Dynamic Backtracking for 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 the other agents in the framework of a distributed system. The notion of dynamic CSP (DCSP) has been introduced to represent them. In spite of its name, the recent *Dynamic Backtracking* algorithm proposed in (Ginsberg 1993) does not solve dynamic CSPs, but static ones. But its *nogood recording* and *backtracking* mechanisms are very interesting in the DCSP framework. In this paper, we propose an extension of this algorithm which provides the user with explanations in case of inconsistency and allows dynamic CSPs to be dealt with very efficiently. After presenting the *Dynamic Backtracking* algorithm, its extension and how to use it in the DCSP framework, we present and discuss some experimental results.

Dynamic backtracking

In (Ginsberg 1993), Matthew L. Ginsberg proposed a new algorithm for solving CSPs. This algorithm, called *Dynamic Backtracking*, can be viewed as an improvement of the *Backjumping* (Gaschnig 1979), *Graph Based Backjumping* (Dechter 1990) and *Conflict Directed Backjumping* algorithms (Prosser 1993; Schiex & Verfaillie 1993).

Conflict Directed Backjumping records, at each step of the search and for each variable v, the set of previously assigned variables whose assignments have reduced the domain of v. Such sets are called *conflict sets* and the space which is required to record them is $O(n^2)$, where n is the number of variables. Let v be a variable whose current domain is empty, let $E(v)$ be its *conflict set*, let v' be the last variable in $E(v)$ according to the assignment order. In such a situation, the algorithm does not *backtrack* to the variable which precedes v, but *backjumps* to v', unassigning v' and all the variables between v' and v. Experiments showed that this very simple method often avoids much useless search, especially on sparse constraint graphs.

Dynamic Backtracking records the same kind of information, but at a finer level: at each step of the search, for each variable v and each value *val* which has been eliminated from the current domain of v, it records the set of previously assigned variables which are responsible for this elimination. Such sets are called *eliminating explanations* and the space which is required to record them is $O(n^2d)$, where n is the number of variables and d the maximum domain size. The *conflict set* of a variable is the union of the *eliminating explanations* of all its eliminated values. Let v be a variable whose current domain is empty, let $E(v)$ be its *conflict set*, let v' be the last variable in $E(v)$ according to the instantiation order. In such a situation, the algorithm *backtracks* to v', unassigning it, but *not assigning any other variable*. Obviously, all of the *eliminating explanations* which contain v' become unjustified and have to be removed. The corresponding values have to be returned in their corresponding current domains. Moreover, a new *eliminating explanation* has to be created for the value val' previously assigned to v'. This algorithm can be more precisely described as follows:

dbt(*csp*)
  let V be the set of the variables
  return *dbt-variables*$(\emptyset, V)$

dbt-variables(V_1, V_2)
; V_1 is a set of assigned variables
; V_2 is a set of unassigned variables
  if $V_2 = \emptyset$
  then return *success*
  else let v be a variable chosen in V_2
    let d be its current domain
    if *dbt-variable*$(V_1, v, d) =$ *failure*
    then return *dbt-bt-variable*(V_1, V_2, v)
    else return *dbt-variables*$(V_1 \cup \{v\}, V_2 - \{v\})$

dbt-variable(V_1, v, d)
  if $d = \emptyset$
  then return *failure*
  else let *val* be a value chosen in d
    if *dbt-value*$(V_1, v, val) =$ *success*
    then return *success*
    else return *dbt-variable*$(V_1, v, d - \{val\})$

```
dbt-value(V1, v, val)
    assign-variable(v, val)
    let be c = backward-checking(V1, v)
    if c = success
    then return success
    else let V3 be the set of the variables of c
         unassign-variable(v)
         create-eliminating-explanation(v, val, V3 − {v})
         return failure
dbt-bt-variable(V1, V2, v)
    let V3 be the conflict set of v i.e., the union of the
    eliminating explanations of all its eliminated values
    if V3 = ∅
    then return failure
    else let v' be the last variable of V3 in V1
         let val' be its current value
         let V4 be the set of variables following v' in V1
         unassign-variable(v')
         create-eliminating-explanation(v', val', V3 − {v'})
         remove-eliminating-explanations(v', V4 ∪ V2)
         dbt-variables(V1 − {v'}, V2 ∪ {v'})
```

backward-checking(V, v) checks the current value of v against the current values of the variables of V and returns the first failing constraint (if any) or *success* (if none). *create-eliminating-explanation*(v, val, V) records the set V of variables as an explanation for the elimination of the value val from the current domain of the variable v. For all the variables in V, *remove-eliminating-explanations*(v, V) removes the *eliminating explanations* which contain v and returns the corresponding values in their corresponding current domains.

