Nogood Recording for Static and Dynamic Constraint Satisfaction Problems*

Thomas Schiex† Gérard Verfaillie‡

C.E.R.T. – O.N.E.R.A. (France)

Abstract

Many AI synthesis problems such as planning, scheduling or design may be encoded in a constraint satisfaction problem (CSP). A CSP is typically defined as the problem of finding any consistent labeling for a fixed set of variables satisfying all given constraints between these variables. However, for many real tasks, the set of constraints to consider may evolve because of the environment or because of user interactions. The problem we consider here is the solution maintenance problem in such a dynamic CSP *(DCSP). We propose a new class of constraint recording algorithms called* Nogood Recording *that may be used for solving both* static *and* dynamic *CSPs. It offers an interesting compromise, polynomially bounded in space, between an ATMS-like approach and the usual static constraint satisfaction algorithms.*

1 Introduction

The constraint satisfaction problem (CSP) model is widely used to represent and solve various AI related problems and provides fundamental tools in areas such as truth maintenance, expert systems or constraint logic programming . It offers a simple, natural and yet powerful knowledge representation framework and various algorithms for solving various requests (the most usual being satisfaction, a NP-hard problem).

Nevertheless, many problems in Artificial Intelligence involve reasoning in dynamic environments: the knowledge is not defined once and for all, but may incrementally evolve between successive requests. The problem considered here is the *solution* maintenance problem when the underlying knowledge is embedded in a constraint network [1]. In practice, this problem may appear when the knowledge represented by a CSP may be modified either by the user (interactive problem solving), by another process (distributed problem solving) or by an external disturbance (unexpected event). In each of these applications, the problem may be reduced to the maintenance of a solution when a constraint is either added or suppressed from the CSP.

Most existing satisfaction algorithms are written for static CSPs. So, if we add or retract constraints in a CSP and try to find a new solution after each modification with any of these algorithms, we will probably repeat most of the work at each satisfaction. Another approach, embodied in the ATMS and related to k-consistency enforcing in [2], is to store, in a "concise" way, the whole frontier of the solution space explored along with justifications for its existence. However, the space needed may grow exponentially with the size of the problem, the complexity is far beyond satisfaction complexity [3] and the services offered are beyond what is needed. We propose a class of algorithms, built on top of existing satisfaction algorithms, that both solve the static satisfaction problem more efficiently and compute an *approximate description* of the frontier of the space explored, justified in terms of set of constraints, which will be used for satisfaction on future problems, thus outperforming existing static algorithms for the solution maintenance problem.

The paper is organized as follows: section 2 presents the CSP model, usual techniques and defines the dynamic constraint satisfaction problem (DCSP). Section 3 presents our *Nogood Recording* scheme and how it may be built on top of the *Backtrack* and *Forward-Checking* algorithms. Various limitations and extensions to the scheme are proposed. Section 4 deals with experimental results and is followed by some final remarks.

2 Definitions

2.1 The CSP model

Definition 2.1 *A constraint satisfaction problem* (X, C) *involves a set* $X = \{x_1, \ldots, x_n\}$ *of* n *variables and a set* C *of constraints. Each variable* $x_i \in X$ *takes its value in its finite domain* $d(x_i)$*. Each constraint* $c \in C$ *involves a subset* $X_c = \{x_{c_1}, \ldots, x_{c_{a_k}}\}$ *of* X *and is defined by a subset* R_c *of* $d(x_{c_1}) \times \cdots \times d(x_{c_{a_k}})$ *which specifies which values of the variables are compatible with each other.*

*This work has been supported by the "Direction des Recherches et Études Techniques" of the French "Délégation Générale de l'Armement" under contract 89.002.109.

†e-mail address: schiex@cert.fr

‡e-mail address: verfail@cert.fr

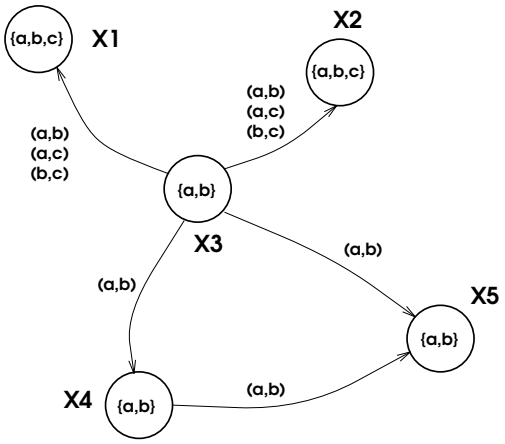

Figure 1: An example of binary CSP

A CSP may be associated with its constraint hyper-graph in which nodes represent variables and hyper-arcs represent constraints. A binary CSP is a CSP whose constraints are binary *i.e.*, involve at most two variables. The methods proposed in this paper are not restricted to binary CSPs.

Let us consider for instance the CSP graph presented in figure 1 (modified from [4]). Each node represents a variable whose domain is explicitly indicated, each arc is labeled with the set of tuples permitted by the constraint (note that the constraint between connected variables is a strict lexicographic order along the arrows).

