

Maintaining Arc-Consistency within Dynamic Backtracking

Narendra Jussien, Romuald Debruyne, and Patrice Boizumault

École des Mines de Nantes
4 rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3
{jussien,rdebruyne,boizu}@emn.fr

Abstract. Most of complete search algorithms over Constraint Satisfaction Problems (CSP) are based on *Standard Backtracking*. Two main enhancements of this basic scheme have been studied: first, to integrate constraint propagation as **mac** which maintains arc consistency during search; second, intelligent backtrackers which avoid repeatedly falling in the same dead-ends by recording nogoods as *Conflict-directed BackJumping* (**cbj**) or *Dynamic Backtracking* (**dbt**). Integrations of constraint propagation within intelligent backtrackers have been done as **mac-cbj** which maintains arc consistency in **cbj**. However, Bessière and Régin have shown that **mac-cbj** was very rarely better than **mac**. However, the inadequacy of **mac-cbj** is more related to the fact that **cbj** does not avoid thrashing than to the cost of the management of nogoods.

This paper describes and evaluates **mac-dbt** which maintains arc-consistency in **dbt**. Experiments show that **mac-dbt** is able to solve very large problems and that it remains very stable as the size of the problems increases. Moreover, **mac-dbt** outperforms **mac** on the structured problems we have randomly generated.

1 Introduction

Most of complete search algorithms over Constraint Satisfaction Problems (CSP) are based on *Standard Backtracking* (**sb**): a depth-first search is performed using chronological backtracking. Various *intelligent* backtrackers have been proposed: *Conflict-directed BackJumping* (**cbj**) [16], *Dynamic Backtracking* (**dbt**) [10], *Partial order Dynamic Backtracking* (**pdb**) [11], *Generalized Dynamic Backtracking* (**gpb**) [6], etc. In those algorithms, information (namely nogoods) is kept when encountering inconsistencies so that the forthcoming search will not get back to already known traps in the search space.

Constraint propagation has been included in **sb** leading to forward checking **fc** and more recently to the *Maintaining Arc-Consistency* algorithm (**mac**) [18]. **mac** is nowadays considered as one of the best algorithms for solving CSP[5].

Several attempts to integrate constraint propagation within intelligent backtrackers have been done: for example, Prosser has proposed **mac-cbj** which maintains arc consistency in **cbj** [16]. But, Bessière and Régin [5] have stopped further

research in that field by showing that **mac-cbj** was very rarely better than **mac**. They concluded that there was no need to spend time nor space for intelligent backtracking because the brute force of **mac** simply does it more quickly.

From our point of view, the inadequacy of **mac-cbj** is more related to the fact that **cbj** does not avoid thrashing¹ than to the cost of the management of nogoods. When backtracking occurs, **cbj** comes back to a relevant assignment, and then forgets all the search space developed since this assignment has been performed: as **sb**, **cbj** has a **multiplicative** behavior on independent sub-problems. **dbt** does not only use nogoods to perform *intelligent* backtracking but also to avoid thrashing and so becomes **additive** on independent sub-problems [10].

[5] had another point preventing the use of nogoods: it is always possible to find an intelligent labeling heuristic so that a *standard backtracking*-based algorithm will perform a search as efficiently as an intelligent backtracker. In our experience, using a good heuristic reduces the number of problems on which the algorithm thrashes but does not make it additive on independent subproblems: there are still problems on which the heuristic cannot prevent thrashing.

Although many works have been done about **dbt**, nothing, as far as we know, has ever been published on maintaining arc consistency in **dbt**. Even *Forward Checking* and *Dynamic Backtracking* (**fc+dbt**) has never been fully described [21].

The aim of this paper is to describe and evaluate **mac-dbt** which maintains arc-consistency in **dbt**. We first recall the principles of **dbt** and then describe how to integrate constraint propagation. Then we experiment **mac-dbt** and compare it to **dbt**, **fc-dbt** and **mac**. These experiments show that **mac-dbt** is able to solve very large problems and that it remains very stable as the size of the problems arise. Furthermore, **mac-dbt** outperforms **mac** on the structured problems we have randomly generated.

