rather satisfactory for the problems found in books or journals. This shows how it is interesting to give a system the possibility to generate very difficult problems.

Some other experiments were made for the cases where MALICE found a contradiction shortly after backtracking or when an unknown had the same value for all of the possible values given to the unknown chosen for backtracking: this shows that there is perhaps a possibility to reduce the size of the tree if it uses more rules before deciding to backtrack. In the experiments performed in those situations, MANAGER has found that it is sometimes possible to eliminate a few backtracks, but this seldom occurs and it would also increase the time necessary to solve a problem. On 3,894 situations, it quickly found the value of an unknown and decided to see whether it could find the same value before backtracking, there were only 237 cases where it could be found before backtracking; these situations were found while solving 415 problems and for only 46 problems at least one trial was successful.

There were 264 situations where an unknown always had the same value after choosing an unknown for backtracking; in 149 of them it could find this value before backtracking, which is in 56% of the cases. These situations were found while solving 111 problems and the system was successful at least once for 27 of these problems, so there are a few problems in which this happens more frequently. Moreover, among the 149 successes, 141 was made for problems of one particular family; this fact would certainly be useful to create a more clever MANAGER. From these results, we can see that an improvement is feasible, but it is not a priority because these situations seldom occur.

In 128 problems, MANAGER asked MALICE to be less restrictive in the use of potentially dangerous rules. For only 37 of them it found a better solution, but that required much more time. For these 37 problems, on the whole there are only 91 contradictions instead of 181, but it takes 540 seconds instead of 109. Thus this improvement is costly, and except for some interesting problems where MANAGER wants to find a more elegant solution, it is not necessary to spend more time on these rules.

When there are several attempts to solve the same problem, MANAGER often tries to understand why it has not found the best solution in all of its attempts; in many cases it has found a meta-explanation. Firstly it may be due to a poorer choice of the unknowns when it backtracked. Secondly, it may be that it has found the "good" constraint, but it has not kept it due to low interest. Thirdly, it may not have applied a rule with the appropriate arguments, and in that case there is a flaw in its method. Finally MONITOR may have over-restricted the use of the dangerous rules. Overall, it tried to compare 202 problems where the number of contradictions were different when MALICE used different methods. It was not able to find a meta-explanation for 5 problems. For 67 problems, the reason was that the choice of the first unknown for backtracking was not the same: when MALICE chose the same unknown in both cases, the number of leaves of the trees was the same. For each of the 130 remaining problems, MANAGER found a correct meta-explanation.

These experiments show that there is one serious possibility to improve the quality of the solutions: to modify the choice of the unknown for backtracking. On the contrary MALICE may not improve a whole lot in modifying the choice of its rules, it seems they have already been used efficiently. Unfortunately, for all of these experiments, although MANAGER has all the information to improve the system, it does not know how to learn from the failures it detected; however, this information can be utilized by the user of the system.

12. Future work

Trying to improve the performance by modifying the learning expertises could only lead to a small improvement, for very few problems it could solve them perhaps 5 times faster. However, when we find a better formalization of a problem or when we add a new rule, it may occur that MALICE solves the same problem 10,000 times faster; moreover it generates a considerably smaller tree. This happened for instance for AUTOREF with large values of P or for the last formalization that I found for LOGIGRAPHE problems. Thus, we have to work on two more important limitations of the system. Firstly, a family of problems is given in a formalized language and a large part of the work has been made by the user when he makes this translation: it would be better to describe the family in a natural language. Secondly, it would be necessary to increase the theorem proving ability of the system significantly: it must be able to generate new rules and modify the formulation of a particular problem more drastically. For both of these reasons, I am over-helping the system. The user's importance is too high in CAIA, the final goal is to build a system that does not depend on human intelligence at all.