Since only the variable on which one backtracks is unassigned, one can say that the backtracking mechanism of *Dynamic Backtracking* is less destructive and *a priori* more efficient than that of *Conflict Directed Backjumping*. One can also say that it allows the set of previously assigned variables to be reordered or that the search space becomes a graph, as opposed to a tree.

In (Ginsberg & McAllester 1994), the authors propose an extension of this algorithm, called *Partial Order Dynamic Backtracking*, which offers a greater freedom for the choice of the variable on which backtracking occurs.

In (Ginsberg 1993), termination, correctness and completeness of *Dynamic Backtracking* are proven. Some experimental results related to a crossword-puzzle generation program are given. Our own results related to randomly generated CSPs, are contradictory (see *Results*): whereas *Dynamic Backtracking* is far better than *Conflict Directed Backjumping* when using only *Backward-checking*, it is often a little worse when using *Forward-checking* and more generally any filtering method[1].

But the backtracking mechanism included in *Dynamic Backtracking* offers other opportunities, particularly in the framework of *Dynamic Constraint Satisfaction Problems* (DCSP) (Dechter & Dechter 1988). This led us to extend the notion of *eliminating explanations*, taking into account constraints as assumptions. The following sections present this extension and how it is used to produce inconsistency explanations and to deal with dynamic problems.

Extending Dynamic Backtracking

Both *Conflict Directed Backjumping* and *Dynamic Backtracking* are built on the notion of *nogood*, which comes itself from the *TMS* community (Doyle 1979). In the CSP framework, a *nogood* is an assignment of a subset of the CSP variables which is, explicitly or not, forbidden by the CSP.

Definition 1 *Let A be an assignment of a subset of the CSP variables; A is a nogood iff there is no CSP solution which contains A. More precisely, let A be an assignment of a subset V of the CSP variables; A is a nogood iff there is no solution S such that $S\downarrow_V = A$*[2].

One can say that a *nogood* is an n-ary constraint induced by the CSP.

The following property allows the notions of *nogood* and of *eliminating explanation* to be connected.

Property 1 *Let A be an assignment of a subset V of the CSP variables, let v be any variable in V and val be its value in A; A is a nogood iff $A\downarrow_{V-\{v\}} \rightarrow v \neq val$.*

Both algorithms use the same two theorems to produce *nogoods* during the search:

Theorem 1 *Let c be a constraint and A be an assignment of the variables of c; if the assignment A violates the constraint c, then A is a nogood.*

Theorem 2 *Let v be a variable and $val_1, \ldots, val_i, \ldots, val_d$ its possible values; let $A_1, \ldots, A_i, \ldots, A_d$ be assignments of subsets of the CSP variables (not including v); if*

$$A_1 \cup \{(v, val_1)\}, \ldots, A_i \cup \{(v, val_i)\}, \ldots, A_d \cup \{(v, val_d)\}$$

are all nogoods, then

$$A_1 \cup \ldots \cup A_i \cup \ldots \cup A_d$$

is also a nogood.

Note that Theorem 2 may produce an empty *nogood*, which means that the CSP is inconsistent (there is no solution at all). Note also that it forms the operational basis of k-consistency enforcing and that it is used in another form in the *ATMS* framework.

This *nogood* definition and the corresponding theorems are sufficient for dealing with *static* CSPs, but not with *dynamic* ones. In the former case, constraints are implicit and can remain implicit, since they do not evolve. In the latter, they have to become explicit assumptions at the same level as assignments.

The previous definition and theorems can be easily extended:

[1] On the problems we generated, *Forward-checking* benefits *Conflict Directed Backjumping* four times more than *Dynamic Backtracking*.

[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'.

Definition 2 *Let A be an assignment of a subset of the CSP variables and C be a subset of the CSP variables; (A, C) is a nogood iff there is no solution of the CSP restricted* [3] *to C which contains A.*

If (A, C) is a *nogood*, one can say that A is an n-ary constraint induced by C.

