Two Approaches to the Solution Maintenance Problem in Dynamic Constraint Satisfaction Problems

Thomas Schiex*
Gérard Verfaillie†

June 17, 1993

Abstract

Many AI synthesis problems such as planning or scheduling may be modelized as constraint satisfaction problems (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 notion of dynamic CSP (DCSP) [DD88] has been proposed to represent such evolutions. The problem we consider here is the solution maintenance problem in a DCSP.

Naively applying usual satisfaction algorithms to this problem results in redundant search and inefficiency. Two distinct approaches to this problem are proposed in this paper. The first one, called the *Local Changes* approach, reuses the previous solution (if any) to compute a new solution. The second one, called the *Nogood Recording* approach, tries to store in a concise way, an approximate description of the frontier of the solution space and justifications of this frontier in terms of set of constraints.

Introduction

The so-called *Constraint Satisfaction Problem* (or CSP) framework may be considered as a fundamental tool for AI scheduling systems [FS90]. One of the problems encountered when applying CSP tools in scheduling applications is the difficulty to efficiently maintain a solution when the CSP is modified. This may occur either because of:

- **changes**: the problem to be solved changes (urgent orders, machine failures...);
- **decentralization/interaction**: the CSP only represented a part of a larger (decentralized) problem and another entity involved in the process of solving the global problem changed its mind on how it should be done[1].

In any case, we are brought to try to solve, as efficiently as possible, the *solution maintenance problem* in a dynamic CSP. A dynamic CSP may be simply defined as a sequence of classic CSP, each one differing from the previous one in some constraints (either added or deleted). The aim is to successively solve each of these CSP. A first naive approach is to simply solve each CSP, one after the other, using a classical CSP algorithm. This approach has two major flaws:

*e-mail address: schiex@cert.fr.
†e-mail address: verfail@cert.fr.
[1]This includes possible interactive problem solving: the entity being the user.

- **inefficiency**: it seems obvious that in many cases, some of the job done on the previous CSP could and *should* be somehow reused to solve the new CSP. A minimal approach is to, somehow, exploit the previous solution (if any). More complex approaches are worth considering;
- **lack of stability**: if the previous CSP solution represents a scheduling currently under execution, any new solution should be minimize the effort needed to apply this new scheduling. As for the decentralized/interactive problem, minimal change seems to be a reasonable local attitude to ease the convergence towards a global solution.

This paper considers two new approaches trying both to (more or less) reuse previous work. The first one applies *Local Changes* on a previous solution and has the desirable property of being complete[2]. The second one extends usual CSP tree search algorithms (Backtrack, Forward Checking) using a *Nogood Recording* scheme.

1 Definitions

1.1 The CSP model

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 e constraints. Each variable $x_i \in X$ take 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 the Cartesian product $d(x_{c_1}) \times \cdots \times d(x_{c_{a_k}})$ which specifies which values of the variables involved in the constraint 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 one in which all the constraints are binary *i.e.*, involve at most two variables. Its associated constraint hyper-graph is a graph. The methods proposed in this paper are not restricted to binary CSP.

Consider for instance the CSP graph presented in figure 1. 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. A CSP is said* consistent *if it has (at least) one solution. An assignment that may be extended to a solution is said* globally consistent.

The usual task in a CSP is to find one solution and 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 since each ordering defines how soon a constraint may be checked. The ordering used may either be pre-computed or may vary dynamically during 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.

[2] When a solution exists, a solution is found.

