Solution Reuse in Dynamic Constraint Satisfaction Problems

Gérard Verfaillie and **Thomas Schiex**
ONERA-CERT
2 avenue Edouard Belin, BP 4025
31055 Toulouse Cedex, France
{verfail,schiex}@cert.fr

Abstract

Many AI problems can be modeled as constraint satisfaction problems (CSP), but many of them are actually dynamic: the set of constraints to consider evolves because of the environment, the user or other agents in the framework of a distributed system. In this context, computing a new solution from scratch after each problem change is possible, but has two important drawbacks: inefficiency and instability of the successive solutions. In this paper, we propose a method for reusing any previous solution and producing a new one by *local changes* on the previous one. First we give the key idea and the corresponding algorithm. Then we establish its properties: termination, correctness and completeness. We show how it can be used to produce a solution, either from an empty assignment, or from any previous assignment and how it can be improved using filtering or learning methods, such as *forward-checking* or *nogood-recording*. Experimental results related to efficiency and stability are given, with comparisons with well known algorithms such as *backtrack*, *heuristic repair* or *dynamic backtracking*.

Problem description

Recently, much effort has been spent to increase the efficiency of the constraint satisfaction algorithms: filtering, learning and decomposition techniques, improved backtracking, use of efficient representations and heuristics ...This effort resulted in the design of constraint reasoning tools which were used to solve numerous real problems.

However all these techniques assume that the set of variables and constraints which compose the CSP is completely known and fixed. This is a strong limitation when dealing with real situations where the CSP under consideration may evolve because of:

- the *environment*: evolution of the set of tasks to be performed and/or of their execution conditions in scheduling applications;
- the *user*: evolution of the user requirements in the framework of an interactive design;
- other *agents* in the framework of a *distributed system*.

The notion of dynamic CSP (DCSP) (Dechter & Dechter 1988) has been introduced to represent such situations. A DCSP is a sequence of CSPs, where each one differs from the previous one by the addition or removal of some constraints. It is indeed easy to see that all the possible changes to a CSP (constraint or domain modifications, variable additions or removals) can be expressed in terms of constraint additions or removals.

To solve such a sequence of CSPs, it is always possible to solve each one from scratch, as it has been done for the first one. But this naive method, which remembers nothing from the previous reasoning, has two important drawbacks:

- *inefficiency*, which may be unacceptable in the framework of real time applications (planning, scheduling, etc.), where the time allowed for replanning is limited;
- *instability* of the successive solutions, which may be unpleasant in the framework of an interactive design or a planning activity, if some work has been started on the basis of the previous solution.

Existing methods

The existing methods can be classified in three groups:

- *heuristic* methods, which consist of using any previous consistent assignment (complete or not) as a heuristic in the framework of the current CSP (Hentenryck & Provost 1991);
- *local repair* methods, which consist of starting from any previous consistent assignment (complete or not) and of repairing it, using a sequence of local modifications (modifications of only one variable assignment) (Minton *et al.* 1992; Selman, Levesque, & Mitchell 1992; Ghedira 1993);
- *constraint recording* methods, which consist of recording any kind of constraint which can be deduced in the framework of a CSP and its justification, in order to reuse it in the framework of any new

CSP which includes this justification(de Kleer 1989; Hentenryck & Provost 1991; Schiex & Verfaillie 1993).

The methods of the first two groups aim at improving both efficiency and stability, whereas those of the last group only aim at improving efficiency. A little apart from the previous ones, a fourth group gathers methods which aim at minimizing the distance between successive solutions (Bellicha 1993).

Key idea

The proposed method originated in previous studies for the French Space Agency (CNES) (Badie & Verfaillie 1989) which aimed at designing a scheduling system for a remote sensing satellite (SPOT). In this problem, the set of tasks to be performed evolved each day because of the arrival of new tasks and the achievement of previous ones. One of the requirements was to disturb as little as possible the previous scheduling when entering a new task.

For solving such a problem, the following idea was used: it is possible to enter a new task t iff there exists for t a location such that all the tasks whose location is incompatible with t's location can be removed and entered again one after another, without modifying t's location.