Definition 2.2 *An assignment $\mathscr{A}$ of values to variables in $Y \subset X$* satisfies *a constraint c such that $X_c \subset Y$ iff the projection (noted $\mathscr{A}\downarrow X_c$) of the assignment on the constraint variables belongs to its associated relation R_c. An assignment of values to a subset Y of the variables is* consistent *iff it satisfies all the constraints c such that $X_c \subset Y$. A* solution *is a consistent assignment of values to all the CSP variables. A CSP is said to be* consistent *if it has (at least) one solution.*

The usual task in a CSP is to find one solution. This problem is NP-hard. It may be solved using the *Backtrack* algorithm that provisionally assigns consistent values to a subset of variables and attempts to append to it a new instantiation such that the whole set is consistent.

The order by which variables get instantiated has usually an important effect on the efficiency of the algorithm [5] since each ordering defines how soon a constraint may be checked. The ordering used may be either pre-computed or may vary dynamically during the execution. The methods proposed in this paper are not dependent on a particular ordering scheme and we assume, without loss of generality, that the ordering used is given as part of the problem input.

One way to improve backtrack efficiency is to make it possible for the algorithm to rapidly detect the assignments that can not participate in a solution. This may be done by adding *induced* constraints to the CSP either before or during the search.

Definition 2.3 *A constraint c is* induced *by a CSP (X, C) if it is satisfied by all the CSP solutions.*

Thus, adding an induced constraint to a CSP does not change its solution set and may enable some non globally consistent assignments to be discovered early. This is usually called constraint propagation and has been formalized in various levels of so-called local consistency: arc-consistency, path-consistency, k-consistency [4] and has been related to ATMS nogood inference in [2].

2.2 Dynamic constraint satisfaction problems

Definition 2.4 *A dynamic constraint satisfaction problem $\mathscr{P}$ is a sequence $\mathscr{P}_0 \ldots, \mathscr{P}_\alpha \ldots$ of static CSPs, each one resulting from a change in the preceding one [6]. This change may be a* restriction *(a new constraint is imposed on a subset of variables) or a* relaxation *(a constraint is removed from the CSP).*

The problem of solution maintenance in a DCSP $\mathscr{P}$ can now be defined as the problem of sequentially computing a solution for each of the CSPs $\mathscr{P}_0 \ldots, \mathscr{P}_\alpha \ldots$ Given any existing static satisfaction algorithm, a naive approach is to successively apply this algorithm to each CSP. The CSP framework being monotonous (restrictions suppress solutions, relaxations create new solutions), some obvious optimizations are relevant:

1. if an assignment $\mathscr{A}$ is a solution of $\mathscr{P}_\alpha = (X, C_\alpha)$ then it is also a solution of $(X, C_\alpha - c)$;
2. conversely, if an assignment $\mathscr{A}$ is not a solution of $\mathscr{P}_\alpha = (X, C_\alpha)$ then it is not a solution of $(X, C_\alpha + c)$ either. Stated otherwise: any constraint c' induced by $\mathscr{P}_\alpha$ is also induced by $(X, C_\alpha + c)$;
3. finally, if the CSP (X, C_α) is inconsistent, then the constraint $\neg c$[1] is induced by the CSP $(X, C_\alpha - c)$ and thus may be added to $(X, C_\alpha - c)$ prior to search.

Most of these simple remarks have been used in existing proposals for the solution maintenance problem: the first remark is used by [7] in a local changes static satisfaction algorithm applied to the previous solution if any. Remarks 1 and 2 are used by [8] oracles scheme: using a static order on variables and values, the space explored by a tree search algorithm is totally and strictly ordered and the space explored before the occurrence of a *first solution* may be

[1] The constraint such that $X_{\neg c} = X_c$ and $R_{\neg c}$ is the complementary of R_c in $\Pi_{x \in X_c} d(x)$.

concisely described by the *first solution itself*. However, this scheme is difficult to apply when repeated relaxations and restrictions occur: after a relaxation, the solution given by 1 may be not the *first solution* of the new problem, and therefore, the scheme 2 may not be applied if a restriction occurs afterwards.

Other related approaches, inspired from Truth Maintenance Systems, solve the arc-consistency maintenance problem [6, 9]: unary induced constraints are computed and justified by constraint set(s). These approaches naturally exploit CSP monotonicity but *do not solve* the satisfaction problem.

3 A DCSP algorithm

Basic backtrack tree search has been enhanced by various mechanisms: backmarking, backjumping, forward checking... Our aim is to enhance backtrack by a mechanism that will, during the exploration, build an approximate (polynomially bounded in space) description of the frontier of the space explored along with justifications relating the frontier to the CSP constraints.