12.1. Understanding problems given in a natural language

A system can be helped significantly by the way a problem given in a natural language is formalized. It would be better to state problems in a natural language so the system would translate it into MALICE's language to define families of problems. More than forty years ago, Daniel Bobrow [2] had already shown that STUDENT could solve electricity problems stated in English. For constraint satisfaction problems, C. Lopez-Laiseca [21] had realized a system translating some of these problems from French into ALICE's formalism. This attempt had shown that this goal is possible for restricted families of problems, but it is difficult to generalize it for every family. Human beings even have difficulties in formalizing a problem given in a natural language. Defining a problem as a constraint satisfaction problem is one of the most difficult steps in research.

Let us consider the following problem, given to the students who begin the study of algebra:
A father is 40 years old and his son 10 years old. When will the father be twice as old as his son?

Good students give the following solution: if X is the wanted number of years, in X years the father will be 40+X years old and the son 10+X. Thus they write 40+X=2*(10+X) and the solution is 20. In 20 years the father will be 60 years old and the son will be 30. Everybody, including the teacher, is happy, but the formulation of the problem is wrong: a staircase function has been replaced by a linear function. If we want to solve this problem correctly, we must know the father's and the son's dates of birth. This problem was given to MALICE with the following parameters: DS, MS and YS for the day, month and year of the son's date of birth and DF, MF and YF for the father. N is the coefficient multiplying the son's age, 2 in the preceding problem. In the formulation given to MALICE, it has to find 3 unknowns, R, S and T. The value of T is 1 if the father's birthday is before the son's, otherwise its value is 2. Four constraints define the value of T from the parameters DS, MS, DF, and MF; I do not present them here because they are obvious. The value of R is the year where the event occurs. The value of S is 1 if it occurs between the beginning of the year and the first birthday, 2 if it occurs between both birthdays and 3 if it occurs after the second birthday. There are four constraints defining R and S when T is known:

$$\mathrm{OR}(S\neq 1,\ R-YF-1 = N*(R-YS-1))$$
$$\mathrm{OR}(S\neq 2, T\neq 1,\ R-YF = N*(R-YS-1))$$
$$\mathrm{OR}(S\neq 2, T\neq 2,\ R-YF-1 = N*(R-YS))$$
$$\mathrm{OR}(S\neq 3,\ R-YF = N*(R-YS))$$

If the father was born on February 1, 1900 and the son on August 1, 1930, the problem was stated between August 1, 1940 and January 31, 1941. MALICE finds T=1 and three periods where the father is twice as old as his son:

1-8-1960 to 31-12-1960 with S=3, R=1960. The father is 60 years old and the son is 30.

1-1-1961 to 31-1-1961 with S=1, R=1961. The father is 60 years old and the son is 30.

1-2-1962 to 31-7-1962 with S=2, R=1962. The father is 62 years old and the son is 31.

There are two separate periods of time and their sum is exactly one year. This is general when N=2; for N=3 there is only one solution which covers less than a year and for the higher values of N there is one or zero solution depending on the father's and the son's dates of birth.

This problem is complicated only because it is difficult to formalize it from its English definition, for MALICE it is trivial.

Let us consider another problem where the difficulty comes from its translation from English into a formalized language:

Peter and Sophie are two mathematicians; the first one receives the product of two integers between 2 and 100 and the second one receives their sum. Each of them does not know the number that is given to the other person but they must find these integers. Peter says: « I cannot find them! » . Sophie replies: « I knew it. » . Then Peter says: « Then I know them! » and Sophie concludes: « Then I also know them! » . The question is: what are these numbers?

It is very difficult to formalize this problem, but once it is formalized, it is quite easy for MALICE to find the solution. I defined five parameters, whose value is the set of possible products or sums after each of the preceding sentences. With this formalization of the problem, MALICE immediately finds that the set of possible sums after the last sentence of Sophie has only one element which is 17 and finally the set of possible products has only one element which is 52. Thus the numbers are 4 and 13. The definition of the problem directly gives the solution, so it is impossible to find an easier problem for MALICE! The difficulty of this problem is only in its formalization: it is not easy to define correctly the five sets. MALICE has not really solved the problem, it only reaped the benefits of the user who formalized the problem.

