# Stubbornness: a Possible Enhancement for Backjumping and Nogood Recording

**Thomas Schiex**[1]
**Gérard Verfaillie**[2]

**Abstract.** The *Conflict directed Backjumping* (CBJ) algorithm attempts to reduce the number of nodes visited within the constraint satisfaction problem by analyzing failures. The *Nogood Recording* ($NR_*$) algorithms incorporate, during the search, part of the results of these analyzes as constraints in the CSP solved itself. In both cases, *failures* are the basic information used to increase efficiency. It is shown how artificially augmenting the number of failures may lead to notable improvements in efficiency.

## Introduction

In the Constraint Satisfaction Problem (CSP) we are given a set of variables, where each variable has a finite domain, and a set of constraints, where each constraint acts between a subset of the variables. The problem is then to find an assignment of values to variables, from their respective domains, such that the constraint are satisfied. Many real world problems, such as scheduling, design or planning, can be cast as CSP's. However, the CSP framework presents some significant limitations when confronted with the demands of practical applications. Among them, the combinatorial complexity of the satisfaction problem (NP-hard) and the dynamically evolving nature of most real world problems. Among the algorithms proposed to tackle these limitations, the CBJ algorithm, proposed in [7], is an dependency directed backjumping algorithm which lead to significant improvement in efficiency w.r.t. the well known *Gaschnig Backjumping* [4] and *Graph-Based Backjumping* [3] algorithms.

The class of algorithms $NR_*$, independently proposed in [8], may be seen as CBJ enhanced with a "learning while searching" device. A significant improvement with previously described learning algorithms [3] lies in the ability to learn not only constraints but also a reason justifying why this constraint is learned (for a more precise comparison of the $NR_*$ algorithms with usual learning algorithms, see [8]). This leads to further improvement in efficiency and enable the resolution of both static CSP's and *dynamic* CSP's [1], in which the set of active constraints may evolve between successive requests.

The basic mechanism of both algorithms is to try to prove, during a backtrack tree search, and by analyzing failures, that a "small" assignment $\mathscr{A}$, that satisfies all currently checkable constraints and which is included in the current assignment, can not lead to a solution. The two algorithms then act differently: the CBJ algorithm will jump back to the lowest state such that $\mathscr{A}$ is not included in the current assignment (non chronological backtracking). The $NR_*$ algorithms will backjump as above and may furthermore add a constraint in the CSP stating that the assignment $\mathscr{A}$ is forbidden because of a given set of constraint. These justified induced constraints (called *nogoods*) avoid repeated discovery of the same inconsistency and may be used later to solve restrictions or relaxations of the CSP solved.

Since failure is the basic information that enables actual backjumping and nogood recording, one may wonder if it is possible to globally enhance efficiency by making more mistakes than it is strictly necessary ?

The paper presents possible extensions of the $NR_*$ algorithms along this general idea. The experimentations performed seem to show that additional failures may actually enhance performances, sometimes outperforming the supposedly efficient *Forward Checking* algorithm.

## 1 Definitions and conventions

**Definition 1.1** *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.*

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 [6]). 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 1.2** *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. An assignment is* globally consistent *iff it can be extended to a solution. 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. 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.

[1] e-mail address: schiex@cert.fr.
[2] e-mail address: verfail@cert.fr.


ECAI 94. *11th European Conference on Artificial Intelligence* Edited by A. Cohn
Published in 1994 by John Wiley & Sons, Ltd.

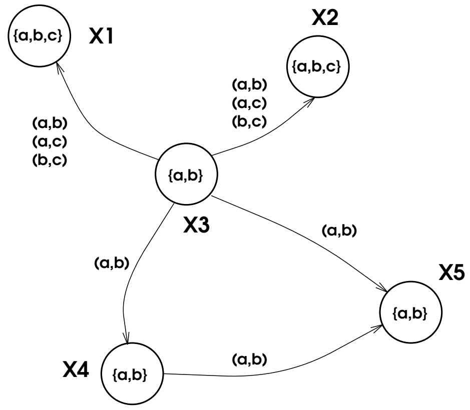


**Figure 1.** An example of binary CSP

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 1.3** *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.