In terms of CSP, the same idea can be expressed as follows: let us consider a binary CSP; let A be a consistent assignment of a subset V of the variables;[1] let v be a variable which does not belong to V; we can assign v *i.e.*, obtain a consistent assignment of $V \cup \{v\}$ iff there exists a value *val* of v such that we can assign *val* to v, remove all the assignments (v', val') which are inconsistent with (v, val) and assign these unassigned variables again one after another, without modifying v's assignment. If the assignment $A \cup \{(v, val)\}$ is consistent, there is no variable to unassign and the solution is immediate. Note that it is only for the sake of simplicity that we consider here a binary CSP. As we will see afterwards, the proposed method deals with general n-ary CSPs.

With such a method, for which we use the name *local changes* (*lc*) and which clearly belongs to the second group (*local repair* methods), solving a CSP looks like solving a *fifteen puzzle* problem: a sequence of variable assignment changes which allows any consistent assignment to be extended to a larger consistent one.

Algorithm

The corresponding algorithm can be described as follows:

```
lc(csp)
    return lc-variables(∅, ∅, variables(csp))

lc-variables(V1, V2, V3)
; V1 is a set of assigned and fixed variables
; V2 is a set of assigned and not fixed variables
; V3 is a set of unassigned variables
    if V3 = ∅
    then return success
    else let v be a variable chosen in V3
         let d be its domain
         if lc-variable(V1, V2, v, d) = failure
         then return failure
         else return lc-variables(V1, V2 ∪ {v}, V3 − {v})

lc-variable(V1, V2, v, d)
    if d = ∅
    then return failure
    else let val be a value chosen in d
         save-assignments(V2)
         assign-variable(v, val)
         if lc-value(V1, V2, v, val) = success
         then return success
         else unassign-variable(v)
              restore-assignments(V2)
              return lc-variable(V1, V2, v, d − {val})

lc-value(V1, V2, v, val)
    let be A1 = assignment(V1)
    let be A12 = assignment(V1 ∪ V2)
    if A1 ∪ {(v, val)} is inconsistent
    then return failure
    else if A12 ∪ {(v, val)} is consistent
         then return success
         else let V3 a non empty subset of V2 such that
                  let A123 = assignment(V1 ∪ V2 − V3)
                  A123 ∪ {(v, val)} is consistent
              unassign-variables(V3)
              return lc-variables(V1 ∪ {v}, V2 − V3, V3)
```

Properties

Let us consider the following theorems:

Theorem 1 *If the CSP csp is consistent (resp. inconsistent), the procedure call lc(csp) returns success (resp. failure); in case of success, the result is a consistent assignment of csp's variables.*

Theorem 2 *Let V_1 and V_2 be two disjunct sets of assigned variables and let V_3 be a set of unassigned variables; let be $V = V_1 \cup V_2 \cup V_3$; let be $A_1 = assignment(V_1)$; if there exists (resp. does not exist) a consistent assignment A of V, such that $A{\downarrow_{V_1}} = A_1$, the procedure call lc-variables(V_1, V_2, V_3) returns success (resp. failure);*[2] *in case of success, the result is a consistent assignment of V.*

Theorem 3 *Let V_1 and V_2 be two disjunct sets of assigned variables; let v be an unassigned variable; let d be its domain; let be $V = V_1 \cup V_2 \cup \{v\}$; let be*

[1] An assignment A of a subset of the CSP variables is consistent iff all the constraints assigned by A are satisfied; a constraint c is assigned by an assignment A iff all its variables are assigned by A.

[2] Let A be an assignment of a subset V of the CSP variables and V' be a subset of V; the notation $A{\downarrow_{V'}}$ designates the restriction of A to V'.

A_1 = assignment(V_1); if there exists (resp. does not exist) a consistent assignment A of V, such that $A\downarrow_{V_1} = A_1$, the procedure call lc-variable(V_1, V_2, v, d) returns success (resp. failure); in case of success, the result is a consistent assignment of V.