Moreover there is a serious ambiguity in the formulation of this problem: do Peter and Sophie know that the integers are less than 100? That is not explicitly stated, and in my first formulation I assumed that they knew this restriction. However, if the problem is reformulated assuming that they do not know it, MALICE finds another solution: the product is 244, the sum is 65 and the integers are 4 and 61. That seems paradoxical, both integers are less than 100, nevertheless they are not found when it is assumed that Peter and Sophie know that they are less than 100! The reason is that 244 has two decompositions, 4*61 and 2*122; if Peter knows that the integers are less than 100, the second decomposition is not acceptable, so if the product given to Peter were 244, the solution could only be 4*61: he would have immediately found the integers. Thus, 244 would not belong to the first set of possible products and it is normal that the second solution was not found. On the contrary, if they do not receive an upper bound for these integers, 244 belongs to the set of possible products: Peter cannot find the integers since there are two acceptable decompositions and this will lead to the second solution. This remark shows the difficulty for human beings in formalizing a problem: in the literature I have always seen only the first solution. It seems that most people do not see that there is an ambiguity and those who see it erroneously believe that it does not matter. I, on the other hand, had the chance to be helped by CAIA.

Formalizing problems is a very difficult task which requires much intelligence. Therefore it must be conducted by an AI system and not by human AI scientists.

12.2. Increasing MALICE's theorem proving ability

MALICE's mathematical possibilities are too restricted. It is able to simplify a mathematical formula, to generate new constraints or to find some symmetries in the formulation of a problem, but this is not sufficient. For instance, it cannot create new rules. Let us take the example of rule R5 as seen in 5.1.

Rule R5 is very helpful for solving AUTOREF problems. I have given it to MALICE, but this is not a rule that everybody is familiar with, I did not know it when I tried to solve these problems. To discover it, I first had to perform a special kind of theorem proving since I did not know which theorem to prove. Let us present this rule in a simplified form: n unknowns D1, D2,...., Dn are taking their values among the integers 0, 1, 2,..., p and there are p constraints:

Ai = NBVAL(i; D1, D2,.., Dn)

So Ai is the number of unknowns that have the value i. In this case, the rule states that:

$$0xA0 + 1xA1 + 2xA2 + 3xA3+=....+ pxAp = D1 + D2 + D3 +....+ Dn$$

This rule is evident for a human being: if K unknowns have the value L, their contribution to the sum of the unknowns is KxL. The rule requires p and not p+1 constraints: the knowledge of A0 is unnecessary because it is multiplied by 0. In order to find or understand this proof, we used information on NBVAL that is not given in a formalized language. Rule R5 is essential for the performance of the system: for the problem AUTOREF with P=17 and N=3, MALICE solves the problem in 28 seconds when it can use rule R5 and in 7,648 seconds when it is forbidden to use it. For P=27, it is unable to solve the problem without using this rule. Although it seems very specialized, it can also be used with other problems: if the unknowns Di are boolean and if there is the constraint:

$$A = NBVAL(1; D1, D2,..., Dn)$$

Rule R5 generates the new constraint:

$$D1+D2+....+Dn = A$$

The formulation of the problem has been changed, the constraint with NBVAL becomes a usual algebraic constraint.

It is essential that MANAGER could find such new rules: when a new problem is defined, it is often important to add such rules. They are general because the system can use them to solve any problem, but we cannot find all of them at the beginning. A good human problem solver is able to find and use such new rules when it is useful for a new problem, an intelligent system must also be able to do this. If it is unable to do this and if all of the rules are given by the user, it is still the user that has to do most of the work. Unfortunately, finding new rules such as R5 and proving them is very different from what has been done in automatic theorem proving. Personally, I found R5 before proving it because it was quite evident to me.

Another interesting example is building Magic Squares. MALICE receives the definition of a magic square as constraints expressing that the sums of the lines, columns and diagonals are the same. With this formulation, MANAGER is able to generate a combinatorial program that finds a 11x11 magic square, but it is almost running out of time. MALICE succeeds in finding the common value of the sums, but the program has to instantiate 11 unknowns before finding a contradiction; it requires a lot of backtracking is necessary so the tree is very large.