2 Improving Standard Backtracking

To increase the search efficiency, intelligent backtrackers store for each dead-end a **nogood**, namely a subset of assignments responsible of the dead-end. Recording this information avoids falling repeatedly in the same dead-ends. *Dependency Directed Backtracking* (**ddb**) [20] was the first algorithm to use this enhancement, however it has an important drawback: its space complexity is exponential since the number of nogoods it stores increases monotonically.

To address this problem, algorithms such as **cbj** and **dbt** eliminate nogoods that are no longer relevant to the current variable assignment. By doing so, the space complexity remains polynomial.

¹ A thrashing behavior consists in repeatedly performing the same search work due to the backtrack mechanism.

2.1 Nogoods and eliminating explanations

Let consider a CSP (V, D, C) . A **nogood** is a globally inconsistent partial assignment of values a_i to variables v_i (no solution can contain a nogood)²:

$$C \vdash \neg (v_1 = a_1 \wedge \dots \wedge v_k = a_k) \quad (1)$$

For every nogood, a variable v_j can be selected and the previous formula rewritten as:

$$C \vdash \bigwedge_{i \in [1..k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j \quad (2)$$

The left hand side of the implication constitutes an **eliminating explanation** for the removal of value a_j from the domain of variable v_j (noted $\text{expl}(v_j \neq a_j)$).

When the domain of variable v_j becomes empty during filtering, a new nogood is deduced from the eliminating explanations of all its removed values:

$$C \vdash \neg \left(\bigwedge_{a \in d(v_j)} \text{expl}(v_j \neq a) \right) \quad (3)$$

There generally exist several eliminating explanations for a value removal. One may want to record all of them as in **ddb** but as we saw this leads to an exponential space complexity. Another way relies in *forgetting* (erasing) nogoods that are no longer relevant³ to the current variable assignment. By doing so, the space complexity remains polynomial. **dbt** (and its extensions **pdb**, **gpb**) therefore records only **one** explanation at a time for a value removal. In the worst case, the space required to manage nogoods is $O(n^2d)$ where n is the number of variables and d the maximum size of the domains in the CSP. Indeed, the size of each eliminating explanation is at most $(n - 1)$ and there are at most $n \times d$ eliminating explanations: one for each value of each domain.

2.2 From Standard to Dynamic Backtracking

When a failure occurs, **sb**, **cbj** and **dbt** have to identify the assignment to be reconsidered (suspected to be a **culprit** for the failure).

sb always considers the most recent assignment to be a culprit. This selection may be completely irrelevant for the current failure leading to unuseful exploration of parts of the search tree already known to be dead-ends.

cbj stores the nogoods but not like in **dbt**. In **cbj** a conflict set is associated to each variable: CS_{v_i} (for the variable v_i) contains the set of the assigned variables whose value is in conflict with the value of v_i . When identifying a dead-end while assigning v_i , **cbj** considers the most recent variable in CS_{v_i} to be a culprit. But

² The *nogood* is a logical consequence of the set of constraints C .

³ A nogood is said to be relevant if all the assignments in it are still valid in the current search state [3].

as opposed to **dbt**, with **cbj** a backtrack occurs: the conflict sets and domains of the future variables are reset to their original value. By doing so, **cbj** forgets a lot of information that could have been useful. This leads to thrashing.

dbt, similarly to **cbj**, selects the most recent variable in the computed nogood (the conflict set of **cbj**) in order to undo the assignment. However, thanks to the eliminating explanations, **dbt** only removes related information that depends on it and so avoids thrashing: useful information is kept. Indeed, there is no real backtracking in **dbt** and like in a repair method, only the assignments that caused the contradiction are undone.

Consequently, **sb** and **cbj** have both a multiplicative behavior on independent sub-problems while **dbt** is additive.

Note that **sb** can also be considered as selecting the most recent assignment of a nogood, namely the nogood that contains all the current variable assignments (which fails to give really relevant information).

2.3 Dynamic Backtracking

In the remaining, dom_i is the initial domain of the variable i and D_i is the current domain of this variable. The algorithm **Dynamic Backtracking** is presented in fig. 1.