Theorem 4 *Let V_1 and V_2 be two disjunct sets of variables; let v be an unassigned variable; let val be one of its possible values; let be $V = V_1 \cup V_2 \cup \{v\}$; let be A_1 = assignment(V_1); if there exists (resp. does not exist) a consistent assignment A of V, such that $A\downarrow_{V_1\cup\{v\}} = A_1 \cup \{(v, val)\}$, the procedure call lc-value(V_1, V_2, v, val) returns success (resp. failure); in case of success, the result is a consistent assignment of V.*

Theorem 1 expresses the *termination*, *correctness* and *completeness* properties of the algorithm. Theorems 2, 3, 4 express the same properties for the procedures *lc-variables*, *lc-variable* and *lc-value*.

It is easy to show that Theorem 1 (resp. 2 and 3) is a straigthforward consequence of Theorem 2 (resp. 3 and 4).

Let us consider the set V_{23} of the not fixed variables ($V_{23} = V_2 \cup V_3$ for the procedure *lc-variables*, $V_{23} = V_2 \cup \{v\}$ for the procedures *lc-variable* and *lc-value*). It is just as easy to show that, if Theorem 3 (resp. Theorem 4) holds when $|V_{23}| < k$, then Theorem 2 (resp. Theorem 3) holds under the same condition.

Let us now use an induction on the cardinal of V_{23} to prove Theorems 2, 3 and 4.

Let us assume that $|V_{23}| = 1$ and let us prove Theorem 4 in this case. Let us consider a procedure call *lc-value*($V_1, \emptyset, v, val$):

- let us assume that there exists a consistent assignment A of V, such that $A\downarrow_{V_1\cup\{v\}} = A_1 \cup \{(v, val)\}$; since $V = V_1 \cup \{v\}$, $A_1 \cup \{(v, val)\}$ and $A_{12} \cup \{(v, val)\}$ are equal and consistent and the procedure returns *success*; the resulting assignment $A_1 \cup \{(v, val)\}$ of V is consistent;
- let us now assume that there exists no consistent assignment A of V, such that $A\downarrow_{V_1\cup\{v\}} = A_1 \cup \{(v, val)\}$; since $V = V_1 \cup \{v\}$, $A_1 \cup \{(v, val)\}$ is inconsistent and the procedure returns *failure*.

Theorem 4, and consequently Theorem 3 and 2 are proven in this particular case.

Let us assume that Theorems 2, 3 and 4 hold when $|V_{23}| < k$ and let us prove that they hold when $|V_{23}| = k$.

Let us first consider Theorem 4 and a procedure call *lc-value*(V_1, V_2, v, val), with $|V_2| = k - 1$. Let us note that, when the procedure *lc-variables* is recursively called, its arguments satisfy the following relations: $V_2' \cup V_3' = V_2$ ($|V_{23}'| = k - 1$) and $V_1' \cup V_2' \cup V_3' = V_1 \cup V_2 \cup \{v\} = V$. This allows us to use the induction assumption:

- let us assume that there exists a consistent assignment A of V, such that $A\downarrow_{V_1\cup\{v\}} = A_1 \cup \{(v, val)\}$; since $A_1 \cup \{(v, val)\}$ is consistent, the procedure does not immediately return *failure*; either $A_{12} \cup \{(v, val)\}$ is consistent and the procedure returns immediately *success*, with a consistent assignment of V, or it is not and:
  - there exists a non empty subset V_3 of V_2 such that $A_{123} \cup \{(v, val)\}$ is consistent: for example, V_2;
  - whatever the set chosen for V_3, the call to *lc-variables* returns *success* with a consistent assignment of V, according to the induction assumption;
- let us now assume that there exists no consistent assignment A of V, such that $A\downarrow_{V_1\cup\{v\}} = A_1 \cup \{(v, val)\}$; since $A_{12} \cup \{(v, val)\}$ is inconsistent, the procedure does not immediately return *success*; either $A_1 \cup \{(v, val)\}$ is inconsistent and the procedure returns immediately *failure*, or it is not and:
  - there exists a non empty subset V_3 of V_2 such that $A_{123} \cup \{(v, val)\}$ is consistent: for example, V_2;
  - whatever the set chosen for V_3, the call to *lc-variables* returns *failure*, according to the induction assumption.