Theorem 3 *Let c be a constraint and A be an assignment of the variables of c; if the assignment A violates the constraint c, then $(A, \{c\})$ is a nogood.*

Theorem 4 *Let v be a variable and $val_1, \ldots, val_i, \ldots, val_d$ its possible values; let $A_1, \ldots, A_i, \ldots, A_d$ be assignments of subsets of the CSP variables (not including v); let $C_1, \ldots, C_i, \ldots, C_d$ be subsets of the CSP constraints; if*

$$(A_1 \cup \{(v, val_1)\}, C_1), \ldots (A_d \cup \{(v, val_d)\}, C_d)$$

are all nogoods, then

$$(A_1 \cup \ldots \cup A_i \cup \ldots \cup A_d, C_1 \cup \ldots \cup C_i \cup \ldots \cup C_d)$$

is also a nogood.

Note that, as Theorem 2, Theorem 4 may produce a *nogood* of the form $(\emptyset, C)$, which is composed of an empty assignment and a set of constraints, and means that the subset C of the CSP constraints makes the CSP inconsistent.

Dynamic Backtracking can be easily extended to take into account constraints in *eliminating explanations*. The same kind of extension on the basis of *Conflict Directed Backjumping* is described in (Schiex & Verfaillie 1993). Only the last two procedures *dbt-value* and *dbt-bt-variable* have to be modified as follows:

```
dbt-value(V1, v, val)
    assign-variable(v, val)
    let be c = backward-checking(V1, v)
    if c = success
    then return success
    else let V3 be the set of the variables of c
         unassign-variable(v)
         create-eliminating-explanation(v, val, V3 − {v}, {c})
         return failure

dbt-bt-variable(V1, V2, v)
    let V3 be the union
    of all the variable eliminating explanations of v
    let C be the union
    of all the constraint eliminating explanations of v
    if V3 = ∅
    then return C
    else let v' be the last variable of V3 in V1
         let val' be its current value
         let V4 be the set of variables following v' in V1
         unassign-variable(v')
         create-eliminating-explanation(v', val', V3 − {v'}, C)
         remove-eliminating-explanations(v', V4 ∪ V2)
         dbt-variables(V1 − {v'}, V2 ∪ {v'})
```

[3] Let (V, C) be a CSP where V is the set of variables and C the set of constraints, let C' a subset of C; the restriction of the CSP to C' is the CSP (V, C').

The *eliminating explanation* of a value of a variable is now composed of two parts: a set of previously assigned variables (the *variable eliminating explanation*) and a set of constraints (the *constraint eliminating explanation*). Note that the space which is required to record them is $O(nd(n+m))$, where n is the number of variables, m the number of constraints and d the maximum domain size.

create-eliminating-explanation(v, val, V, C) records the set V of variables and the set C of constraints as an explanation for the elimination of the value *val* from the current domain of the variable v.

Inconsistency explanations

The first benefit of this extension is the ability to provide the user (or another program) with an explanation in case of inconsistency: a set of constraints which implies the inconsistency. Although the produced explanation depends on the variable assignment and constraint check orders and there is no guarantee that it is minimal with regard to inclusion, it can be practically very useful:

- inconsistency can be straightforwardly established for any other CSP which contains this set of constraints;
- it is necessary to remove at least one constraint from this set if one wants to reach consistency.

Dynamic problems

In many real applications, the set of variables and constraints which compose the CSP can evolve because of the *environment* (planning and scheduling applications) or the *user* (interactive design) or, more generally, because of 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. Since 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, a DCSP is just a sequence of CSPs, where each one differs from the previous one by some constraints either added or removed.

To solve such a sequence of CSPs, it is always and obviously possible to solve each CSP from scratch, using the usual algorithms. But this naive method has two important drawbacks:

- *inefficiency*, which can become a problem in the framework of real time applications, where the time allowed for solving the new problem is limited;
- possible *instability* of the successive solutions, which can be costly if some work has been started on the basis of the previous solution.