3.1 Preliminaries

The objects that we will consider throughout this paper will be called *nogoods*:

Definition 3.1 *A* nogood *is a pair $(\mathscr{A}, J)$, where $\mathscr{A}$ is an assignment of values to a subset of the CSP variables and $J \subset C$ such that no solution of the CSP (X, J) contains $\mathscr{A}$. Stated otherwise, the constraint forbidding $\mathscr{A}$ is induced by the CSP (X, J). The set J is called the nogood justification.*

In the framework of an ATMS, an assumption being created for each constraint and each single variable-value assignment, a nogood corresponds to an inconsistent ATMS environment. Knowing every nogood of a given CSP (X, C) would be extremely interesting, both in the static and the dynamic case since it would enable every assignment that cannot participate in a solution to be discovered for any CSP (X, C') when $C' \subset C$, so as some of these when $C \subset C'$. But the number of such nogoods may grow exponentially with the number of constraints and variables. Using the inclusion relation, we may order our nogoods, thus giving an opportunity to reduce the spatial complexity:

Theorem 3.1 *Let $(\mathscr{A}, J)$ be a nogood, and $\mathscr{A}'$ J' be such that $J \subset J'$ and $\mathscr{A} \subset \mathscr{A}'$. Then $(\mathscr{A}', J')$ is a nogood. This will be noted $(\mathscr{A}, J) \prec (\mathscr{A}', J')$.*

Proof*: The solution set of (X, J') being included in the solution set of (X, J), if $\mathscr{A}$ is not included in any (X, J) solution, no superset of $\mathscr{A}$ can be included in any (X, J') solution.*□

Naturally, $\prec$-minimal nogoods are preferred, since they implicitly define more nogoods (according to 3.1). However, there is still a possibly exponential number of such nogoods and we will therefore restrict ourselves to a non-complete non-minimal nogood computation during the tree search. The following general properties will be used:

Theorem 3.2 *Let $(\mathscr{A}, J)$ be a nogood, $X_J = \cup_{j \in J} X_j$, $\mathscr{A}{\downarrow}X_J$ be the projection of $\mathscr{A}$ on X_J, then $(\mathscr{A}{\downarrow}X_J, J)$ is a nogood.*

Proof*: let us suppose that $(\mathscr{A}{\downarrow}X_J, J)$ is not a nogood. This means that there is a solution s of the CSP (X, J) which is a superset of $\mathscr{A}{\downarrow}X_J$. Let us consider the assignment s', derived from s by assigning all the variables in $X_{\mathscr{A}} - X_J$ to their values in $\mathscr{A}$. s' is a solution of (X, J) since $s'{\downarrow}X_J = s{\downarrow}X_J$, moreover, s' contains $\mathscr{A}$. This is in contradiction with $(\mathscr{A}, J)$ being a nogood.*□

This theorem allows us to build a better nogood each time a new nogood is known. A last property will allow us to build another nogood from a well-formed collection of nogoods and an induced constraint:

Theorem 3.3 *Let $\mathscr{A}$ be an assignment of variables in $X_{\mathscr{A}}$, $x_s \notin X_{\mathscr{A}}$ be a variable, c be an unary constraint on x_s induced by (X, J) and $\mathscr{A}_1, \ldots \mathscr{A}_m$ be various assignments extending $\mathscr{A}$ on x_s. If both $R_c \subset \cup_{1 \leq i \leq m} \mathscr{A}_i{\downarrow}\{x_s\}$ and $(\mathscr{A}_1, J_1), \ldots (\mathscr{A}_m, J_m)$ are nogoods then $(\mathscr{A}, (\cup_{1 \leq i \leq m} J_i) \cup J)$ is a nogood.*

Proof*: (1) Since c is induced by (X, J), it is also induced by $(X, \cup_i J_i \cup J)$ and any solution s of $(X, \cup_i J_i \cup J)$ satisfies c. Therefore, $s{\downarrow}\{x_s\} \in R_c \subset \cup_i \mathscr{A}_i{\downarrow}\{x_s\}$. This shows that $\exists i', 1 \leq i' \leq m$ such that $s{\downarrow}\{x_s\} = \mathscr{A}_{i'}{\downarrow}x_s$. (2) Let us suppose that $(\mathscr{A}, \cup_i J_i \cup J)$ is not a nogood. It means that there is a solution s of the CSP $(X, \cup_i J_i \cup J)$ that contains $\mathscr{A}$. Moreover, using (1), $\exists i'$ such that $s{\downarrow}\{x_s\} = \mathscr{A}_{i'}{\downarrow}x_s$. Therefore, s contains $\mathscr{A}_{i'}$. This is in contradiction with $(\mathscr{A}_{i'}, J_{i'})$ being a nogood.*□

3.2 Building nogoods during Backtrack search

Consider a CSP (X, C) solved using the usual backtrack algorithm. Each leaf of the tree explored offers an opportunity to build a nogood (by definition):

Theorem 3.4 *If an assignment $\mathscr{A}$ violates a constraint c, then $(\mathscr{A}, \{c\})$ is a nogood.*

Applying theorem 3.2 on this nogood simply brings the fact that $(\mathscr{A}{\downarrow}X_c, \{c\})$ is a nogood, a fact already represented in the CSP by the constraint c itself.

It is now possible to inductively apply a corollary of theorem 3.3 during backtrack search, starting from the nogoods built on inconsistent leaves, using constraints induced by variable domains.