Theorem 4 and consequently Theorems 3 and 4 are proven, when $|V_{23}| = k$. They are therefore proven whatever the cardinal of V_{23}. That allows us to conclude that Theorem 1 is proven *i.e.*, that the algorithm described above ends, is correct and complete.

Practical use

From a practical point of view, the problem is now to choose a set V_3 that is as small as possible, in order to reduce the number of variables that need to be unassigned and subsequently reassigned.

In the general case of n-ary CSPs, a simple method consists of choosing one variable to be unassigned for each constraint which is unsatisfied by the assignment $A_{12} \cup \{(v, val)\}$. The resulting assignment $A_{123} \cup \{(v, val)\}$ is consistent, since all the previously unsatisfied constraints are no longer assigned, but we have no guarantee that the resulting set V_3 is one of the smallest ones. Note that it does not modify the termination, correctness and completeness properties of the algorithm. It may only alter its results in terms of efficiency and stability. We did not compare the cost of searching for one of the smallest sets of variables to be unassigned with the resulting saving.

In the particular case of binary CSPs, a simpler method consists of unassigning each variable whose assignment is inconsistent with (v, val). The resulting set V_3 is the smallest one.

It is important to note that this algorithm is able to solve any CSP, either starting from an empty assignment (from scratch), or starting from any previous assignment. The description above (see *Algorithm*) corresponds to the first situation. In the second one, if A is the starting assignment, then the first step consists of producing a consistent assignment A' that is included in A and as large as possible. The method presented above can be used. If V_2 (resp. V_3) is the resulting set of assigned (resp. unassigned) variables, the CSP can be solved using the procedure call *lc-variables*($\emptyset, V_2, V_3$) (no fixed variable).

Comparisons and improvements

The resulting algorithm is related to the *backjumping* (Dechter 1990; Prosser 1993), *intelligent backtracking* (Bruynooghe 1981), *dynamic backtracking* (Ginsberg 1993) and *heuristic repair* (Minton *et al.* 1992) algorithms, but is nevertheless different from each of them. Like the first one, it avoids useless backtracking on choices which are not involved in the current conflict. Like the following two ones, it avoids, when backtracking, undoing choices which are not involved in the current conflict. Like the last one, it allows the search to be started from any previous assignment. But *backjumping*, *intelligent* and *dynamic backtracking* are not built for dealing with dynamic CSPs, and *heuristic repair* uses the usual backtracking mechanism. Finally, *local changes* combines the advantages of an efficient backtracking mechanism with an ability to start the search from any previous assignment.

Moreover, it can be improved, without any problem, by using any filtering or learning method, such as *forward-checking* or *nogood-recording* (Schiex & Verfaillie 1993). The only difference is the following one: for *backtrack*, *forward-checking* and *nogood-recording* are applied from the assigned variables; for *local changes*, as for *heuristic repair*, they are applied from the assigned and fixed variables. Note that the combination of *local changes* and *nogood-recording* is an example of solution and reasoning reuse.

Experiments

In order to provide useful comparisons, eight algorithms have been implemented on the basis of the following four basic algorithms: *backtrack* (*bt*), *dynamic backtracking* (*dbt*), *heuristic repair* (*hrp*) and *local changes* (*lc*), using *conflict directed backjumping* (*cbj*) and *backward* (*bc*) or *forward-checking* (*fc*): *bt-cbj-bc*, *bt-cbj-fc*, *dbt-bc*, *dbt-fc*, *hrp-cbj-bc*, *hrp-cbj-fc*, *lc-bc* and *lc-fc*.

Each time there is no ambiguity, we will use the abbreviations *bt*, *dbt*, *hrp* and *lc* to designate these algorithms. Note that *dbt* and *lc* can not be improved by *cbj*, because they already use a more powerful *backtracking* mechanism.