Several methods have been proposed in the last few years. Among them:

- *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 one variable assignment) (Minton *et al.* 1992; Selman, Levesque, & Mitchell 1992; Selman & Kautz 1993; Verfaillie 1993; Verfaillie & Schiex 1994; 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(Hentenryck & Provost 1991; Schiex & Verfaillie 1993).

Other methods aim at minimizing the distance between two successive solutions (Bellicha 1993).

At the border between the *local repair* and *constraint recording* methods, *Dynamic Backtracking* offers a new opportunity to deal with dynamic CSPs. Indeed solving a CSP by using *Dynamic Backtracking* produces:

- when the CSP is consistent:
  - a *solution* , *i.e.*, a complete and consistent assignment;
  - a set of *eliminating explanations* related to some of the possible variable assignments.
- when it is inconsistent:
  - a partial and consistent assignment, which can be empty;
  - an *inconsistency explanation*;
  - a set of *eliminating explanations* related to some of the possible variable assignments.

It should be interesting to record these results and to reuse them when solving a new CSP which differs from the previous one only by some constraints, either added or removed. On one hand, we can expect that most of the recorded explanations (*eliminating explanations* and eventual *inconsistency explanation*) remain valid in the framework of the new CSP. On the other hand, we can expect that the non-destructive backtracking mechanism of *Dynamic Backtracking* helps us to produce a solution of the new CSP which is close to the partial or complete assignment produced in the framework of the previous CSP. The explanation recording should allow the new search space to be significantly pruned. The non destructive mechanism should allow us to satisfy both the efficiency and stability requirements.

Exploiting these ideas, a *Dynamic Backtracking* algorithm for dynamic CSPs can be described as follows:

```
ddbt(V1, V2, AC, RC)
; V1 is the ordered set of assigned variables,
; result of the previous search
; V2 is the set of unassigned variables,
; result of the same search
; AC is the set of added constraints
; RC the set of removed constraints
   let be V = V1 ∪ V2
   let be V3 = remove-assignments(V1, AC)
   remove-variable-eliminating-explanations(V3, V)
   remove-constraint-eliminating-explanations(RC, V)
   return dbt-variables(V1 − V3, V2 ∪ V3)
```

The three procedures *remove-assignments*, *remove-variable-eliminating-explanations* and *remove-constraint-eliminating-explanations* aim at deriving a consistent starting point for the new search (procedure *dbt-variables*) from the result of the previous one.

For each constraint c in AC, let $V(c)$ be the set of its variables; if the assignment of the variables of V_1 violates c, *remove-assignments*(V_1, AC) unassigns the last variable of $V(c)$ in V_1 and creates the corresponding *elimiminating explanation*; it returns the set V_3 of the variables which have been unassigned.

For all the variables in V, *remove-variable-eliminating-explanations*(V_3, V) removes the *eliminating explanations* whose *variable eliminating explanation* intersects with V_3 and *remove-constraint-eliminating-explanations*(RC, V) the *eliminating explanations* whose *constraint eliminating explanation* intersects with RC.

Experiments

Algorithms

Three algorithms were used in our experiments:

- *cbj* (for *Conflict Directed Backjumping*) and *dbt* (for *Dynamic Backtracking*), which work as usual, except that the result of each search (consistency or inconsistency) is recorded, as well as the first solution found (in case of consistency) or the produced inconsistency explanation (in case of inconsistency); each new search starts from scratch (as the first one), except two cases:
  - the previous CSP is consistent and all the added constraints are satisfied by the previous solution; this solution remains valid;
  - the previous CSP is inconsistent and there is no removed constraint which belongs to the previous inconsistency explanation; this inconsistency explanation remains valid;
- *ddbt* (for *Dynamic Dynamic Backtracking*), which works as described in the previous section.

Heuristics

The same heuristics were used in the three cases:

- choice of the *variable* to be assigned: choose the variable whose current domain is the smallest;

- choice of the *value*: first use the value the variable had in the previous solution, if it exists.

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 $(nv(nv-1)/2)$, 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 (*i.e.*, the product of the cardinals of both domains), 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 constraints added or removed and the initial 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 starting 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. This allows the 10 successive problems to have approximatively the same characteristics (connectivity, tigthness) as the initial one. All this generation process provides us with a significant set of $125 * 6 * 10 = 7500$ dynamic problems.

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, that is the number of variables which get different values in both solutions, was measured, each time both exist.