Corollary 3.1 *Let $\mathscr{A}$ be an assignment of the variables in $X_{\mathscr{A}}$, x_s an unassigned variable, and $\{\mathscr{A}_1, \ldots, \mathscr{A}_m\}$ be all the possible extensions of $\mathscr{A}$ along x_s, using every value of $d(x_s)$. If $(\mathscr{A}_1, J_1), \ldots, (\mathscr{A}_m, J_m)$ are nogoods, then $(\mathscr{A}, \cup_i J_i)$ is a nogood.*

3.3 Using nogoods

Each time a nogood is built, it will be systematically enhanced using theorem 3.2. The nogood $(\mathscr{A}\downarrow X_J, J)$ obtained *may* serve three different purposes:

- it is possible to *backjump* to the lowest variable in $V_J = \cup_{j \in J} X_J$. The backjumping obtained is a dependency-directed backtracking [10], closely related to the *conflict based backjumping* algorithm independently proposed in [11].
- the constraint forbidding $\mathscr{A}\downarrow X_J$ being induced by $(X, J), J \subset C$, it is also induced by (X, C). It is therefore possible to add it to the CSP during the search, possibly pruning branches and thus improving efficiency.
- when solving a CSP (X, C'), and for every nogood $(\mathscr{A}, J)$ built such that $J \subset C'$, the constraint forbidding $\mathscr{A}$ may be added to the CSP before starting the search.

Because *induced* constraint may be added to the CSP during the search, a constraint violation may be an induced constraint violation. The theorem 3.4 may be simply extended to handle this:

Theorem 3.5 *If an assignment $\mathscr{A}$ violates a constraint c belonging to C then $(\mathscr{A}, \{c\})$ is a nogood, otherwise if c had previously been added because of a nogood $(\mathscr{A}', J)$, then $(\mathscr{A}, J)$ is a nogood.*

This extension is a direct consequence of theorem 3.1. The following pseudo-code defines the algorithm more precisely. Such as it is defined, it will find the first solution. The function **check** tests whether a constraint is violated by an assignment. If none, it returns the empty set. If the constraint violated is a member of C, it returns the set composed of the constraint violated itself. Otherwise, the constraint was recorded because of a nogood $(\mathscr{A}, J)$: it returns J. The function **record** simply adds the induced constraint corresponding to the nogood built to the CSP (so that **check** may work as defined above). The function **project** projects an assignment on a set of variables. The function **involved**(x, J) returns *true* iff the variable x is a member of the variable set X_c of one of the constraints c in J. The function **Nogood-Builder** should initially be called with $\mathscr{A} = \varnothing$ and $V = X$ the set of all the CSP variables.

NogoodBuilder$(\mathscr{A}, V)$
If $V = \varnothing$
  Then $\mathscr{A}$ is a solution, Stop
  Else
  Let x be a variable in V, $J = \varnothing$, *BJ=false*
  For each $\nu \in d(x)$ until *BJ*
    Let $\mathscr{A}'$ be $\mathscr{A} \cup \{x = \nu\}$, K be **check**$(\mathscr{A}')$
    If $K = \varnothing$,
      Then Let J-sons be **NogoodBuilder**$(\mathscr{A}', V - \{x\})$
      If **involved**$(x, J$-sons$)$
        Then $J \leftarrow J \cup J$-sons
        Else $J \leftarrow J$-sons, *BJ=true*
      Endif
      Else $J \leftarrow J \cup K$
    Endif
  Endfor
  If *BJ=false* then **record**(**project**$(\mathscr{A}, J), J)$ Endif
  Return J
Endif

Correctness*: the backtrack algorithm being complete, we may simply show that the additional pruning gained by backjumping or induced constraint suppresses only assignments that can not participate in a solution. Note that* suppressing or limitating any of these features *cannot endanger this result.*

We will suppose that the algorithm is applied on a CSP (X, C) augmented with induced constraints forbidding $\mathscr{A}$ (added because of previously built nogoods $(\mathscr{A}, J), J \subset C$).

- *a first point is that inductively using theorems 3.5 and 3.2 and corollary 3.1 will only build nogoods. The assignments unexplored because of the violation of a constraint forbidding $\mathscr{A}$, added because of a nogood $(\mathscr{A}, J)$ cannot participate in a solution (by definition).*
- *at a given node, since it has been inferred that the constraint forbidding the assignment $\mathscr{A}\downarrow X_J$ is induced by $J \subset C$, any assignment containing $\mathscr{A}\downarrow X_J$ can not lead to a solution. These are precisely the assignments whose exploration will be avoided by backjumping.* □

3.4 An example

Let us consider again the problem in figure 1. Figure 2 illustrates the tree explored by the *Backtrack* and *NogoodBuilder* algorithms using the variable ordering $(x_1, x_2, x_3, x_4, x_5)$. All the arcs are explored by *backtrack*, a total of 48 nodes and 45 constraint checks. The *NogoodBuilder* algorithm only explores "thick" arcs, leading to a total of 24 nodes and 21 constraint checks. Nogoods built are indicated on the left-hand side, a dotted arrow giving their "birth place".