## 2 The NR$_*$ algorithms

These algorithms are more precisely defined in [8], we will simply outline their main principles here.

The underlying mechanics of the NR$_*$ algorithm is the usual backtrack (BT) tree search algorithm. The main difference lies in the fact that, during tree search, and potentially at each node, three simple mechanisms will be used to build *nogoods*:

**Definition 2.1** *A* nogood *is a pair $(\mathscr{A}, J)$, where $\mathscr{A}$ is an assignment of values to a subset of the CSP variables and $J \subset C$ is 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.*

The first mechanism collects data from failures: consider a CSP $(X, C)$ solved using the usual BT algorithm. Each inconsistent leaf of the tree explored offers an opportunity to build a nogood:

**Theorem 2.1 (Basic Nogoods)** *If an assignment $\mathscr{A}$ violates a constraint c, then $(\mathscr{A}, \{c\})$ is a nogood (cf. [8]).*

Nogoods built in this way will be called *basic nogoods*. They bring no new information since the fact that $\mathscr{A}$ is inconsistent is already stated in $c$ itself, but the next mechanism will merge basic nogoods during tree search to form new nogoods, which may themselves be merged. . . possibly up to the root of the tree explored. More precisely, if all the sons of a given node have built *nogoods*, it will be possible to merge these nogoods in a new nogood with a smaller assignment. This is done *inductively* during search, as long as a nogood is available on each son of a node:

**Theorem 2.2 (Merging Nogoods)** *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 values 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 (cf. [8]).*

The last mechanism allows to build, from a given nogood $(\mathscr{A}, J)$, a new nogood $(\mathscr{A}', J)$ where $\mathscr{A}' \subset \mathscr{A}$. It will be systematically used each time a nogood is built.

**Theorem 2.3 (Projecting Nogoods)** *Let $(\mathscr{A}, J)$ a nogood, $X_J = \cup_{j\in J} X_j$, $\mathscr{A}{\downarrow}X_J$ the projection of $\mathscr{A}$ on $X_J$, then $(\mathscr{A}{\downarrow}X_J, J)$ is a nogood (cf. [8]).*

The three previous operations: creating basic nogoods (th. 2.1), projecting nogoods (th. 2.3) and merging nogoods (th. 2.2) are performed each time it is possible. When a nogood $(\mathscr{A}{\downarrow}X_J, J)$ is obtained at the node corresponding to assignment $\mathscr{A}$, it *may* serve three different purposes:

- it will be possible to *backjump* to the *lowest* variable in $V_J = \cup_{j\in J} X_J$. The backjumping obtained is a simple dependency-directed backtracking [2]. It corresponds to the backjumping performed in the CBJ algorithm;
- 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 record it in the current CSP. This will possibly prune branches during the actual search and improve *static CSP* search. Only non-basic nogoods are recorded.
- when solving a CSP $(X, C')$, and for every nogood $(\mathscr{A}, J)$ previously recorded such that $J \subset C'$, the constraint forbidding $\mathscr{A}$ may be be kept in the CSP before starting search. All other nogoods will simply be removed from the CSP and forgotten[3]

The number of nogoods built being possibly tremendous, the recording is always limited to a subset of “interesting” nogoods. The restriction performed, similar to the limitation proposed in [3], is to record only nogoods whose assignment $\mathscr{A}$ is such that $|\mathscr{A}| \leq k$, where $k$ is a given integer constant. This guarantees a polynomial bound on the number of nogoods recorded and defines $k^{th}$ order nogood recording or NR$_k$. When no information is recorded ($k = 0$), the only pruning obtained is via backjumping, and therefore NR$_0$ is closely related to CBJ.

The pseudo-code in table 1 defines the NR$_k$ algorithms more precisely. They will always find the first solution when it exists: the function **check** tests whether any 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**$_k$ compares the cardinality of the nogood assignment to the constant $k$. It is is less than $k$, it 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 $J$. Finally, the algorithm is embodied in the function **Nogood-Builder**. If the CSP is inconsistent, it always returns a non necessarily minimal justification of inconsistency (a set of constraints). It should be initially called with $\mathscr{A} = \varnothing$, the empty assignment and $V = X$ the set of all the CSP variables.