Results

The following results are related to dynamic problems with different change sizes: 0.02, 0.08 and 0.32. The three algorithms: *cbj*, *dbt* and *ddbt* have been used with *forward-checking*.

The first three tables show the mean number of *constraint checks*. The last three ones show the mean *distance* between successive solutions.

At the top left corner of each cell, a letter c (resp i) at the right corner 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.

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

If we consider the first three tables related to the number of *constraint checks*, we can see that:

- the results of *cbj* and *dbt* are very close, with a little advantage for *cbj*;
- in 75% of the cases, *ddbt* is better than *cbj* and *dbt*;
- *ddbt* is systematically the best one, both on the least constrained problems and on the most constrained ones; it can become worse on the intermediate ones (the hardest ones to be solved);
- the results of *ddbt* are better on dynamic problems with small and intermediate changes ($ch = 0.02$ or 0.08) than with large changes ($ch = 0.32$).

We do not show any results related to the *cpu time*, because they would not modify the previous lessons. Number of *constraint checks* and *cpu time* are strongly correlated, in spite of an overhead of around 10% for *dbt* and *ddbt*

If we consider now the last three tables related to the *distance* between successive solutions, the lessons are fairly similar:

- the results of *cbj* and *dbt* are most of the time equal; when they are different, there is a little advantage for *cbj*;
- in nearly 90% of the cases, *ddbt* is better than or as good as *cbj* and *dbt*;
- *ddbt* is systematically the best one or one of the best ones on the least constrained problems; it can become worse on the intermediate ones;
- the results of *ddbt* are better on dynamic problems with small and intermediate changes than with large changes.

nv = 15, $6 \leq$ dom ≤ 16
forward-checking, number of constraint checks

ch = 0.02

mt	0.1			0.3			0.5			0.7			0.9		
con															
0.2	c	cbj	6	c	cbj	40	c	cbj	46	ci	**cbj**	107	i	cbj	32
		dbt	6		dbt	40		dbt	47		dbt	124		dbt	26
		ddbt	5		**ddbt**	4		**ddbt**	8		ddbt	412		**ddbt**	13
0.4	c	cbj	10	c	cbj	22	ci	**cbj**	243	i	cbj	219	i	cbj	18
		dbt	10		dbt	22		dbt	281		dbt	163		dbt	18
		ddbt	5		**ddbt**	2		ddbt	270		**ddbt**	114		**ddbt**	10
0.6	c	cbj	49	c	cbj	48	i	cbj	2 049	i	cbj	169	i	cbj	8
		dbt	49		dbt	50		dbt	2 107		dbt	168		dbt	8
		ddbt	4		**ddbt**	13		**ddbt**	1 091		**ddbt**	71		**ddbt**	2
0.8	c	cbj	77	c	**cbj**	623	i	cbj	1 182	i	cbj	99	i	cbj	10
		dbt	77		dbt	1 022		dbt	1 191		dbt	100		dbt	10
		ddbt	6		ddbt	1 013		**ddbt**	572		**ddbt**	38		**ddbt**	2
1	c	**cbj**	1	ci	**cbj**	7 751	i	cbj	1 480	i	cbj	69	i	cbj	13
		dbt	1		dbt	10 137		dbt	1 457		dbt	69		dbt	13
		ddbt	5		ddbt	8 642		**ddbt**	470		**ddbt**	15		**ddbt**	8

ch = 0.08

mt	0.1			0.3			0.5			0.7			0.9		
con															
0.2	c	cbj	35	c	cbj	71	c	cbj	57	ci	cbj	394	i	cbj	61
		dbt	35		dbt	71		dbt	56		dbt	399		dbt	52
		ddbt	11		**ddbt**	8		**ddbt**	18		**ddbt**	319		**ddbt**	36
0.4	c	cbj	55	c	cbj	103	ci	**cbj**	503	i	cbj	320	i	cbj	19
		dbt	55		dbt	103		dbt	821		dbt	215		dbt	19
		ddbt	13		**ddbt**	10		ddbt	553		**ddbt**	136		**ddbt**	11
0.6	c	cbj	66	c	cbj	288	ci	cbj	2 841	i	cbj	458	i	cbj	29
		dbt	66		dbt	382		dbt	3 206		dbt	490		dbt	29
		ddbt	19		**ddbt**	242		**ddbt**	2 636		**ddbt**	178		**ddbt**	8
0.8	c	cbj	160	c	cbj	434	i	cbj	1 630	i	cbj	186	i	cbj	37
		dbt	160		dbt	531		dbt	1 927		dbt	186		dbt	37
		ddbt	27		**ddbt**	387		**ddbt**	1 319		**ddbt**	74		**ddbt**	16
1	c	**cbj**	3	ci	**cbj**	4 962	i	cbj	2 520	i	cbj	321	i	cbj	45
		dbt	3		dbt	6 638		dbt	2 592		dbt	324		dbt	45
		ddbt	24		ddbt	6 061		**ddbt**	1 565		**ddbt**	100		**ddbt**	20