Let us see how the first nogood is built: the assignment $\mathscr{A}_1 = \{x_1 = a, x_2 = a, x_3 = a\}$ is inconsistent because it violates (in the order constraints are checked) the constraint c_{32}: $(\mathscr{A}_1, c_{32})$ is a nogood. In the next state, the assignment $\mathscr{A}_2 = \{x_1 = a, x_2 = a, x_3 = b\}$ violates the same constraint: $(\mathscr{A}_2, c_{32})$ is also a nogood. None of these will be

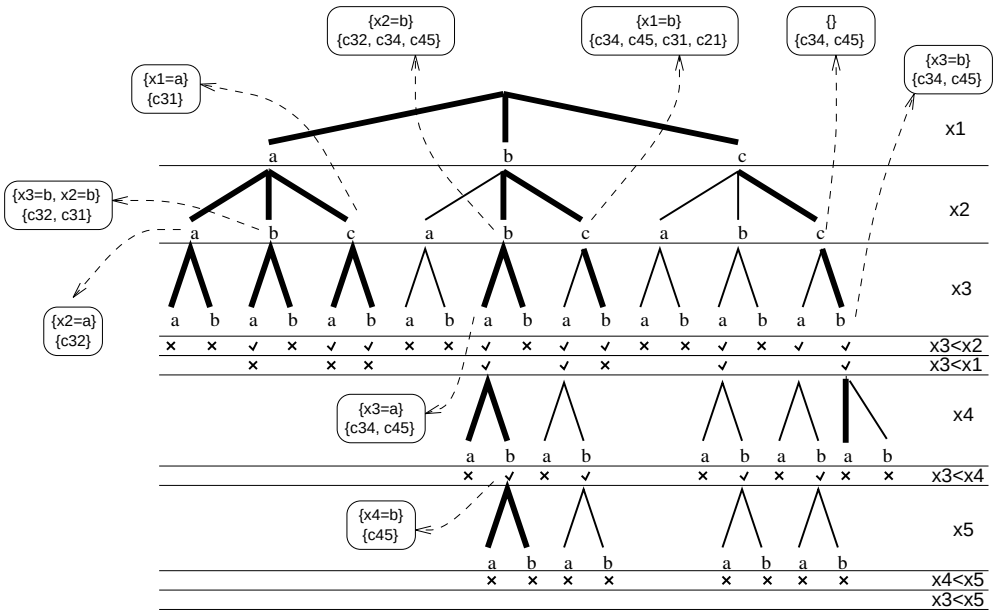

Figure 2: *Backtrack* and *Nogood recording*

added to the CSP since the information they hold is already contained in the constraint c_{32}. But it is possible to apply corollary 3.1 and infer that $(\{x_1 = a, x_2 = a\}, c_{32})$ is a nogood. Using theorem 3.2, we finally get that $(x_2 = a, c_{32})$ is a nogood. It is immediately incorporated into the CSP to prune the tree later. One may note that the unusual empty 0-ary constraint is induced last. This constraint (the CSP equivalent of the "empty clause" in propositional logic) simply states that the CSP is inconsistent and its justification is a (here minimal) justification of the CSP inconsistency. On the same problem, using the same ordering, the *full shallow learning* algorithm [12] explores 35 nodes and performs 49 constraint checks.

3.5 Practical limitations

If one nogood is generated at each node of the tree explored, the number of nogoods that may be recorded is tremendous. We must therefore limit ourselves to a subset of the nogoods. The most interesting nogoods are defined by the $\prec$ order. To get a polynomial bound on the space used for nogoods recording, we will limit the recording to nogoods $(\mathscr{A}, J)$ such that $|\mathscr{A}| \leq i$, where i is a constant. This will be called i^{th} order nogood recording. Note that the only modifications imposed on the previous algorithm are in the function **record** and that the correctness sketch of proof is still valid. There are only $O(n^i)$ constraints of arity i or less, and for each one $O(d^i)$ assignments of length i or less. Furthermore, recording and backjumping make certain that we won't learn two nogoods $(\mathscr{A}, J)$ and $(\mathscr{A}, J')$ such that both $J \subset C$ and $J' \subset C$. Therefore, when solving a dynamic CSP, if every nogood $(\mathscr{B}, J'')$ such that $J'' \not\subset C$ is discarded from the recorded nogood list after each relaxation, the space used by recorded nogoods is polynomially bounded.

Another possible restriction is to record only nogoods $(\mathscr{A}, J)$ such that a constraint already exists between the variables of $\mathscr{A}$. This makes certain that the CSP hypergraph topology won't be modified, which may be a desirable property. This will be called topology-directed recording. It may be mixed with the i^{th} order restriction.

The algorithm defined by i^{th} order nogood recording is not easily compared with other existing approaches since the task accomplished is not "denotationally" well defined. Nevertheless, the following properties give some information on the task achieved using the i^{th} order restriction:

Theorem 3.6 *Using a static vertical ordering, the nogoods recorded by the i^{th} order nogood recording algorithm on the k^{th} variable form a subset of what would have been inferred by strong $(i, n-k)$-consistency enforcing. Furthermore, it is possible that some of these nogoods would not have been inferred by strong $(i, n-k-1)$-consistency enforcing [13].*

Proof*: Let $(\mathscr{A}\downarrow X_J, J)$ be a nogood built at level k ($\mathscr{A}$ is an assignment on $x_1, \ldots, x_k$). We know that $\mathscr{A}$ may not be extended to a full solution by assigning values to variables $(x_{k+1}, \ldots, x_n)$. By theorem 3.2, this is also true for $\mathscr{A}\downarrow X_J$ which is therefore not $(|\mathscr{A}\downarrow X_J|, n-k)$-consistent and $|\mathscr{A}\downarrow X_J| \leq i$. The second property will be justified by an example: consider a CSP with n variables, domains all equal to the first $n-1$ integers and a complete graph of $\neq$ constraints. It is both strongly $(0, n-1)$-consistent*

*but inconsistent (*i.e.*, non $(0, n)$-consistent). It may be shown that the 0^{th} order nogood recording algorithm will build and record one nogood $(\varnothing, J')$ stating the non $(0, n)$-consistency.*□

This gives an upper bound and a lower bound on the task possibly achieved. Furthermore, the algorithm solves the satisfaction problem. It is now possible to bring to light the main differences between our approach and the usual constraint recording approach [12]:

1. constraint recording algorithms do not *justify* constraints learned. This makes the schemes useless when relaxation occurs.