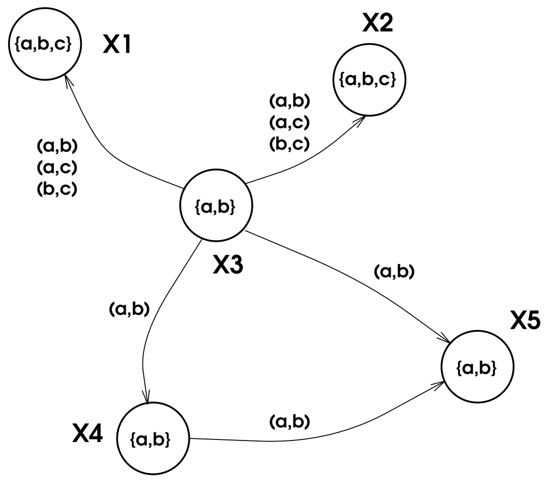

Figure 1: An example of binary CSP

One way to improve *Backtrack* efficiency is by making possible earlier detection of non globally consistent assignments. 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.*

This is usually called constraint propagation and has been formalized in various levels of so-called local consistencies: arc-consistency, path-consistency, k-consistency and has been related to ATMS nogood inference in [dK89]. Adding enough induced constraints may even lead to backtrack-free or backtrack-bounded search [Fre88].

1.2 Dynamic constraint satisfaction problems

Definition 1.4 *A dynamic constraint satisfaction problem $\mathscr{P}$ is a sequence $\mathscr{P}_0, \ldots, \mathscr{P}_\alpha, \ldots$ of static CSPs each resulting from a change in the preceding one [Bes91]. 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).*

So, for any α, if $\mathscr{P}_\alpha = (X, C_\alpha)$, then we have $\mathscr{P}_{\alpha+1} = (X, C_{\alpha+1})$ where $C_{\alpha+1} = C_\alpha \pm c$, c being a constraint. 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 $\mathscr{P}_0 \ldots, \mathscr{P}_\alpha \ldots$

1.3 Solution maintenance in DCSPs

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 apply:

1. if an assignment $\mathscr{A}$ is a solution of $\mathscr{P}_\alpha = (X, C_\alpha)$ then it is also a solution of $(X, C_\alpha - c)$: upon relaxation, and if there is a previous solution, one may simply keep this solution;
2. conversely, if an assignment $\mathscr{A}$ is not a solution of $\mathscr{P}_\alpha = (X, C_\alpha)$ then it is neither a solution of $(X, C_\alpha + c)$. Stated otherwise: any constraint c' induced by $\mathscr{P}_\alpha$ is also induced by $(X, C_\alpha + c)$. Therefore, if $\mathscr{P}_\alpha$ is inconsistent, then $\mathscr{P}_{\alpha+1}$ is also inconsistent and there is nothing to do.

3. finally, if the CSP (X, C_α) is inconsistent, then the constraint $\neg c^3$ is induced by the CSP $(X, C_\alpha - c)$ and may thus be added to $(X, C_\alpha - c)$ without losing any solution.

These properties are the basis for two differentapproaches to the DCSP solution maintenance problem:

1. *solution reuse*: each time a CSP $\mathscr{P}_\alpha$ is solved, store the solution so as to try to build a solution for $\mathscr{P}_{\alpha+1}$;
2. *reasoning reuse*: each time a CSP $\mathscr{P}_\alpha$ is solved, store constraints induced by $\mathscr{P}_\alpha$ to solve $\mathscr{P}_{\alpha+1}$ more efficiently.

2 Solution reuse approaches

These approaches suppose that the only information we have for solving $\mathscr{P}_{\alpha+1}$ is a solution of $\mathscr{P}_\alpha$. Thus, the first previous property immediately applies on relaxations of a consistent CSP. Additional algorithmic is needed for restrictions, or relaxation of an inconsistent CSP (the last remark may be used to improve performance in this case).

2.1 State of the art

A first approach was suggested in [MJPL90]. It was developed in [MJPL92] and presented in another way in [CCT91]. The algorithm, developed from the classic *Backtrack* algorithm and called the *Repair* algorithm, can be described as follows :

- the starting situation is a complete assignment, consistent or not, without any *fixed* variable-value assignment;
- any situation considered will be a complete assignment, consistent or not. Some of the variable-value assignments are said *fixed* (vars-done);
- the basic operation consists in :
  - choosing an unsatisfied constraint,
  - choosing a variable connected by this constraint, whose assignment is not fixed,
  - choosing a new assignment for it and
  - fixing its assignment (new var-done);
- the global dynamics consists in repeating this operation until a consistent assignment is found (no unsatisfied constraint) or until a failure occurs (unsatisfied constraint whose connected variables have their assignment fixed (vars-done), which implies backtracking on the last choice).