Consider again the problem in figure 1. The figure 2 illustrates the tree explored both by the *Backtrack* and the *NogoodBuilder* algorithm when using the variable ordering $(x_1, x_2, x_3, x_4, x_5)$. Each

[3] This is the *only* way a recorded nogood may actually be thrown away.

```
NogoodBuilder_k(𝒜, V)
If V = ∅
    Then 𝒜 is a solution, Stop
    Else
    Let x be a variable in V, J = ∅, BJ=false
    For each ν ∈ d(x) until BJ
        Let 𝒜' be 𝒜 ∪ {x = ν}, K be check(𝒜')
        If K = ∅,
            Then Let J-sons be NogoodBuilder(𝒜', V − {x})
            If involved(x, J-sons)
                Then J ← J ∪ J-sons
                Else J ← J-sons, BJ=true
            Endif
            Else J ← J ∪ K
        Endif
    Endfor
    If BJ=false Then
        Let Nog-A be project(𝒜, J)
        record_k(Nog-A,J)
    Endif
    Return J
Endif
```

**Table 1.** Pseudo-code for the *Nogood Recording* algorithm

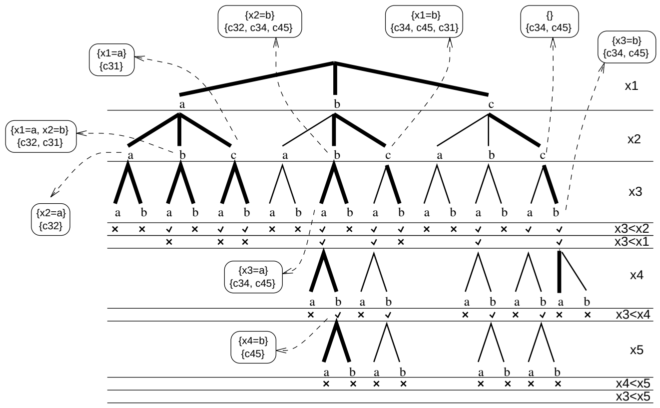


**Figure 2.** *Backtrack* and *Nogood recording*

node corresponds to an assignment. All the arcs are explored by *backtrack* while the *NogoodBuilder* algorithm only explores "thick" arcs. Nogoods built are indicated on the left side, a dotted arrow giving their "birth place".

Let us see how the first nogood is built: the state corresponding to 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 $c_{32}$ between $x_3$ and $x_2$: $(\mathscr{A}_1, c_{32})$ is a nogood. In the next state, corresponding to the assignment $\mathscr{A}_2 = \{x_1 = a, x_2 = a, x_3 = b\}$, the same constraint is violated: $(\mathscr{A}_2, c_{32})$ is also a nogood. It is now possible to apply theorem 2.2 and infer that $(\{x_1 = a, x_2 = a\}, c_{32})$ is a nogood. Using theorem 2.3, we finally get that $(x_2 = a, c_{32})$ is a nogood. It is immediately incorporated in the CSP to prune the tree later. The unusual empty 0-ary constraint induced last states that the CSP is inconsistent and its justification is a (here minimal) justification of the CSP inconsistency. On this small problem, BT performs 45 constraint checks (*cc*) whereas $\mathrm{NR}_2$ performs only 21 *cc*.

## 3 Artificially augmenting failures

The NR algorithms involves too different "inference schemes":

1. the usual BT top down scheme states that if an assignment is consistent, we may try to extend it by assigning a value to a unassigned variable. Backtrack occurs as soon as a *single* constraint violation is detected.
2. the new $\mathrm{NR}_*$ bottom up scheme notices *one* single elementary nogood $(\mathscr{A}, \{c\})$ on each inconsistent leaf ($c$ being the first constraint checked as violated by $\mathscr{A}$), and then try to merge and improve these data when going back up in the tree.

The second inference scheme gets its data (basic nogoods) from the failure of the first one. The general idea is then to artificially lower the performances of the BT scheme to get more nogoods and make the cooperation of the two schemes more fruitful.