However, mathematicians have found several efficient algorithms for generating magic squares of size N. More than three hundred years ago an algorithm for generating magic squares for odd N has been found by the Belgian canon Poignand and improved by the French mathematician La Hire. It generates magic squares for N odd and greater than or equal to 5, although it is not able to generate all of the possible magic squares. In fact, it finds solutions to the more constrained family of problems MAGICYL where the board is cylindrical: 2*N diagonals must have the same value instead of only 2. I have implemented this algorithm into a C program which only has 25 lines. In 8,382 seconds, it has generated (but not printed) a magic square for N=500,001, so it has found a bijection of the set of integers from 1 to 250,001,000,001 into itself that satisfies 1,000,003 constraints (and even the 2,000,004 constraints of the MAGICYL problem). Thus, studying a problem and using mathematical results can improve the performance considerably. MANAGER is completely unable to generate algorithms such as Poignand's.

Reformulating the definition of a problem and theorem proving are certainly the most efficient ways to improve the performance of a problem solver; this is why AI scientists spend a great amount of time trying to improve their systems. Certainly, it is also interesting to find a better way to remove some leaves in the tree or to generate the useful constraints more quickly. But that cannot bring forth a discontinuity as significant as what occurs when a new rule is added or when a new specific algorithm is generated. Unfortunately, MANAGER is able to perform such tasks in a very restricted manner; it is essential to improve its possibilities in these directions. However the problem of finding a good formulation of a problem is related to the possibility of defining it in a natural language: it is much easier to find a good formulation from the initial definition of the problem than from a formulation in a language such as ALICE where many choices have already been made. For instance, some auxiliary correspondences may have been added, such as the carries in a crypt-addition problem; if it is a poor choice, it may be difficult to go back to the original formulation in natural language. To avoid the difficulties which come from natural languages, it would be interesting to define an artificial language where the problems would be stated in a similar way as in the natural language, but with simple grammar and without the ambiguities that complicate the processing of natural language texts.

13. Conclusion

CAIA is not able to carry on all of the activities of a human AI scientist. MANAGER is the agent whose activities are similar to those of a human AI scientist; we have seen that it can perform several of these tasks satisfactorily although many others are still beyond its scope. However, it developed some decent results: the general problem solving system that it generated found elegant solutions for many constraint satisfaction problems and it can write an efficient combinatorial program for solving problems when the combinatorial approach is the best one. It can also inspect the formulation of a problem and find results that will improve the resolution considerably. Besides conducting experiments, it analyzes their results and finds some defects that would be interesting to modify. Unfortunately, it is not able to use these results to ameliorate itself, but they are very useful for a human AI scientist. For instance, knowing the good choices of unknowns for backtracking would be useful to improve these choices. In the same way, finding out that some good choices were missed by a module is helpful to improve this module as well. Finding out where is the bug is the first step correcting it. Finding an elegant solution after wasting a whole lot of computer time could enable the system to find the same solution within a more reasonable time. The experiments made by CAIA also helped me to improve the system. I am bootstrapping it and the collaboration between CAIA and myself was very productive: I was unable to choose and analyze all of the experiments made by CAIA but I was able to use its results to improve the system in a way that CAIA was not capable of doing. Finally, the autonomy of the system is satisfactory since it is still able to find interesting results after 3,000,000 seconds, this is more than one month.

It would be relatively easy to improve the performance of the system with better learning capabilities. However, I do not want to develop the system in that way: why try to run the computer 10 times faster when it would be possible to run it millions of times faster by modifying the formulation of the problem drastically? This is one of the directions we have considered in the preceding section: using theorem proving methods in order to change the formulation of the problems. It has been a long time since AI scientists [1,29] and more recently [8] have been convinced of the importance of reformulating problems; usually, this is done by an AI scientist and much of the intelligence used for solving a problem is human. CAIA needs to be a better mathematician so that it can modify the formulation of the problems, define new rules, and find new methods to simplify mathematical expressions.