Heuristics

For each algorithm, we used the following simple yet efficient heuristics:

- choice of the *variable* to be assigned, unassigned or reassigned: choose the variable whose domain is the smallest one;
- choice of the *value*:
  - for *bt* and *dbt*: first use the value the variable had in the previous solution, if it exists;
  - for *hrp* and *lc*: choose the value which minimizes the number of unsatisfied constraints.

In the case of *bt*, *dbt* and *hrp*, the previous solution is recorded, if it exists. In the case of *bt* and *dbt*, it is used in the framework of the choice of the value. In the case of *hrp*, it is used as a starting assignment. In the case of *lc*, the greatest consistent assignment previously found (a solution if the previous problem is consistent) is also recorded and used as a starting assignment.

For the four algorithms, two trivial cases are solved without any search: the previous CSP is consistent (resp. inconsistent) and there is no added (resp. removed) constraint.

CSP generation

Following (Hubbe & Freuder 1992), we randomly generated a set of problems where:

- the number *nv* of variables is equal to 15;
- for each variable, the cardinality of its domain is randomly generated between 6 and 16;
- all the constraints are binary;
- the connectivity *con* of the constraint graph *i.e.*, the ratio between the number of constraints and the number of possible constraints, takes five possible values: 0.2, 0.4, 0.6, 0.8 and 1;
- the mean tightness *mt* of the constraints *i.e.*, the mean ratio between the number of forbidden pairs of values and the number of possible pairs of values, takes five possible values: 0.1, 0.3, 0.5, 0.7 and 0.9; for a given value of *mt*, the tightness of each constraint is randomly generated between $mt - 0.1$ and $mt + 0.1$.
- the size *ch* of the changes *i.e.*, the ratio between the number of additions or removals and the number of constraints, takes six possible values: 0.01, 0.02, 0.04, 0.08 and 0.16 et 0.32.

For each of the 25 possible pairs (con, mt), 5 problems were generated. For each of the 125 resulting initial problems and for each of the 6 possible values of *ch*, a sequence of 10 changes was generated, with the same probability for additions and removals.

Measures

In terms of efficiency, the three usual measures were performed: number of *nodes*, number of *constraint checks* and *cpu time*. In terms of stability, the *distance* between two successive solutions *i.e.*, the number of variables which are differently assigned in both solutions, was measured each time both exist.

nv = 15, 6 ≤ dom ≤ 16, ch = 0.04
number of constraint checks

backward checking

mt	0.1			0.3			0.5			0.7			0.9		
con															
0.2	c	bt	12	c	bt	10	c	bt	127	ci	bt	21 954	i	bt	96
		hrp	13		hrp	11		hrp	24		hrp	30 330		hrp	3 862
		dbt	12		dbt	10		**dbt**	23		dbt	2 508		dbt	21
		lc	3		**lc**	4		lc	27		**lc**	248		**lc**	6
0.4	c	bt	32	c	bt	84	ci	bt	21 536	i	bt	788	i	bt	297
		hrp	33		hrp	61		hrp	100 752		hrp	110 240		hrp	10 263
		dbt	32		dbt	45		dbt	11 020		**dbt**	326		dbt	95
		lc	8		**lc**	39		**lc**	6257		lc	471		**lc**	49
0.6	c	bt	56	c	bt	518	i	bt	29 601	i	bt	3 189	i	**bt**	5
		hrp	57		hrp	472		hrp	159 511		hrp	27 617		hrp	2 050
		dbt	56		dbt	104		**dbt**	8 050		**dbt**	802		**dbt**	5
		lc	19		**lc**	89		lc	17 399		lc	941		lc	8
0.8	c	bt	75	c	bt	13 558	i	bt	2 777	i	bt	72	i	bt	8
		hrp	63		hrp	81 291		hrp	125 966		hrp	10 046		hrp	2 161
		dbt	68		**dbt**	5 126		**dbt**	1 235		**dbt**	57		dbt	7
		lc	25		lc	10 591		lc	1 470		lc	131		**lc**	3
1	c	**bt**	0	ci	bt	45 179	i	bt	1 424	i	bt	370	i	bt	16
		hrp	0		hrp	469 701		hrp	110 541		hrp	3 459		hrp	234
		dbt	0		**dbt**	19 265		**dbt**	805		**dbt**	171		dbt	8
		lc	0		lc	132 203		lc	3 500		lc	181		**lc**	7