2. constraint recording algorithms only learn (and therefore backjump) in *dead-end* situations, when the current assignment can not be extended to a consistent assignment on the *next* variable. Our algorithm learns whenever an assignment can not be extended to a consistent assignment on the *complete set* of n CSP variables (this includes *dead-end* situations). This is why constraint recording, when limited to the i^{th} order, only enforces a subset of strong $(i, 1)$-consistency.

3. the *full shallow learning* algorithm builds an assignment which is always a *superset* of the assignment $\mathscr{A}\downarrow X_J$ built by our algorithm. Not only does this mean worse nogoods, but also worse backjumping at comparable costs.

As for the ATMS, if an assumption is associated to each constraint and each (variable-value) pair (using the encoding proposed in [2]), it would achieve strong n-consistency and would give all (a possibly exponentially growing number of) $\prec$-minimal nogoods. Some TMS limit the nogood inference rule to weaker propagation, usually arc-consistency).The task is easier, but we are in fact brought back to arc-consistency maintenance as proposed in [6, 9]. This is far from satisfaction, but could be useful for preprocessing however.

3.6 Extension to forward-checking

The algorithm previously described simply extends the usual *backtrack* tree search algorithm using theorems 3.4 (producing nogoods), corollary 3.1 (merging nogoods) and theorem 3.2 (improving nogoods). Using the *Forward-Checking* algorithm, we may:

- *build nogoods*: each time a constraint is forward-checked *and* suppresses value(s) from a variable domain, it will be noted as a *value killer* of the domain. When a domain becomes empty, we may build a nogood composed of the current assignment and the set of all the *value killers* of this domain.

- *merge nogoods*: Let K be the set of *value killers* of the variable, then $(\cup_i J_i) \cup K$ (where the J_i are the constraints sets in the nogoods $(\mathscr{A}_i, J_i)$ built for the sons) may form a nogood when paired with the current assignment (see theorem 3.3).

- *improve nogoods*: theorem 3.2 directly applies.

4 Experimental results

The algorithms considered in this section are the nogood recording algorithms limited by order 0, 1 and 2, with or without topology directed restriction for order 2 and relying on both *Backtrack* and *Forward-checking* (defining eight algorithms named NR_0, NR_1, NR_2, NR_{2T}, NR-$FC_0 \ldots$), the *Backtrack* algorithm (BT), the *Forward-Checking* algorithm (FC), the order 1 and 2 full shallow learning algorithms (FSL_1 and FSL_2) [12].

The first experimentation will try to estimate the price paid for building and recording nogoods and the amount of pruning obtained. The Zebra problem (see [14]) has been solved using 50 random vertical orderings. No heuristics has been used. For each ordering, the problem is solved twice (using induced constraints if any): the percentage avoided in the new satisfaction (100 % would mean "new satisfaction for free") is given in the "Saved" column. For each measure, rounded mean (μ) and standard deviation (σ) values are given.

Usually, consistency checks performed are considered as a good measure of the search effort. The algorithms are ranked according to this criterium in each table. FSL algorithms bring almost an order of magnitude when compared to BT, but seem to perform relatively poor learning with a saving of about 20 % only. The NR algorithms give almost another order of magnitude in speedup and a learning that can lead to more than 80 % of savings. Nogood recording

Alg.	μ	σ	Saved
BT	5 300 000	7 150 000	0 %
FSL_1	930 000	993 000	18 %
FSL_2	930 000	1 130 000	19 %
NR_0	375 000	867 000	0 %
NR_1	181 000	274 000	50 %
NR_{2T}	173 000	271 000	53 %
NR_2	161 000	256 000	86 %
FC	40 400	34 300	0 %
NR-FC_0	27 600	27 900	0 %
NR-FC_1	27 100	27 700	53 %
NR-FC_{2T}	26 800	27 700	56 %
NR-FC_2	25 300	26 900	83 %

Table 1: Consistency checks

brings much less to FC than it brings to BT. This may be explained by the fact that the space explored by FC is smaller,

and give fewer opportunities to learn. However, this does not seem to lessen the pruning when a new satisfaction is performed. The number of nodes visited gives comparable results. It should be noted that topology directed learning doesn't seem to be interesting (at least for this problem). We

Alg.	μ	Saved	checks/sec.
BT	620.6	0 %	8 650
FSL_1	153.3	18 %	6 060
FSL_2	148.7	18 %	6 160
NR_0	56.1	0 %	6 590
NR_1	26.6	51 %	6 590
NR_{2T}	25.5	54 %	6 590
NR_2	23.3	86 %	6 700
FC	8.5	0 %	4 850
NR-FC_0	8.5	0 %	3 580
NR-FC_1	8.4	52 %	3 540
NR-FC_{2T}	8.3	55 %	3 540
NR-FC_2	7.9	84 %	3 520

Table 2: Time and checks per second

also measured run times. These measures are much more difficult to interpret since they depend very much on the qualities of the implementations and it is obviously simpler to code a high quality BT than a high quality sophisticated NR-FC_2. Moreover, further improvement could also be obtained by not maintaining justifications (this is a static problem). However, the timings show that the overhead for building and recording nogoods is not only reasonable but balanced by the improvement obtained.