An effective improvement would be to define MANAGER's knowledge in the same form as MALICE's knowledge: MANAGER' behavior would also be supervised by using rules and methods. It would be easier to define and modify this knowledge, and we could give MANAGER the possibility to change some parts of its own triggers. This is certainly a promising direction: the events must be defined (which is not very difficult) as well as the rules (which is not easy to do).

The far-reaching goal of creating a complete artificial AI scientist has not been fulfilled and there will be many years before we, AI scientists, will succeed. However, the results were much better than what I expected. Using powerful computers, it is feasible to perform a large meta-combinatorial search on the rules that can be used to solve a problem; thus, a general system may be able to find elegant solutions in a reasonable time. Moreover, a human scientist does not have the time required to analyze all of the results found by its system so much in depth as an artificial scientist because he does not have the time to launch and analyze many experiments. With all its limitations, CAIA performed better choices than myself in several situations; as a result, it created a general problem solver and wrote combinatorial programs whose results were comparable to those of the general problem solver that I had generated. The quality of CAIA's performance definitely surprised me.

In conclusion, when CAIA is able to perform a task, it is rather a little better than myself; unfortunately, there are many activities that it has not been able to perform completely. However, for some of these activities, its partial results have already been very helpful for a human scientist.

Acknowledgments

I would like to thank Jean-Louis Laurière, Jean-François Lucas, Pierre-Yves Lucas, Tristan Pannérec, Basile Starynkevitch, Gérard Tisseau, and Carla for their useful comments on preliminary drafts of this paper. I am indebted to Francis Sourd who experimented two programs with the magic cube problem. I gratefully acknowledge an anonymous reviewer for his many detailed and excellent suggestions that helped to improve the paper significantly.

References

[1] S. Amarel, On representation of problems of reasoning about actions, in D. Michie (Ed.), Machine Intelligence 3, Edinburgh University Press, 1968, pp. 131-171.

[2] D. Bobrow, A question-answering system for high school algebra word problems, AFIPS Conference Proceedings, Vol. 26, Fall Joint Computer Conference, Spartan Books, 1964.

[3] B. Buchanan, G. Livingston, Toward Automated Discovery in the Biological Sciences, AI Magazine 25, Number 1, 2004, pp. 69-84.

[4] S. Colton, A. Bundy, T. Walsh, Automatic Concept Formation in Pure Mathematics, Proceedings of the 16th IJCAI, 1999, pp. 786-791.

[5] S. Colton, Automated theory Formation in Pure Mathematics, Springer, 2002.

[6] J. Crawford, M. Ginsberg, E. Luks, A. Roy, Symmetry Breaking Predicates for Search Problems, Proceedings of KR'96, pp. 148-159.

[7] R. Davis, D. Lenat, Knowledge-Based Systems in Artificial Intelligence, Mc-Graw Hill, 1982.

[8] T. Ellman, F. Giunchiglia, Introduction to the Special Volume on Reformulation, Artificial Intelligence 162, 2005, pp. 3-5.

[9] M. Fujita, J. Slaney, F. Benett, Automatic Generation of Some Results in Finite Algebra, in Proceedings of the thirteenth IJCAI, Chambéry, 1993, pp. 52-57.

[6] H. Gelernter, A note on syntactic symmetry and the manipulation of formal systems by machine, Information and Control 2, 1959, pp. 80-89.

[10] A. Junghanns, J. Schaeffer, Sokoban: Enhancing general single-agent search methods using domain knowledge, Artificial Intelligence 129, 2001, pp. 219-251.

[11] S. Lahlou, Attracteurs cognitifs et travail de bureau, Intellectica Nr.30, 2000, pp. 75-113.

[12] J. Laird, A. Newell, P. Rosenbloom, SOAR: An architecture for general intelligence, Artificial Intelligence 33, 1987, pp. 1-64.