As for the *Backtrack* algorithm, this one is complete (a solution is found if a solution exists). Experimental results show that it can be greatly improved by using the so-called *min-conflicts* heuristics: when assigning a variable, always choose the value which minimizes the number of unsatisfied constraints.

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

2.2 The *Local Changes* approach

It is inspired by both *Backjumping* techniques and *Iterative Repair* approaches. The main difference with the *Repair* approach is that we start from *any partial or complete consistent assignment* of a previous CSP.

- *Backjumping* [Gas79, Dec90, Pro93] avoids exploring useless new alternatives, but it destroys, with no reason, part of the current assignment when backjumping;
- *Iterative Repair* approaches [MJPL92, CCT91] use an efficient conflicts minimization heuristics, but usually rely, if completeness is desired, on an inefficient *Backtrack* algorithm.

Using the same framework, the proposed *Local Changes* algorithm can be described as follows :

- the starting situation is a partial consistent assignment, without any *fixed* variable-value assignment;
- any situation considered will be a partial consistent assignment. Some of the variable-value assignments are said *fixed*;
- the basic operation consists in :
  - choosing a variable x to be assigned or reassigned,
  - choosing an assignment for it,
  - fixing its assignment,
  - determining the set C' of the unsatisfied constraints by this assignment,
  - determining a set X' of variables, whose assignments are not fixed, which covers the constraints of C' *i.e.*, such that each constraint of C' has at least one of its variables in X',
  - un-assigning, then reassigning the variables of X' and un-fixing the assignment of x;
- the global dynamics consists in repeating this operation until a complete assignment is found (no variable to be assigned or reassigned) or until a failure occurs (unsatisfied constraint whose all connected variables have their assignment fixed, which implies backtracking on the choice made for the last fixed variable).

This has been proven terminating and complete. An horizontal order heuristics may be used: for example, choose the value which minimizes the number of variables to be unassigned.

When used in the framework of dynamic CSP, the following scheme may be applied to the previous solution (or the largest consistent assignment found):

- for each violated constraint among those added, choose one variable to be unassigned (possible heuristics);
- attempt to assign all the unassigned variables using the previous algorithm.

Let us note that these two algorithms, derived from the *Backtrack* one, can similarly be improved by using filtering techniques (*e.g.* forward-checking). The single difference lies in the fact that the filtering can only be done from variables whose assignments are fixed.

Let us also note that, in the framework of the two algorithms, the cardinal of the smallest set of variables which covers the set of unsatisfied constraints, is a lower bound of the distance between the current assignment and a solution. If the distance between two assignments is the number of variables whose assignments differ. This remark allows an A^*-like algorithm to be used for minimizing the distance between two consecutive solutions.

3 Reasoning reuse approaches

These approaches suppose that some induced constraints, discovered during the resolution of $\mathscr{P}_\alpha$, should be used when solving $\mathscr{P}_{\alpha+1}$. This scheme is immediately usable for restrictions (2nd property exposed in section 1.3), additional algorithmic is needed for relaxations (the third property may be used when an inconsistent CSP is being relaxed).

3.1 State of the art

[HP91] proposed the following "oracles" scheme: given an 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 concisely described by *the first solution itself*. Therefore, each time solution s for a CSP (X, C) is found, we remember the pair (C, s) (called an oracle). It may be seen as an induced constraint: every assignment preceding s (according to the previous order) can not lead to a solution. This induced constraint may be used upon restriction (remark 2) by starting the next tree search from s. Upon relaxation, the previous oracle simply gives a solution. However, this scheme is difficult to apply when repeated relaxations and restrictions occur: after a relaxation, the solution given by a previous oracle may be not the *first solution* of the new problem, and therefore, if a restriction occurs next, we have to find another older oracle (s', C') such that the current CSP is a restriction of the CSP (X, C'). If none exists, a satisfaction is performed from scratch.