ch = 0.32

mt	0.1			0.3			0.5			0.7			0.9		
con															
0.2	c	**cbj**	36	c	cbj	92	ci	**cbj**	281	ci	cbj	1 566	ci	cbj	106
		dbt	36		dbt	92		dbt	324		**dbt**	862		dbt	77
		ddbt	37		**ddbt**	31		ddbt	424		ddbt	988		**ddbt**	61
0.4	c	cbj	256	c	cbj	227	ci	**cbj**	2 815	ci	cbj	501	i	cbj	129
		dbt	256		dbt	228		dbt	3 259		dbt	465		dbt	108
		ddbt	68		**ddbt**	66		ddbt	3 067		**ddbt**	366		**ddbt**	62
0.6	c	cbj	147	c	**cbj**	614	ci	**cbj**	1 632	i	cbj	695	ci	cbj	90
		dbt	147		dbt	826		dbt	1 934		dbt	697		dbt	80
		ddbt	89		ddbt	885		ddbt	1 805		**ddbt**	423		**ddbt**	54
0.8	c	cbj	239	ci	cbj	3 213	ci	**cbj**	2 209	ci	cbj	564	i	cbj	131
		dbt	239		**dbt**	1 873		dbt	2 518		dbt	528		dbt	130
		ddbt	98		ddbt	4 294		ddbt	2 385		**ddbt**	362		**ddbt**	61
1	c	**cbj**	12	ci	**cbj**	4 593	ci	**cbj**	3 782	ci	cbj	758	ci	cbj	199
		dbt	12		dbt	6 509		dbt	3 947		dbt	802		dbt	186
		ddbt	107		ddbt	6 514		ddbt	4 072		**ddbt**	656		**ddbt**	130

nv = 15, 6 ≤ dom ≤ 16
forward-checking, distance between successive solutions

ch = 0.02

mt	0.1			0.3			0.5			0.7			0.9		
con															
0.2	c	**cbj**	0.02	c	cbj	0.28	c	cbj	0.36	ci	**cbj**	1.46	i		
		dbt	0.02		dbt	0.28		dbt	0.36		**dbt**	1.46			
		ddbt	0.02		**ddbt**	0.18		**ddbt**	0.34		ddbt	1.77			
0.4	c	cbj	0.04	c	cbj	0.18	ci	cbj	1.02	i			i		
		dbt	0.04		dbt	0.18		**dbt**	0.92						
		ddbt	0.02		**ddbt**	0.08		ddbt	1.12						
0.6	c	cbj	0.16	c	cbj	0.42	i			i			i		
		dbt	0.16		dbt	0.50									
		ddbt	0.08		**ddbt**	0.32									
0.8	c	cbj	0.20	c	cbj	0.70	i			i			i		
		dbt	0.20		dbt	0.70									
		ddbt	0.14		**ddbt**	0.68									
1	c	**cbj**	0.00	ci	**cbj**	0.00	i			i			i		
		dbt	0.00		**dbt**	0.00									
		ddbt	0.00		**ddbt**	0.00									