Function **dbt** performs the main loop which tries to assign values to variables as long as a complete consistent assignment has not been found. \mathcal{V} will denote the set of variables to be assigned and I is the current instantiation. Function **dbt-giveValueAndCheck**(I, \mathcal{V}, i, a) determines if the new partial assignment (including the new assignment $i = a$) is consistent; if not, this function returns a nogood explaining the failure. In order to restore a coherent state of computation, the function **dbt-handleContradiction** jumps to another consistent partial assignment. Domains and nogoods are restored thanks to the eliminating explanations (see function **dbt-updateDomains**).

dbt-checkConstraintsBackwards checks backwards whether the constraints are verified for the new current partial assignment. If not, this function returns such a failing constraint. From that constraint, **dbt-giveValueAndCheck** computes a nogood (line 8). This nogood contains only the assignments involved in the failure.

Function **dbt-handleContradiction** is the contradiction handling mechanism. The assignment to be undone is determined on line 2 and *backtracking* (or more exactly *jumping*) is achieved by removing irrelevant nogoods which is performed by the **dbt-updateDomains** function.

In fact, **dbt** does not perform real backtracks. When a dead-end occurs, it reconsiders only the most recent assignment that caused the contradiction. Especially, all the assignments that not caused the dead-end remain unchanged. This is why **dbt** has an additive behavior on independent sub-problems.

```

function dbt()
1   $I \leftarrow \emptyset$ ;
2  while  $\mathcal{V} \neq \emptyset$  do
3     $(i, a) \leftarrow \text{chooseAssignment}(\mathcal{V}, \mathcal{D})$ ;
4     $E \leftarrow \text{dbt-giveValueAndCheck}(I, \mathcal{V}, i, a)$ ;
5    if  $E$  is not a success then  $\text{dbt-handleContradiction}(E, I, \mathcal{V})$ 
6  return  $I$ ;

function dbt-giveValueAndCheck( $I, \mathcal{V}, i, a$ )
1   $C_i = \text{constraint}(i = a)$ ;  $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_i\}$ ;
2  foreach  $b \in D_i$  s.t.  $b \neq a$  do
3     $\text{expl}(i \neq b) \leftarrow \{C_i\}$ ;
4   $D_i \leftarrow \{a\}$ ;
5   $I \leftarrow I \cup \{(i, a)\}$ ;  $\mathcal{V} \leftarrow \mathcal{V} \setminus \{i\}$ ;
6   $c \leftarrow \text{dbt-checkConstraintsBackwards}(I)$ ;
7  if  $c$  is a success then return success
8  else return  $\{C_i\} \cup \{C_k | k \in \text{vars}(c)\}$ 

function dbt-checkConstraintsBackwards( $I$ )
1  foreach  $c \in \mathcal{C}$  s.t.  $(\text{vars}(c) \cap \mathcal{V} = \emptyset)$  do
2    if  $c$  is not verified then return  $c$ ;
3  return success;

function dbt-handleContradiction( $E, I, \mathcal{V}$ )
1  if  $E = \emptyset$  then fail
2   $C_j \leftarrow \text{mostRecentCulprit}(E)$ ;  $b \leftarrow I[j]$ ;
3   $\text{dbt-updateDomains}(\{(k, c) | C_j \in \text{expl}(k \neq c)\})$ ;
4   $I \leftarrow I \setminus \{(j, b)\}$ ;  $\mathcal{V} \leftarrow \mathcal{V} \cup \{j\}$ ;
5   $D_j \leftarrow D_j \setminus \{b\}$ ;  $\text{expl}(j \neq b) \leftarrow E \setminus \{C_j\}$ ;
6  if  $D_j = \emptyset$  then  $\text{dbt-handleContradiction}(\bigcup_{a \in \text{dom}_j} \text{expl}(j \neq a), I, \mathcal{V})$ ;

function dbt-updateDomains( $Back$ )
1  foreach  $(i, a) \in \text{back}$  do
2     $\text{expl}(i \neq a) \leftarrow \emptyset$ ;  $D_i \leftarrow D_i \cup \{a\}$ ;

```

Fig. 1. Dynamic Backtracking

3 Integrating constraint propagation

Integrating constraint propagation in dbt is more complex than integrating forward checking [19].