The most sophisticated approaches are inspired from Truth Maintenance Systems and have been applied to the arc-consistency maintenance problem [Bes91, PCM92]. In each of these approaches, unary induced constraints are computed and justified by constraint set(s) that may be not exhaustive and not minimal with respect to inclusion. Such approaches could be extended to higher consistency degrees and would need space in $O(n^k)$, where k is the consistency degree achieved. They naturally exploit CSP monotonicity but *do not solve* the satisfaction problem.

3.2 The *Nogood recording* approach

The constraint recording or so-called *learning* approach, proposed in [Dec90] proves, during satisfaction of a CSP (X, C) by a *Backtrack* algorithm, that some assignments are *non globally consistent*. These assignments define induced constraints that may have various uses. Specially, they may help in solving the solution maintenance problem upon restriction (remark 2), but additional algorithmic is needed for relaxations. The approach proposed extends this approach.

Rather than simply building "non globally consistent assignments", we will try to build *nogoods*, defined as pairs (A, C') where A is an assignment, C' is a subset of C (the set of constraints of the CSP being solved) and such that A is non globally consistent *with respect to the CSP* (X, C'). Naturally, we are interested in building nogoods with sets A and C' as small as possible. However, we will not try, as the ATMS does, to build all minimal nogoods (such a task is to heavy [Pro90] from a temporal and a spatial complexity point of view).

The following theorems (or inference rules) may be used to build and improve nogoods during *Backtrack* tree search:

Theorem 3.1 *If an assignment A violates a constraint $c \in C$, then $(A, \{c\})$ is a nogood.*

Obvious from nogood definition. However, such nogoods are uninteresting since they are already made explicit by c itself; they will only serve as a basis for building more interesting nogoods.

Theorem 3.2 *If $A_1 \ldots, A_n$ are all the possible extensions of an assignment A along a variable x (not assigned in A) and if $(A_1, C_1) \ldots, (A_n, C_n)$ are all nogoods then $(A, \cup_{i \in \{1 \ldots, n\}} C_i)$ is also a nogood.*

Informally, if A can not be extended to a globally consistent assignment on a given variable, it can not be a globally consistent assignment. Obviously, the first theorem applies on failure leaves, the second one at each node that corresponds to a non globally consistent assignment. Note that this situation is more general than the *dead-end* situation defined in [Dec90].

A last property offers a possible important improvement of nogoods. It is systematically applied each time a nogood is built.

Theorem 3.3 *Let (A, C') be a nogood, X' being the set of variables involved in the constraints in C', then $(A{\downarrow}X', C')$ is also a nogood.*

Informally, it is not possible for a variable involved in no *constraint to be responsible for non global consistency.*

Each nogood (A, C') may be simply interpreted as follows: the assignment A if forbidden because of the constraints in C'. The nogoods will serve four different purposes:

- each time a nogood is built at a given node, it will be possible to backjump to the lowest variable involved in the nogood constraint set, since any variable assignment should necessarily satisfy the constraint forbidding A;
- it will also be possible to add this constraint to the CSP being solved *during* the current tree search. This may enable earlier detection of non global consistencies and thus, prune the tree explored;
- it may be used to solve the CSP $\mathcal{P}_{\alpha+1}$: the constraint associated to each nogood (A, C') such that $C' \subset C_{\alpha+1}$ may be added to the CSP and prune the tree, thus enhancing efficiency;
- the nogoods give basic explanations for more or less local inconsistencies. In case of global inconsistency, the nogood built for the root is a (non necessarily minimal) explanation of inconsistency.

Naturally, there is a possibly exponential number of nogoods generated, and one should limit the recording to a subset of the most interesting nogoods. The usual restriction is to restrict oneself to nogoods whose assignment cardinality is less than a constant k [Dec90]. This is called k^{th} order nogood recording. Another possible restriction is to record only nogoods whose assignments connect variables that are already connected by a constraint in the CSP. When compared with the *full shallow* learning algorithm, the first obvious improvement is the justification handling: each constraint learned also gets a justification. The other improvement is on the *quality* of learning: Dechter *full shallow* kth order learning algorithm enforces in fact a subset of strong k-consistency. Our kth order nogood recording algorithm enforces a subset of strong $(k, n-j)$-consistency [Fre88] on the jth variable in the ordering used, a property much stronger than strong k-consistency. It will, for example, built a justification of inconsistency for any inconsistent CSP.