The next experimentation aims at evaluating the impact of nogood recording after a non trivial relaxation or restriction (relaxing an inconsistent CSP and restricting a consistent CSP) on various random binary CSPs generated along 2 parameters: probability Π of existence of a tuple in the constraints and percentage P of number of constraints between tree-like topology and complete graph topology. The two parameters range from 0.1 to 0.9 by 0.2 steps, 10 CSP are built at each point (a total of 250 CSPs).

We restricted the algorithm set to FSL_1, FSL_2, NR_1, NR_2, FC, NR-FC_1 and NR-FC_2. All the CSPs have 16 variables and domains of size 9 (to get a reasonable ratio between consistent and inconsistent CSP). For each CSP, a first satisfaction is done, a randomly chosen constraint is either added or suppressed according to the CSP satisfiability. Two satisfactions of this new problem are performed, respectively using (or not) every learned information that is still valid[2]. No ordering heuristics is used[3]. Table 3 gives the number of consistency checks for the first satisfaction and to what extent contraint recording globally enhanced next satisfaction. A small $+, -$ or $\pm$ indicates whether the samples for given (Π, P) were all consistent $(+)$, all inconsistent $(-)$ or sometimes consistent, sometimes inconsistent $(\pm)$. The hardest problems occur at the transition between inconsistent and consistent CSP. A similar remark was made for various NP-hard problems in [16]. It is on these problems that nogood recording is especially useful. As previously, NR and NR-FC algorithms outperform the FSL algorithms both in number of consistency checks and in the improvement observed when learned information is used. Second order recording seems to be the best choice. However, nothing is learned (either by FSL or NR) on highly consistent (easy) problems. Run-time measures were very well correlated with consistency checks measures.

[2] FSL algorithms give valid informations only for restrictions.

[3] There are obvious vertical and horizontal ordering heuristics for dynamic CSP: (1) the values that appeared in the previous solution (if any) should be used first, or perhaps even better: the *min-conflict* heuristics [15] should be used starting from the previous solution). (2) Upon restriction, consider the variables of the constraint added last. . .

5 Conclusion

We have defined the maintenance solution problem in dynamic constraint satisfaction problems and underlined that the iterative application of usual satisfaction algorithms would result in redundant search and inefficiency and that a complete description of the space explored justified in terms of set of constraints may grow exponentially in space.

The class of nogood recording algorithm we propose solves the satisfaction problem and simultaneously offers a polynomially bounded space compromise between both of these approaches. They outperform all algorithms considered for the solution maintenance problem in dynamic CSPs, and give also very good results for static CSPs. Furthermore, they give some basic information that partially answers inconsistency explanation problems.

References

[1] Thomas Schiex and Gérard Verfaillie, "Two approaches to the solution maintenance problem in dynamic constraint satisfaction problems", *in Proc. of the IJCAI-93/SIGMAN Workshop on Knowledge-based Production Planning, Scheduling and Control*, Chambery, France, Aug. 1993.

[2] Johan de Kleer, "A comparison of ATMS and CSP techniques", *in Proc. of the 11^{th} IJCAI*, pp. 290–296, Detroit, MI, Aug. 1989.

[3] Gregory M. Provan, "The computational complexity of multiple-context truth maintenance systems", *in Proc. of the 8^{th} ECAI*, pp. 522–527, Stockholm, 1990.

[4] Alan K. Mackworth, "Consistency in networks of relations", *Artificial Intelligence*, vol. 8, pp. 99–118, 1977.

[5] Rina Dechter and Itay Meiri, "Experimental evaluation of preprocessing techniques in constraint satisfaction prob-