There are obviously many ways to make BT less efficient. In this paper, we will only consider the following simple "modification": instead of stopping consistency checks as soon as 1 constraint is violated, we will wait until either a given number $s$ of constraints are violated or all the constraints checkable at this node have been checked. This will be called *stubborn checking*. The usual *backward-checking* corresponds to $s = 1$ and the case we will now consider corresponds to $s = 2$.

In this case, stubborn checking allows us to build between 1 and 2 elementary nogoods on inconsistent leaves. All the previous theorems still apply and projecting will be applied to each nogood built. There is however a problem with merging: if all possible extensions of a consistent assignment $\mathscr{A}$ along a variable $x$ whose domain has cardinal $d$ are inconsistent, we may build 2 nogoods on each son of the corresponding node. Theorem 2.2 may then be applied on *any* set of $d$ nogoods, one nogood being chosen from each son. It is therefore possible to build up to $2^d$ nogoods ! Among the various possible limitations of this exponential scheme, the following one has been implemented: we effectively build the $2^d$ nogoods but only keep $j$ nogoods with the smallest justification set cardinality. Only these nogoods will be used for nogood recording, backjumping and for further merging. We are aware of the weaknesses of this simple limitation scheme:

1. it can not be applied on CSP with large domains since up to $j^d$ nogoods are built on each node[4];
2. nogood recording could well be performed on the $j^d$ effectively built nogoods, the $j$ best nogoods only being returned for merging.

However, this simple scheme is well suited to a first evaluation of the interest of stubborn checking.

One last choice has to be made: if two (or more) nogoods $(\mathscr{A}, J)$ and $(\mathscr{A}', J')$ are finally selected at a given node, we may choose to backjump using either of these nogoods. Two possible choices will be considered here:

1. backjump using the nogood whose assignment has the higher lowest variable. This will be called "*High Stubborn Nogood Recording*" (HSNR). One should note that the other nogood justification should not be returned after backjumping since it is not a justification of the global inconsistency of the assignment reached after backjumping. This only affects backjumping, not recording.

[4] Actually the exponential number of nogoods built at each node simply prevents the stubborn algorithms from being applied to problems with large domains. However, it is not necessary to build all the possible nogoods. A smarter approach would consist in directly generating $j$ nogoods with a small cardinality, even if this is not the smallest possible cardinality.

2. backjump using the nogood whose assignment has the lower lowest variable. This will be called "*Low Stubborn Nogood Recording*" (LSNR). In this case, the justification sets of both nogoods are still valid justification of the global inconsistency of the assignment reached after backjumping.

```
StubbornNR_k (𝒜, V)
If V = ∅
    Then 𝒜 is a solution, Stop
    Else
    Let x be a variable in V, SJ = ∅, BJ=false
    For each ν ∈ d(x) until BJ
        Let 𝒜' be 𝒜 ∪ {x = ν}, K be stubborn-check(𝒜')
        If K = ∅,
            Then Let SJ-sons be StubbornNR(𝒜', V − {x})
            If involved+(x, SJ-sons)
                Then SJ ← SJ ⋓ SJ-sons
                Else SJ ← SJ-sons, BJ← true
            Endif
            Else SJ ← SJ ⋓ K
        Endif
    Endfor
    SJ ← best-j(SJ)
    If BJ=false Then
        Let SNog-A be project+(𝒜, SJ)
        record+_k(SNog-A,SJ)
    Endif
    Return SJ
Endif
```

**Table 2.** Pseudo-code for the *Low Stubborn NR* algorithm

The usual $k^{th}$ order limitation directly applies. The code of LSNR$_*$ is directly derived from the code of NR$_*$ by redefining some functions (cf. table 2): the function **check** is replaced by the function **stubborn-check** which acts as **checks** but returns a set of (at most $s$) sets of violated constraints (or justifications) instead of a simple set of constraints. The function $\cup$ is replaced by the function $\Cup$, which applied to two sets of sets $A$ and $B$ returns $\{a \cup b, a \in A, b \in B\}$. The function **involved+** returns *true* iff its first argument, a variable $x$ is **involved** in one of the member of its second argument, a set of justifications. The function **best-**$j$ receives a set $SJ$ of justification and returns a subset of $SJ$ with at most $j$ elements whose cardinality is minimum. Finally, functions **record+**$_k$ and **project+** are simply versions of **record**$_k$ and **project** extended to cope with set of sets of constraints, applying the previous version on each element and returning the set of results (if any), as would the `mapcar` function do in most Lisp dialects. If the CSP is inconsistent, the function **LowStubbornNR** will return a set of justifications of inconsistency. It should be initially called with $\mathscr{A} = \varnothing$, the empty assignment and $V = X$ the set of all the CSP variables.