The following pseudo-code defines the algorithm more precisely. Such as it is defined, it will find the first solution only. Additional sophistications are required for the search of more than one solution. Let us first give the definition of simple functions used in the pseudo-code :

- the function **record** simply record the nogood built *i.e.*, incorporates the constraint induced by the nogood in the CSP along with the justification set itself so that the **check** function may work as described below ;

- the function **check** tests whether any constraint is violated by an assignment. If no constraint is violated, 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 by **record** because of a nogood (A, J): it returns J;
- 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-recorder**, which should initially be called with $A = \emptyset$, the empty affectation and $V = X$, the set of all the CSP variables.

```
NogoodRecorder(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 NogoodRecorder(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
  Let Nog-A be project(𝒜, J)
  record(Nog-A,J)
  Return J
Endif
```

Such as it is defined, the nogood recording algorithm has many interesting features:

- it does not rely, in any way on any fixed vertical or horizontal ordering;
- instead of the underlying *Backtrack* algorithm, one may use the supposedly more efficient *Forward-Checking* algorithm. The hybrid algorithm obtained checks forward, backjumps, and learns nogoods.

4 An example

Let us consider the problem exposed in figure 1. On this problem, the *Local Changes* algorithm, starting from an empty assignment, performs 18 constraint checks. The execution is sketched in the figure 3. Each time some constraints are violated (constraints violated are given near the boxes), the variable to be assigned is *fixed* (underlined in the figure) and a set of non fixed assignment involved in the violations rejected. These rejected variables are then reassigned until a consistent assignment is found. In this case, the fixed variable get "un-fixed", otherwise a backtrack occurs. One should note that the biggest consistent assignment found is $(x_1 = b, x_2 = b, x_3 = a, x_4 = b)$. It will be used for the next satisfaction.