First, when a failure occurs, computing nogoods as before (the variable assignments in the failing constraint) will not even provide a nogood. Effects of propagation (value removals) have to be taken into account: eliminating explanations produced by the filtering algorithm need to be kept.

Second, another problem arises when undoing a variable assignment. Putting back in the domains values with irrelevant explanations will not be sufficient since there may exist another relevant explanation for the deleted value. Indeed, there may exist several ways of removing a value through propagation and since only one way is retained as an explanation, any value restoration need to be confirmed by the propagation algorithm. This is similar to what is done for maintaining arc-consistency in dynamic CSPs [4]. A proof of termination can be found in [12].

```

function mac-dbt()
1  if (AC4() = false) then return  $\emptyset$ ;
2   $I \leftarrow \emptyset$ ;  $Q \leftarrow \emptyset$ ;
3  while  $\mathcal{V} \neq \emptyset$  do
4     $(i, a) \leftarrow \text{chooseAssignment}(\mathcal{V}, \mathcal{D})$ ;
5     $E \leftarrow \text{mac-dbt-giveValueAndCheck}(I, \mathcal{V}, i, a, Q)$ ;
6    if  $E$  is not a success then mac-dbt-handleContradiction( $E, I, \mathcal{V}, Q$ )
7  return  $I$ ;

function mac-dbt-handleContradiction( $E, I, \mathcal{V}, Q$ )
1  if  $E = \emptyset$  then fail
2   $C_j \leftarrow \text{mostRecentCulprit}(E)$ ;  $b \leftarrow I[j]$ ;
3   $E' \leftarrow \text{mac-dbt-updateDomains}(\{(k, c) | C_j \in \text{expl}(k \neq c)\}, Q)$ ;
4   $I \leftarrow I \setminus \{(j, b)\}$ ;  $\mathcal{V} \leftarrow \mathcal{V} \cup \{j\}$ ;  $\mathcal{C} \leftarrow \mathcal{C} \setminus C_j$ ;
5  if  $E'$  is not a success then mac-dbt-handleContradiction( $E', I, \mathcal{V}, Q$ );
6  if  $(E \setminus \{C_j\}) \subseteq \mathcal{C}$  then
7     $\text{expl}(j \neq b) \leftarrow E \setminus \{C_j\}$ ;  $D_j \leftarrow D_j \setminus \{b\}$ ;
8     $Q \leftarrow Q \cup \{(j, b)\}$ ;
9     $E' \leftarrow \text{mac-dbt-propagSuppress}(Q)$ ;
10   if  $E'$  is not a success then mac-dbt-handleContradiction( $E', I, \mathcal{V}, Q$ );

function mac-dbt-propagSuppress( $Q$ )
1  while ( $Q \neq \emptyset$ ) do
2     $(i, a) \leftarrow \text{dequeue}(Q)$ ;
3    if  $\text{expl}(i \neq a) \subseteq \mathcal{C}$  then
4      foreach  $C_{ij} \in \mathcal{C}$  do
5         $E \leftarrow \text{mac-dbt-localArcConsExpl}(C_{ij}, i, a, Q)$ ;
6        if  $E$  is not a success then return  $E$ ;
7  return success;

function mac-dbt-localArcConsExpl( $C_{ij}, i, a, Q$ )
1  foreach  $b \in D_j$  s.t.  $b \in \text{supports}(C_{ij}, a)$  do
2     $\text{nbSupports}(j, b) \leftarrow -$ ;
3    if  $\text{nbSupports}(j, b) = 0$  then
4       $D_j \leftarrow D_j \setminus \{b\}$ ;  $\text{expl}(j \neq b) \leftarrow \bigcup_{a' \in \text{supports}(C_{ji}, b)} \text{expl}(i \neq a')$ ;
5       $Q \leftarrow Q \cup \{(j, b)\}$ ;
6      if  $D_j = \emptyset$  then return  $\bigcup_{a' \in \text{dom}_j} \text{expl}(j \neq a')$ ;
7  return success

function mac-dbt-giveValueAndCheck( $I, \mathcal{V}, i, a, Q$ )
1   $C_i = \text{constraint}(i = a)$ ;  $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_i\}$ ;
2  foreach  $b \in D_i$  s.t.  $b \neq a$  do
3     $\text{expl}(i \neq b) \leftarrow \{C_i\}$ ;
4     $Q \leftarrow Q \cup \{(i, b)\}$ ;
5   $D_i \leftarrow \{a\}$ ;
6   $E \leftarrow \text{mac-dbt-propagSuppress}(Q)$ ;
7  if  $E$  is a success then
8     $I \leftarrow I \cup \{(i, a)\}$ ;  $\mathcal{V} \leftarrow \mathcal{V} \setminus \{i\}$ ;
9  return  $E$ ;

function mac-dbt-updateDomains( $Back, Q$ )
1  foreach  $(i, a) \in \text{back}$  do
2     $D_i \leftarrow D_i \cup \{a\}$ ;
3    foreach  $C_{ij}$  do
4      foreach  $b \in D_j$  s.t.  $b \in \text{supports}(C_{ij}, a)$  do  $\text{nbSupports}(j, i, b)++$ ;
5  foreach  $(i, a) \in \text{back}$  do
6    if  $\exists C_{ij} \in \mathcal{C}$  s.t.  $(i, a)$  has no support on  $C_{ij}$  then  $Q \leftarrow Q \cup \{(i, a)\}$ ;
7  return mac-dbt-propagSuppress( $Q$ );