	P	0.1			0.3			0.5			0.7			0.9		
Alg.	Π	μ	Imp.		μ	Imp.		μ	Imp.		μ	Imp.		μ	Imp.	
FSL_1		30 100	0 %		5 320	0 %		881	0 %		1 280	0 %		289	0 %	
FSL_2		20 600	0 %		5 180	0 %		706	0 %		566	0 %		289	0 %	
NR_1	0	1 010	73 %		595	48 %		203	73 %		363	100 %		184	81 %	
NR_2	.	1 150	77 %	$-$	533	62 %	$-$	216	75 %	$-$	351	100 %	$-$	184	81 %	$-$
FC	1	1 220	0 %		220	0 %		210	0 %		210	0 %		213	0 %	
NR-FC_1		844	27 %		220	30 %		210	33 %		210	70 %		213	83 %	
NR-FC_2		851	27 %		220	30 %		210	33 %		210	70 %		213	83 %	
FSL_1		732 000	0 %		46 800	0 %		2 560	0 %		2 650	0 %		995	0 %	
FSL_2		621 000	18 %		42 700	0 %		2 410	0 %		2 590	0 %		993	0 %	
NR_1	0	71 000	56%		9 060	96 %		1 510	89 %		1 660	79 %		691	100 %	
NR_2	.	43 300	73 %	$\pm$	9 020	97 %	$-$	1 370	95 %	$-$	1 580	89 %	$-$	701	100 %	$-$
FC	3	14 200	0 %		3 240	0 %		1 080	0 %		1 170	0 %		1 030	0 %	
NR-FC_1		9 200	40 %		2 900	44 %		1 070	88 %		1 170	73 %		1 030	90 %	
NR-FC_2		8 240	68 %		2 940	47 %		1 070	88 %		1 170	73 %		1 030	90 %	
FSL_1		35 400	0 %		460 000	0 %		253 000	0 %		29 700	0 %		10 100	0 %	
FSL_2		36 400	1 %		458 000	0 %		255 000	0 %		29 700	0 %		10 100	0 %	
NR_1	0	1 280	1 %		187 000	27 %		89 700	71 %		19 100	77 %		6 650	88 %	
NR_2	.	1 240	21 %	$+$	190 000	64 %	$\pm$	85 300	75 %	$-$	18 700	93 %	$-$	6 370	92 %	$-$
FC	5	3 020	0 %		39 600	0 %		11 200	0 %		6 090	0 %		3 210	0 %	
NR-FC_1		424	0 %		33 200	31 %		10 600	80 %		6 030	73 %		3 210	78 %	
NR-FC_2		424	0 %		32 400	61 %		10 400	85 %		6 040	84 %		3 210	82 %	
FSL_1		257	0 %		38 100	0 %		16 100	0 %		602 000	0 %		1 100 000	0 %	
FSL_2		257	0 %		38 100	0 %		16 100	0 %		602 000	0 %		1 100 000	0 %	
NR_1	0	54	0 %		1 570	0 %		4 920	0 %		272 000	67 %		577 000	10 %	
NR_2	.	54	0 %	$+$	1 570	0 %	$+$	4 920	0 %	$+$	272 000	91 %	$+$	576 000	16 %	$-$
FC	7	177	0 %		289	0 %		1 150	0 %		30 520	0 %		98 000	0 %	
NR-FC_1		177	0 %		284	0 %		991	0 %		29 000	68 %		94 400	15 %	
NR-FC_2		177	0 %		284	0 %		991	0 %		29 000	91 %		64 400	34 %	
FSL_1		30	0 %		67	0 %		125	0 %		196	0 %		274	0 %	
FSL_2		30	0 %		67	0 %		125	0 %		196	0 %		274	0 %	
NR_1	0	30	0 %		67	0 %		105	0 %		175	0 %		239	0 %	
NR_2	.	30	0 %	$+$	67	0 %	$+$	105	0 %	$+$	175	0 %	$+$	239	0 %	$+$
FC	9	206	0 %		355	0 %		474	0 %		570	0 %		680	0 %	
NR-FC_1		206	0 %		255	0 %		474	0 %		570	0 %		680	0 %	
NR-FC_2		206	0 %		355	0 %		474	0 %		570	0 %		680	0 %	

Table 3: Consistency checks

lems", *in Proc. of the 11^{th} IJCAI*, pp. 271–277, Detroit, MI, Aug. 1989.

[6] Christian Bessière, "Arc-consistency in dynamic constraint satisfaction problems", *in Proc. of AAAI-91*, pp. 221–226, Anaheim, CA, 1991.

[7] Gérard Verfaillie, "Problèmes de satisfaction de contraintes : production et révision de solution par modifications locales", *in Proc. of the 13^{th} international Avignon workshop*, 1993.

[8] Pascal Van Hentenryck and Thierry Le Provost, "Incremental search in contraint logic programming", *New Generation Computing*, vol. 9, pp. 257–275, 1991.

[9] Patrick Prosser, Chris Conway, and Claude Muller, "A constraint maintenance system for the distributed allocation problem", *Intelligent Systems Engineering*, vol. 1, pp. 76–83, 1992.

[10] M. Bruynooghe and L.M. Pereira, "Deduction revision by intelligent backtracking", in J.A. Campbell, editor, *Implementation of Prolog*, chapter «Deduction revision by intelligent backtraking», pp. 194–215. Ellis Horwood, 1984.

[11] Patrick Prosser, "Hybrid algorithms for the Constraint Satisfaction Problem", *Computational Intelligence*, vol. 9, Aug. 1993.

[12] Rina Dechter, "Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition", *Artificial Intelligence*, vol. 41, pp. 273–312, 1990.

[13] Eugene C. Freuder, "Backtrack-free and backtrack-bounded search", in L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 10, pp. 343–369. Springer-Verlag, 1988.

[14] Barbara M. Smith, "How to solve the Zebra problem, or path consistency the easy way", *in Proc. of the 10^{th} European Conference on Artificial Intelligence*, pp. 36–37, Vienna, Austria, 1992.

[15] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems", *Artificial Intelligence*, vol. 58, pp. 160–205, Dec. 1992.

[16] P. Cheeseman and B. Kanefsky, "Where the *really* hard problems are", *in Proc. of the 12^{th} IJCAI*, pp. 294–299, Sidney, Australia, 1991.