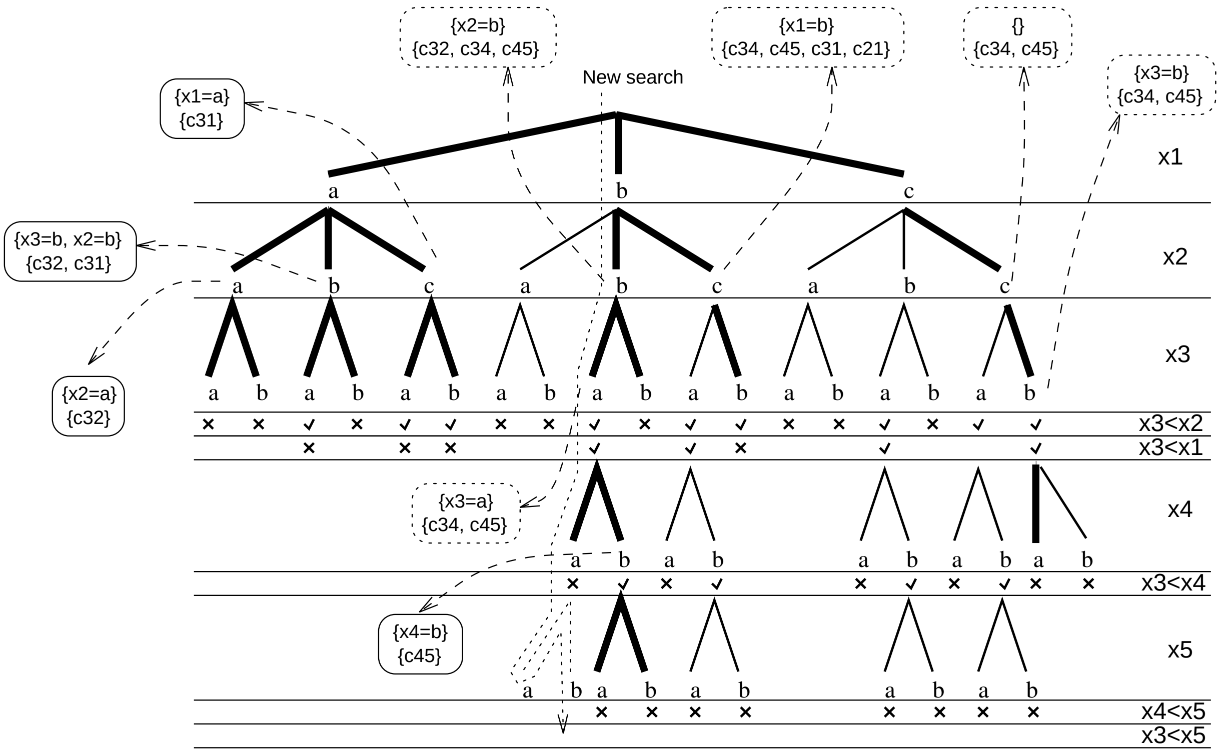

Figure 2: *Backtrack* and *Nogood Recording*

The figure 2 presents the tree explored by the *Backtrack* and *Conflict Based Backjumping* algorithms (both explore the full tree: 45 constraint checks). The *Nogood Recording* algorithm explores only the thick arcs and performs only 21 constraint checks. All the nogoods built are indicated, with their "birthplace". The CSP is insatisfiable and, in this case, a minimal justification for inconsistency is found : the nogood $(\emptyset, \{c_{34}, c_{45}\})$ is built.

Let us now perform a non trivial relaxation: we suppress the constraint c_{34} between x_3 and x_4. The *Local Changes* algorithm, starting from the previous biggest consistent assignment found, performs 9 constraints checks before it finds a solution (cf. figure 4).

A new satisfaction by the *Backtrack* algorithm leads to a total of 16 constraint checks. Using the *Nogood Recording* algorithm, 5 unjustified induced constraints are suppressed (dotted boxes). Nevertheless, we directly go to the first solution, checking 7 constraints only (following the dotted arrow labelled "`New Search`" in figure 2.

5 Experimentations

Experimentations on random static and dynamic CSP have taken place independently for the two approaches. The *Local changes* algorithm has been compared with the *Backtrack*, *Dependency Directed Backtracking* (also referred as *Conflict Based Backjumping* in [Pro93]) and *Forward checking* algorithms. These experiments showed that the *Local changes* algorithm is generally more efficient than the *Backtrack* and *Backjumping* ones, and is the most efficient for sparse and soft-constrained problems ; the *Forward-checking* algorithm is, on the contrary, the most efficient for dense and hard-constrained problems.

The *Local changes* algorithm is even more efficient when dealing with dynamic problems than when dealing with static ones. But an experimental comparison between the *Repair* and *Local changes*