The HSNR$_*$ algorithm is obtained by a suitable modification of **involved+**: not only should this function check whether the variable $x$ is **involved** in one of the justifications, but it should also remove all such justifications from $SJ$-sons. Backjump occurs (*BJ*← *true*) if the variable $x$ is not **involved** in any justification in $SJ$-sons (as in LSNR$_*$) but also if $SJ$-sons is not empty after the removal of all the justifications in which $x$ was **involved** (this means that there are still some nogoods that allows us to backjump further).

Consider again the example of figure 1 and the two assignments $\mathscr{A}_1 = \{x_1 = a, x_2 = a, x_3 = a\}$ and $\mathscr{A}_2 = \{x_1 = a, x_2 = a, x_3 = b\}$. When stubborn checking (with $s > 1$) is performed, $\mathscr{A}_1$ and $\mathscr{A}_2$ are found locally inconsistent because of $c_{32}$ **or** $c_{31}$. Merging these 2 pairs of basic nogoods, keeping only the two smallest justifications and then projecting makes it possible to built the two nogoods $(\{x_2 = a\}, \{c_{32}\})$ and $(\{x_1 = a\}, \{c_{31}\})$. The HSNR algorithms will then backjump to $x_1$, the LSNR will simply backtrack to $x_2$. With $s = j = k = 2$, HSNR simply performs 17 *cc* on this problem.

## 4 Analysis and Conclusion

Experiments have been performed using the single ZEBRA problem, described in [3]. This problem has some good properties: it is not completely trivial, it includes various types of different constraints and involves 64 constraints (including two unary constraints) and 25 variables with uniform domain size 5. By using different static order of instantiation, the ZEBRA problem defines 25! significantly different search problems, often used for a first evaluation of CSP algorithms [3]. According to the order used, a simple BT algorithm may perform from thousands to several hundred millions of constraint checks. A total of 800 different instances of this problem have been considered and solved using the following algorithms *backtrack* (BT), *nogood recording* order 0, 1 and 2 (NR$_0$ ≈ CBJ, NR$_1$, NR$_2$) [9, 8, 7] with their stubborn versions (LSNR$_0$, HSNR$_0$, LSNR$_1$... with $s = 2$ and $j = 1$), *forward checking* (FC) and its hybrid nogood recording versions (NR-FC$_0$ ≈ FC-CBJ, NR-FC$_1$ and NR-FC$_2$), presented in [7, 8].

Each problem is solved *twice* by each algorithm, without any horizontal or vertical ordering heuristics. A second satisfaction is performed in the same way, using all the constraints recorded during the first satisfaction (if any).

For each algorithm, table 3 below gives the average ($\mu$), the standard deviation ($\sigma$), the minimum ($\veebar$) and the maximum ($\barwedge$) numbers of consistency checks for the first satisfaction over the 800 instances, rounded to enhance readability. This measure is usually very well correlated with run times. The last column indicates how often one algorithm bettered *all* the others. The HSNR$_2$ gives promising results

| Alg. | $\mu$ | $\sigma$ | $\veebar$ | $\barwedge$ | best |
|---|---|---|---|---|---|
| BT | 7 530 000 | 20 000 000 | 3540 | 483 000 000 | 0 |
| NR$_0$ | 440 000 | 819 000 | 1580 | 9 800 000 | 0 |
| NR$_1$ | 233 000 | 349 000 | 1570 | 3 720 000 | 0 |
| NR$_2$ | 172 000 | 271 000 | 1570 | 3 440 000 | 0 |
| LSNR$_0$ | 307 000 | 642 000 | 1340 | 9 530 000 | 0 |
| LSNR$_1$ | 99 900 | 136 000 | 1010 | 1 680 000 | 0 |
| LSNR$_2$ | 21 300 | 21 500 | 992 | 252 000 | 16 |
| HSNR$_0$ | 215 000 | 486 000 | 1340 | 8 670 000 | 0 |
| HSNR$_1$ | 72 500 | 100 000 | 1010 | 1 340 000 | 0 |
| HSNR$_2$ | 18 000 | 17 400 | 992 | 128 000 | 394 |
| FC | 103 000 | 139 000 | 279 | 2 150 000 | 1 |
| NR-FC$_0$ | 34 600 | 47 200 | 279 | 377 000 | 110 |
| NR-FC$_1$ | 31 800 | 43 600 | 279 | 377 000 | 163 |
| NR-FC$_2$ | 24 100 | 29 300 | 279 | 312 000 | 302 |