```

Fig. 2. mac-dbt

3.1 Implementing mac-dbt

First, we describe our proposal in the sight of the two previous problems. Then we address an implementation issue and finally discuss complexity results.

Computing nogoods during filtering The main loop of the algorithm remains unchanged (see function `mac-dbt` in fig. 2 compared to function `dbt` in fig. 1) except for the call to the `dbt-giveValueAndCheck` and `dbt-handleContradiction` functions which are replaced by a call to the `mac-dbt-giveValueAndCheck` and `mac-dbt-handleContradiction` functions. `mac-dbt-giveValueAndCheck(I, \mathcal{V}, i, a, Q)` merely assigns the value a to the variable i by removing all the other values from its domain and propagating those removals thanks to the `mac-dbt-propagSuppress` function.

`mac-dbt-propagSuppress` constantly takes a value removal from the propagation queue in order to propagate it on the related constraints. The key point is that the propagation scheme for any constraint needs to *explain* (to give an eliminating explanation for) each of its value removal. Function `mac-dbt-localArcConsExpl` shows how to do it for an `ac4`-like constraint propagation handling. When the number of supports for a given value of a variable reaches zero, that value needs to be removed. An explanation for that can be derived from the explanation for the removal of each of the supports (line 4 of the function).

Undoing past computations without real backtracking The `dbt-updateDomains` function needs to be modified to take into account the complete arc-consistent state restoration after undoing a variable assignment. This leads to the `mac-dbt-updateDomains` function.

The modifications (from `dbt-updateDomains`) starts at line 3 where counters are updated and value restorations are tested against each constraint of the system in order to get back to an arc-consistent state. If a value has been unduly reinserted, it is removed and all those removals are then propagated thanks to `mac-dbt-propagSuppress`.

Handling new contradiction cases First of all undoing a variable assignment as in lines 3–4 of `mac-dbt-handleContradiction` may not be as straightforward as in the original algorithm since that undoing may not be sufficient to come back to a consistent state due to constraint propagation. That is why it may be needed to handle a new contradiction: Henceforth the recursive call to `mac-dbt-handleContradiction` at line 5.

Moreover, that contradiction handling may lead to variable unassignment making irrelevant the would-be explanation for removing the original unassigned value to the first-place failing variable. Hence, the test in line 6 before actually removing the value which conversely may lead to a new contradiction that must be handled.

3.2 Complexity issues

As stated above, the total space complexity needed to manage nogoods is in the worst case $O(n^2d)$. Using local consistency algorithms provides very short explanations since the assignments appearing in the explanations are more relevant than in the explanations of **dbt**. In practice, the space required to store the explanations is less important than the space needed to represent the problem $O(ed^2)$. Furthermore, **mac-dbt** does not store information to backtrack to a previous state on a stack, which leads to lower space requirements compared to traditional approaches. Finally, space requirement had never been a limitation for our experiments, even on very large instances.