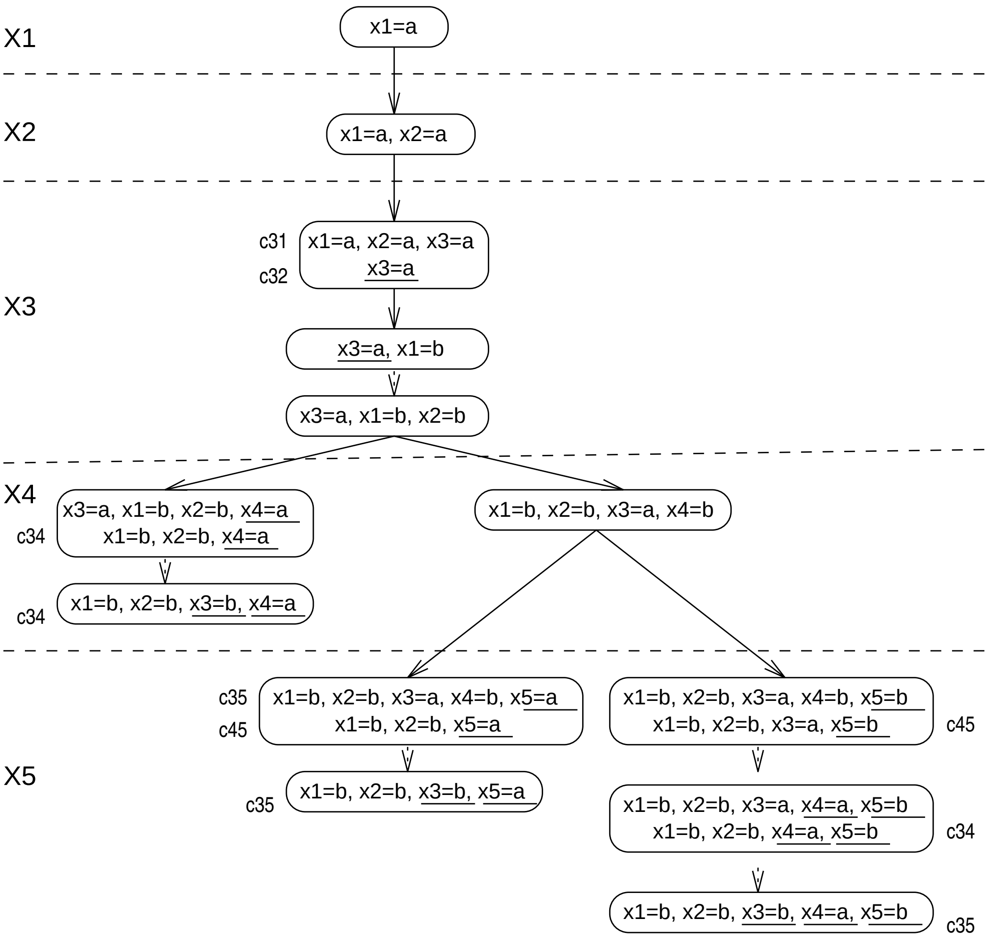

Figure 3: *Local Changes*

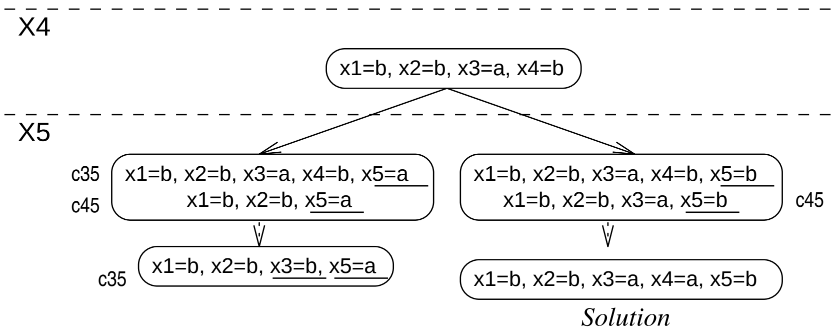

Figure 4: A new search by *Local Changes*

algorithms is still to be done.

The *Nogood Recording* (NR) approach, limited to order 0, 1 and 2, both on top of the *Backtrack* and *Forward-checking* algorithms (defining algorithms NR0 to NR2 and NR-FC0 to NR-FC2) has been compared with the previous algorithms and with the *Full Shallow Learning* algorithm [Dec90]. A first observation is that, for static CSPs, first order recording (simple to implement) brings important savings when compared to 0^{th} order nogood recording (a simple *dependency directed backtracking*, very similar to the *Conflict Directed Backjumping* presented in [Pro93]). However, second order seems to be usually more efficient. This is true in terms of number of nodes explored, number of constraint checks and time.

For the satisfaction of static random CSP, the 2nd order *Nogood Recording* algorithm on top of the *Forward-checking* algorithm is *the most efficient algorithm* (better than simple FC in any terms : nodes, constraint checks and time). The simple 0^{th} order NR algorithm is usually more efficient than the 2nd order *full shallow learning* algorithm.