**Table 3.** Number of consistency checks

and is ranked first, before the NR-FC$_2$ algorithm. One should not forget, however, that these results were obtained without any heuristics and on the ZEBRA problem. On the 800 instances solved, NR-FC$_0$ was never better than NR-FC$_1$. This is also true for HSNR$_0$ and HS-NR$_1$. However, these experimental results of "superiority" should be confirmed by further experiments and... a theorem. We now consider the number of nodes explored, a measure of the size of the space explored (see table 4). NR-FC$_2$ is ranked first, but HSNR$_2$ also gives

| Alg. | $\mu$ | $\sigma$ | $\veebar$ | $\barwedge$ | best |
|---|---|---|---|---|---|
| BT | 3 170 000 | 8 560 000 | 1630 | 195 000 000 | 0 |
| $NR_0$ | 183 000 | 335 000 | 650 | 5 480 000 | 0 |
| $NR_1$ | 97 700 | 146 000 | 650 | 1 640 000 | 0 |
| $NR_2$ | 70 100 | 110 000 | 650 | 1 430 000 | 0 |
| $LSNR_0$ | 95 400 | 191 000 | 488 | 2 720 000 | 0 |
| $LSNR_1$ | 31 100 | 41 900 | 380 | 487 000 | 0 |
| $LSNR_2$ | 5 370 | 5 160 | 350 | 58 000 | 4 |
| $HSNR_0$ | 66 800 | 145 000 | 488 | 2 470 000 | 0 |
| $HSNR_1$ | 22 500 | 30 800 | 380 | 391 000 | 0 |
| $HSNR_2$ | 4 550 | 4 200 | 339 | 29 700 | 142 |
| FC | 15 200 | 20 200 | 46 | 301 000 | 1 |
| $NR\text{-}FC_0$ | 4 800 | 6 430 | 46 | 52 600 | 129 |
| $NR\text{-}FC_1$ | 4 440 | 5 910 | 46 | 52 600 | 207 |
| $NR\text{-}FC_2$ | 3 260 | 3 820 | 46 | 41 600 | 797 |

**Table 4.** Number of nodes explored

| Alg. | Time ($\mu$) | Saved | Quality | checks/sec. |
|---|---|---|---|---|
| BT | 562 | 0 % | 0 | 13 200 |
| $NR_0$ | 42.9 | 0 % | 0 | 10 100 |
| $NR_1$ | 23.3 | 66 % | 1.78 | 9 900 |
| $NR_2$ | 17.2 | 85 % | 2.33 | 9 900 |
| $LSNR_0$ | 24.6 | 0 % | 0 | 12 200 |
| $LSNR_1$ | 8.34 | 62 % | 2.41 | 11 800 |
| $LSNR_2$ | 2.09 | 87 % | 4.55 | 10 300 |
| $HSNR_0$ | 17.2 | 0 % | 0 | 12 300 |
| $HSNR_1$ | 6.00 | 60 % | 2.46 | 11 900 |
| $HSNR_2$ | 1.73 | 86 % | 4.53 | 10 500 |
| FC | 12.1 | 0 % | 0 | 8 490 |
| $NR\text{-}FC_0$ | 4.67 | 0 % | 0 | 7 300 |
| $NR\text{-}FC_1$ | 4.34 | 66 % | 0.68 | 7 240 |
| $NR\text{-}FC_2$ | 3.35 | 86 % | 1.05 | 7 140 |

**Table 5.** Time and checks per second

interesting results (lowest "worst case" performance). This is rather good: prospective algorithms such as FC explore many states at each node during constraint propagation, but just one node is counted. This is not so for the $^{**}NR_*$ algorithms (the nogood recording is related to local consistency enforcing in [8]).