[13] P. Langley, H. Simon, G. Bradshaw, J. Zytkow, Scientific Discovery, The MIT Press, 1987.

[14] J.-L. Laurière, Un langage et un programme pour énoncer et résoudre des problèmes combinatoires, Thèse de l'Université Paris 6, 1976.

[15] J.-L. Laurière, A language and a program for stating and solving combinatorial problems, Artificial Intelligence 10 (1978) pp. 29-127.

[16] J.-L. Laurière, Toward efficiency through generality, in: Proceedings of the sixth IJCAI, 1979, pp. 619-621.

[17] J.-L. Laurière, Propagation de contraintes ou programmation automatique?, Rapport LAFORIA 96/19, Université Paris 8, (1996).

[18] D. Lenat, The nature of Heuristics, Artificial Intelligence 19, 1982, pp. 189-249.

[19] D. Lenat, Theory formation by heuristic search, Artificial Intelligence 21, 1983, pp. 31-59.

[20] D. Lenat, EURISKO: A program that learns new heuristics and domain concepts, Artificial Intelligence 21, 1983, pp. 61-98.

[21] M. Lopez Laiseca, Compréhension de casse-tête en langage naturel. Traduction dans le langage ALICE, Thèse de l'Université Paris 6, 1980.

[22] J.-Y. Lucas, Génération automatique de programmes par règles et compilation de bases de règles, Thèse de l'Université Paris 6, 1989.

[23] J. Lukasiewicz, Aristotle's syllogistic, Oxford University Press, 1958.

[24] P. Meseguer, C. Torras, Exploiting Symmetries within constraint satisfaction search, Artificial Intelligence 129, 2001, pp. 133-163.

[25] M. Minsky, The Society of Mind, Simon and Schuster, 1986.

[26] M. Minsky, P. Singh, A. Sloman, The St. Thomas Common Sense Symposium: Designing Architectures for Human-Level Intelligence, AI Magazine 25, Number 2, 2004, pp. 113-124.

[27] S. Minton, J. Carbonell, C. Knowblock, D. Kuokka, O. Etzioni, Y. Gil, Explanation-based learning: A problem solving perspective, Artificial Intelligence 40, 1989, pp. 63-118.

[28] A. Newell, J. Shaw, H. Simon, A variety of intelligent learning in a General Problem Solver, in: M. Yovits, S. Cameron (Eds.) Self Organizing Systems, Pergamon Press, 1960, pp. 153-189.

[29] A. Newell, Limitations on the current stock of ideas about problem-solving, in: A. Kent, O. Taulbee (Eds), Electronic Information Handling, Spartan Books, 1965, pp. 195-208.

[30] S. Pinson, Credit Risk Assessment and Meta-judgment, Theory and Decision 27, 1989, pp. 117-133.

[31] J. Pitrat, Realization of a program which the theorems he proves, in: Proceeedings of the IFIP Congress 65, New York, 1965.

[32] J. Pitrat, Réalisation de programmes de démonstration de théorèmes utilisant des méthodes heuristiques, Thèse de l'Univesité de Paris, 1966.

[33] J. Pitrat, Heuristic interest of using meta-theorems, in Lectures notes in mathematics 125, Springer Verlag, 1970, pp. 194-206.

[34] J. Pitrat, A program for learning to play chess, in Pattern recognition and artificial intelligence, Chen ed., Academic Press, 1976, pp. 399-419.

[35] J. Pitrat, Implementation of a reflective system, Future Generation Computer systems12, 1996, pp. 235-242.

[36] J. Pitrat, Meta-explanation in a Constraint Satisfaction Solver, in: Information Processing and Management of Uncertainty in Knowledge-based Systems IPMU, 2006, pp. 1118-1125.

[37] J.-F. Puget, Breaking symmetries in all different problems, Proceedings of IJCAI'05, 2005, pp. 272-277.

[38] http://mathoeuf.free.fr/lg/public/index.php3

[39] http://www.csplib.org/