forward checking

mt	0.1			0.3			0.5			0.7			0.9		
con															
0.2	c	bt	144	c	bt	92	c	bt	104	ci	bt	386	i	bt	24
		hrp	16		**hrp**	15		**hrp**	40		hrp	2 591		hrp	11
		dbt	144		dbt	93		dbt	106		dbt	257		dbt	17
		lc	23		lc	21		lc	45		**lc**	113		**lc**	3
0.4	c	bt	346	c	bt	254	ci	**bt**	1 245	i	bt	260	i	bt	29
		hrp	39		**hrp**	63		hrp	6 732		hrp	2 850		hrp	69
		dbt	346		dbt	261		dbt	1 554		**dbt**	233		dbt	29
		lc	75		lc	104		lc	1 953		lc	275		**lc**	27
0.6	c	bt	548	c	bt	321	i	**bt**	2 336	i	bt	316	i	bt	11
		hrp	70		**hrp**	191		hrp	12 392		hrp	1 253		hrp	348
		dbt	548		dbt	341		dbt	2 749		**dbt**	314		dbt	11
		lc	143		lc	205		lc	5 526		lc	468		**lc**	7
0.8	c	bt	558	c	**bt**	1 379	i	bt	987	i	bt	185	i	bt	15
		hrp	76		hrp	6 791		hrp	6 521		hrp	562		hrp	7
		dbt	564		dbt	1 761		dbt	858		dbt	169		dbt	15
		lc	185		lc	2 081		**lc**	757		**lc**	100		**lc**	2
1	c	**bt**	0	ci	**bt**	8 092	i	bt	1 857	i	bt	279	i	bt	28
		hrp	0		hrp	98 755		hrp	4 772		hrp	746		hrp	53
		dbt	0		dbt	10 573		dbt	1 687		dbt	281		dbt	28
		lc	0		lc	37 891		**lc**	1 373		**lc**	124		**lc**	6

Results

The two tables above show the mean number of *constraint checks*, when solving dynamic problems with changes of intermediate size ($ch = 0.04$). The first one show the results obtained when using *backward-checking*: *bt-cbj-bc*, *dbt-bc*, *hrp-cbj-bc* and *lc-bc*. The second one show the same results obtained when using *forward-checking*: *bt-cbj-fc*, *dbt-fc*, *hrp-cbj-fc* and *lc-fc*.

At the top left corner of each cell, a letter *c* (resp. *i*) means that all the problems are consistent (resp. inconsistent). Two letters (*ci*) mean that some of them are consistent and the others inconsistent. The less (resp. more) constrained problems, with small (resp. large) values for *con* and *mt* *i.e.*, with few loose (resp. many tight) constraints, are in the top left (resp. bottom right) of each table.

Each number is the mean value of a set of 50 results ($5 * 10$ dynamic problems). In each cell, the algorithm(s) which provides the best result is(are) pointed out in bold.

Analysis

As it has been previously observed (Cheeseman, Kanefsky, & Taylor 1991), the hardest problems are neither the least constrained (solution quickly found), nor the most constrained (inconsistency quickly established), but the intermediate ones, for them it is difficult to establish the consistency or the inconsistency.

If we consider the first table, with *backward-checking*, we can see that:

- *hrp* is efficient on the least constrained problems, but inefficient and sometimes very inefficient on the others;
- *dbt* is always better than *bt* and the best one on the intermediate problems;
- *lc* is almost always better than *hrp* and the best one, both on the least constrained problems and the most constrained ones; it is better on loosely connected problems than on the others; it is nevertheless inefficient on intermediate strongly connected problems.