Table 5 gives various measures:

- **run-times and improvement**: these figures have much less value than the previous ones, since they depend on many factors (quality of implementation, language used, machine used. here an implementation in Lisp, on a Sparc). However, it is often argued that it is easy to lower the number of consistency checks using sophisticated algorithms, but the overheads induced by the sophistication may finally lead to a *slower* algorithm. The following table gives average run times as well as the improvement observed during the second satisfaction. This gives an estimation of the possible efficiency of the algorithms for solving Dynamic CSP (cf. [8, 9, 10]).
- **learning quality**: for each algorithm, we have measured $\sum \frac{1}{d^i}$ on all recorded nogood assignments that do not contain any other nogood assignment, $d$ being the domain size (here 5), and $i$ the arity of each assignment. It is taken as an overall rough estimate of learning "quality".
- **checking rate**: *i.e.*, the average number of consistency checks per second is given as an estimation of the overheads associated to each algorithm.

This last table shows clearly that the $HSNR_2$ algorithm is the best possible choice among those considered. Not only does it performs less constraint checks than the $NR\text{-}FC_*$ algorithms, but it also checks constraints faster and has the smallest standard deviation. It is therefore the most robust algorithm w.r.t. variable ordering. It also offers, with the $LSNR_2$ algorithm, the best learning quality and is therefore specially well suited for the Dynamic CSP solution maintenance problem [8, 9].

## Conclusion

The previous experiment shows that stubbornness can bring noticeable improvements in the performance of nogood recording and backjumping algorithms. However, further experiments are needed to determine the scope of this conclusion and the precise memory requirements of the algorithms. It has been observed, for example, that the $\{H,L\}SNR_*$ algorithms are less efficient than the simple BT algorithms on the N-queens problem. This clearly shows that neither the $N$-queen problems, nor the ZEBRA problem can be considered alone as valuable benchmarks for any CSP algorithm. Random CSPs (hard or not) may give some additional information.

Among possible further researches, one may consider other ways of being stubborn (*i.e.*, generating more failures), other schemes for limiting the number of nogood built and handled at each node, the effect of constraint ordering heuristics on efficiency and a possible application of the *stubborn-checking* scheme to other backjumping algorithms such as *Dynamic Backtracking* (a specific instance of *Intelligent Backtracking* [2] described in [5]).

Stubbornness defines in fact a space of backjumping and learning algorithms (using parameters $k, j, s$): CBJ corresponds to $k = 0, j = s = 1$ whereas BJ corresponds to $k = j = 0, s = 1$... Only a very small region of this space has been actually considered.

## REFERENCES


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

[2] M. Bruynooghe, 'Solving combinatorial serach problems by intelligent backtracking', *Information Processing Letters*, **12**(1), 36–39, (1981).

[3] Rina Dechter, 'Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition', *Artificial Intelligence*, **41**, 273–312, (1990).

[4] J. Gaschnig, *Performance measurement and analysis of certain Search Algorithms*, Ph.D. dissertation, Carnegie-Mellon University, 1979.

[5] Matthew L. Ginsberg, 'Dynamic backtracking', *Journal of Artificial Intelligence Research*, **1**, 25–46, (1993).

[6] Alan K. Mackworth, 'Consistency in networks of relations', *Artificial Intelligence*, **8**, 99–118, (1977).

[7] Patrick Prosser, 'Hybrid algorithms for the Constraint Satisfaction Problem', *Computational Intelligence*, **9**(3), (August 1993).

[8] Thomas Schiex and Gérard Verfaillie, 'Nogood recording for static and dynamic CSP', in *Proc. of the $5^{th}$ IEEE International Conference on Tools with Artificial Intelligence*, (November 1993).

[9] 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, (August 1993).

[10] Gérard Verfaillie and Thomas Schiex, 'Solution reuse in dynamic constraint satisfaction problems', in *Proc. of AAAI-94*, Seattle, WA, (August 1994).