For dynamic CSP, things are more complicated. The simple NR* are slower than the NR-FC*, but they built more smaller nogoods and may become faster when repeated new satisfactions are performed. Thus, the algorithm actually used depends on the actual temporal pressure. Anyway, both of them rapidly outperform all other algorithms. The authors may give precise experimentation protocol and results (perforce not included here).

6 Conclusion

Each of these two approaches deserve more studies: large-scale experimental evaluations, studies of the various customizations possible in the *Nogood Recording* approach, etc. Longer term directions include:

- mixing these approaches with preferences oriented approaches [BBB+92, FW92]. This is both to take into account preferences when possible and to cope with inconsistent CSP, an undesirable state in the scheduling applications;
- cooperation of these *complete*, but sometimes heavy algorithms, with more reactive approaches;
- using these "dynamic" approaches as the basic algorithm used by each "computerized" entity in a decentralized CSP.

References

[BBB+92] Gérard Bel, Eric Bensana, Pierre Berlandier, Philippe David, Hélène Fargier, Christine Gaspin, Khaled Ghedira, Philippe Janssen, Philippe Jégou, Tibor Kökeny, Jérome Lang, David Lesaint, Roger Martin-Clouaire, Bertrand Neveu, Jean-Pierre Rellier, Thomas Schiex, Brigitte Trousse, Gérard Verfaillie, and Marie-Catherine Vilarem, *Représentation et traitement pratique de la flexibilité dans les problèmes sous contraintes*, Actes des journées PRC-IA (Marseille, France), October 1992.

[Bes91] Christian Bessière, *Arc-consistency in dynamic constraint satisfaction problems*, Proc. of AAAI-91 (Anaheim, CA), 1991, pp. 221–226.

[CCT91] J. Chabrier, J.J. Chabrier, and F. Trousset, *Résolution efficace d'un problème de satisfaction de contraintes : le million de reines*, Proc. of the 11^{th} international Avignon workshop (France), 1991.

[DD88] Rina Dechter and Avi Dechter, *Belief maintenance in dynamic constraint networks*, Proc. of AAAI-88 (St. Paul, MN), 1988, pp. 37–42.

[Dec90] Rina Dechter, *Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition*, Artificial Intelligence **41** (1990), 273–312.

[dK89] Johan de Kleer, *A comparison of ATMS and CSP techniques*, Proc. of the 11^{th} IJCAI (Detroit, MI), August 1989, pp. 290–296.

[Fre88] Eugene C. Freuder, *Backtrack-free and backtrack-bounded search*, Search in Artificial Intelligence (L. Kanal and V. Kumar, eds.), Springer-Verlag, 1988, pp. 343–369.

[FS90] Mark S. Fox and Norman Sadeh, *Why is scheduling difficult ? a CSP perspective*, Proc. of the 9^{th} European Conference on Artificial Intelligence, 1990, pp. 754–767.

[FW92] Eugene C. Freuder and Richard J. Wallace, *Partial constraint satisfaction*, Artificial Intelligence **58** (1992), 21–70.

[Gas79] J. Gashnig, *Performance measurement and analysis of certain search algorithms*, Ph.D. thesis, Carnegie-Mellon University, 1979.

[HP91] Pascal Van Hentenryck and Thierry Le Provost, *Incremental search in contraint logic programming*, New Generation Computing **9** (1991), 257–275.

[MJPL90] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird, *Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method*, Proc. of AAAI-90 (Boston, MA), 1990, pp. 17–24.

[MJPL92] 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 **58** (1992), 160–205.

[PCM92] Patrick Prosser, Chris Conway, and Claude Muller, *A constraint maintenance system for the distributed allocation problem*, Intelligent Systems Engineering **1** (1992), no. 1, 76–83.

[Pro90] Gregory M. Provan, *The computational complexity of multiple-context truth maintenance systems*, Proc. of the 8^{th} ECAI (Stockholm), 1990, pp. 522–527.

[Pro93] Patrick Prosser, *Hybrid algorithms for the Constraint Satisfaction Problem*, Computational Intelligence **9** (1993), no. 3.