If we consider the second table, with *forward-checking*, the previous lessons must be modified, because *forward-checking* benefits *bt* and *hrp* more than *dbt* and *lc* (the number of constraint checks is roughly divided by 12 for *bt* and *hrp*, by 3 for *dbt* and *lc*):

- *hrp* becomes the best one on the least constrained problems;
- *bt* and *dbt* are the best ones on the intermediate ones;
- *lc* remains the best one on the most constrained ones.

Note that these results might be different in case of n-ary constraints on which *forward-checking* is less efficient.

We do not show any results related to the *cpu time* because number of *constraint checks* and *cpu time* are strongly correlated, in spite of a little overhead for *hrp* and *lc* (around 850 constraint checks per second for *bt* and *dbt*, around 650 for *hrp* and *lc*; this aspect depends widely on the implementation).

More surprising, the four algorithms provide very similar results in terms of *distance* between successive solutions. It seems to be the result of the mechanisms of each algorithm and of the heuristics used to choose the value.

Note finally that, although *hrp* and *lc* provide better results with small changes than with large ones, the results obtained with other change sizes do not modify the previous lessons.

Conclusion

Although other experiments are needed to confirm it, we believe that the proposed method may be very convenient for solving large problems, involving binary and n-ary constraints, often globally underconstrained and subject to frequent and relatively small changes, such as many real scheduling problems.

Acknowlegments

This work was supported both by the French Space Agency (CNES) and the French Ministry of Defence (DGA-DRET) and was done both at ONERA-CERT (France) and at the University of New Hampshire (USA). We are indebted to P.Hubbe, R.Turner and J. Weiner for helping us to improve this paper and to E. Freuder and R. Wallace for useful discussions.

References

Badie, C., and Verfaillie, G. 1989. OSCAR ou Comment Planifier Intelligemment des Missions Spatiales. In *Proc. of the* 9^{th} *International Avignon Workshop.*

Bellicha, A. 1993. Maintenance of Solution in a Dynamic Constraint Satisfaction Problem. In *Proc. of Applications of Artificial Intelligence in Engineering VIII*, 261–274.

Bruynooghe, M. 1981. Solving Combinatorial Search Problems by Intelligent Backtracking. *Information Processing Letters* 12(1):36–39.

Cheeseman, P.; Kanefsky, B.; and Taylor, W. 1991. Where the *really* Hard Problems Are. In *Proc. of the* 12^{th} *IJCAI*, 294–299.

de Kleer, J. 1989. A Comparison of ATMS and CSP Techniques. In *Proc. of the* 11^{th} *IJCAI*, 290–296.

Dechter, R., and Dechter, A. 1988. Belief Maintenance in Dynamic Constraint Networks. In *Proc. of AAAI-88*, 37–42.

Dechter, R. 1990. Enhancement Schemes for Constraint Processing : Backjumping, Learning and Cutset Decomposition. *Artificial Intelligence* 41(3):273–312.

Ghedira, K. 1993. *MASC : une Approche Multi-Agent des Problèmes de Satisfaction de Contraintes.* Thèse de doctorat, ENSAE, Toulouse, France.

Ginsberg, M. 1993. Dynamic Backtracking. *Journal of Artificial Intelligence Research* 1:25–46.

Hentenryck, P. V., and Provost, T. L. 1991. Incremental Search in Constraint Logic Programming. *New Generation Computing* 9:257–275.

Hubbe, P., and Freuder, E. 1992. An Efficient Cross-Product Representation of the Constraint Satisfaction Problem Search Space. In *Proc. of AAAI-92*, 421–427.

Minton, S.; Johnston, M.; Philips, A.; and Laird, P. 1992. Minimizing Conflicts: a Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence* 58:160–205.

Prosser, P. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9(3):268–299.

Schiex, T., and Verfaillie, G. 1993. Nogood Recording for Static and Dynamic CSP. In *Proc. of the* 5^{th} *IEEE International Conference on Tools with Artificial Intelligence.*

Selman, B.; Levesque, H.; and Mitchell, D. 1992. A New Method for Solving Hard Satisfiability Problems. In *Proc. of AAAI-92*, 440–446.