ch = 0.08

mt	0.1			0.3			0.5			0.7			0.9		
con															
0.2	c	cbj	0.18	c	cbj	0.48	c	cbj	0.90	ci	cbj	2.96	i		
		dbt	0.18		dbt	0.48		dbt	0.90		dbt	2.93			
		ddbt	0.14		**ddbt**	0.34		**ddbt**	0.70		**ddbt**	1.59			
0.4	c	cbj	0.14	c	cbj	0.92	ci	cbj	1.75	i			i		
		dbt	0.14		dbt	0.92		dbt	1.83						
		ddbt	0.12		**ddbt**	0.34		**ddbt**	1.50						
0.6	c	cbj	0.24	c	cbj	1.60	ci			i			i		
		dbt	0.24		dbt	1.94									
		ddbt	0.14		**ddbt**	1.52									
0.8	c	cbj	0.52	c	**cbj**	1.78	i			i			i		
		dbt	0.52		dbt	1.90									
		ddbt	0.26		ddbt	1.98									
1	c	**cbj**	0.00	ci	cbj	2.59	i			i			i		
		dbt	0.00		dbt	2.63									
		ddbt	0.00		**ddbt**	2.26									

ch = 0.32

mt	0.1			0.3			0.5			0.7			0.9		
con															
0.2	c	**cbj**	0.20	c	cbj	1.10	ci	cbj	3.50	ci	cbj	4.04	ci	**cbj**	1.00
		dbt	0.20		dbt	1.10		dbt	3.50		dbt	4.07		**dbt**	1.00
		ddbt	0.20		**ddbt**	0.82		**ddbt**	2.65		**ddbt**	3.44		**ddbt**	1.00
0.4	c	cbj	0.66	c	cbj	3.04	ci	**cbj**	6.80	ci	**cbj**	3.71	i		
		dbt	0.66		dbt	3.04		dbt	6.80		dbt	5.43			
		ddbt	0.54		**ddbt**	1.68		ddbt	7.47		ddbt	4.43			
0.6	c	cbj	0.46	c	cbj	4.28	ci	cbj	5.89	i			ci		
		dbt	0.46		dbt	4.66		dbt	5.95						
		ddbt	0.32		**ddbt**	3.70		**ddbt**	5.79						
0.8	c	cbj	0.90	ci	cbj	4.27	ci	cbj	12.25	ci			i		
		dbt	0.90		**dbt**	3.31		dbt	12.25						
		ddbt	0.54		ddbt	4.02		**ddbt**	10.50						
1	c	**cbj**	0.00	ci	cbj	2.93	ci	**cbj**	6.00	ci	cbj	13.00	ci	**cbj**	0.00
		dbt	0.00		dbt	3.93		dbt	11.00		dbt	13.00		**dbt**	0.00
		ddbt	0.00		**ddbt**	2.86		**ddbt**	6.00		**ddbt**	11.00		**ddbt**	0.00

Conclusion

In spite of contradictory results in the framework of static problems, we showed that a simple extension of *Dynamic Backtracking* is very convenient for dealing with dynamic problems. Systematic experiments show that it provides good results both in terms of efficiency (number of *constraint checks* and *cpu time*) and stability (*distance* between successive solutions).

This is not surprising if we consider that solving a CSP by using any backtracking mechanism provides a kind of dymamic CSP: *forward* moves add constraints (new variable assignments) and *backward* moves remove constraints (previous assignments).

It would be nice to unify all these changes in the same framework. One of the first results would be an algorithm able to take into account any change at any time, during or after the search.

Acknowlegments

This work was done at the University of New Hampshire (USA) and was supported the French Ministry of Defence (DGA-DRET) . Thanks to R.Turner for helping us to improve this paper and to E. Freuder, M. Ginsberg and D. McAllester for useful discussions.

References

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

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.

Doyle, J. 1979. A Truth Maintenance System. *Artificial Intelligence* 12:231–272.

Gaschnig, J. 1979. *Performance Measurement and Analysis of certain Search Algorithms.* Ph.D. Dissertation, Carnegie-Mellon University.

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

Ginsberg, M., and McAllester, D. 1994. GSAT and Dynamic Backtracking. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming.*

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 5th IEEE International Conference on Tools with Artificial Intelligence.*

Selman, B., and Kautz, H. 1993. Domain Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proc. of the 13th IJCAI*, 290–295.

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

Verfaillie, G., and Schiex, T. 1994. Solution Reuse in Dynamic Constraint Satisfaction Problems. In *Proc. of AAAI-94.*

Verfaillie, G. 1993. Problèmes de Satisfaction de Contraintes : Production et Révision de Solution par Modifications Locales. In *Proc. of the 13th International Avignon Workshop.*