The time complexity involves a slight overhead comparing to the **mac** algorithm to compute explanations but obviously this does not change the worst case time complexity. This is a quite inexpensive additional task that allows avoiding some thrashing and provides explanations for failures.

4 Discussion

4.1 Dynamic arc-consistency

There are similarities between **mac-dbt** and the techniques used to maintain arc-consistency in dynamic CSPs [4, 7] since they both use a deduction maintenance system. However, there are real differences between the system of **justifications** used by DnAC-4 (and DnAC-6) and the system of explanations used by **mac-dbt**. Furthermore, the aim of the algorithms DnAC-* is to maintain arc consistency in dynamic CSPs and not to solve static CSPs. Even if we consider an assignment as a constraint restricting a domain, the ideas of DnAC-* cannot be used to solve a static CSP without an important work (explain how to build the succession of constraint additions/relaxations that have to be performed to solve the problem, specify how to deduce the set of constraints responsible of a dead-end, tell which of the assignments has to be reconsidered when a dead-end occurs, etc.).

4.2 Extensions

The key feature of **mac-dbt** relies on eliminating explanations which are associated to value removals. This could be extended to set of values: a first work on using explanations for intervals has been done for numerical CSP [14].

Any constraint solver able to provide explanations can be integrated in **dbt** in the same way we previously described it for arc-consistency. It would be the case for high level of *stronger* consistencies in binary CSP (see [9, 8] for high-level consistencies) or for non-binary constraints.

mac-dbt can also be used for different kinds of problems, for example to drastically improve the resolution of scheduling problems [13].

5 Experiments

Our experiments have been performed with an implementation of **mac-dbt** where constraints are handled *à la ac6*. **dbt** and **fc-dbt** were implemented using the eliminating explanations developed for **mac-dbt** leading to comparable versions of those algorithms.

5.1 dbt and fc-dbt versus mac-dbt

In our first set of experiments, we compare the three *dynamic backtracking* based algorithms: **dbt**, **fc-dbt** and **mac-dbt**.

Figure 3 shows results obtained on randomly generated⁴ problems of 15 variables whose domain size is 10 with 45 constraints (a 43% density). The varying parameter is the tightness of the considered constraints: from 10% to 90%.

The figure shows that the more constraint propagation is provided into the **dbt**, the less time is required to get an answer. On those problems, the advantage of **mac-dbt** is obvious.

5.2 mac-dbt versus mac

We compare now **mac-dbt** with the **mac7ps** [17] version of **mac**. Following [2] experiments, we generated problems which have an inherent structure⁵: the phase transition characteristics of regular problems is exploited to conveniently generate under-constrained instances containing a well-concealed small sub-problem chosen closed to the phase transition. That structure cannot be identified using a preprocessing phase and the problems need to be actually solved in order to discover that particularity.

Our first experiments were conducted on large instances: a series of problems consisting in 200 variables with domains of size 10 with 2500 constraints containing a small hard to solve instance of 15 variables and 43 constraints. Although **mac-dbt** solved each of that instances in a matter of seconds, it appeared that, for some of the instances, **mac** was unable to give any answer within several days of computation.

We therefore tried to find problems for which **mac** and **mac-dbt** performances were more similar. Figure 4 reports results obtained on a specific set of problems: the structured CSPs we have generated involve from 20 to 26 variables, each having 15 values in its domain. Each CSP contains two subproblems of 8 variables at .57 density and .76 tightness. Additional constraints of .05 tightness have been added in order to connect any couple of variables that are not in the same subproblem. Note that despite the artificial construction of those problems, they are more likely to be encountered in real life than pure random problems.

⁴ D. Frost, C. Bessiere, R. Dechter, and J.C. Regin. Random uniform CSP Generators, 1996, <http://www.ics.uci.edu/~dfrost/csp/generator.html>

⁵ As expected, average results on pure random CSP present a slight overhead for **mac-dbt** due to the explanations management that is of no benefit on that kind of problems.

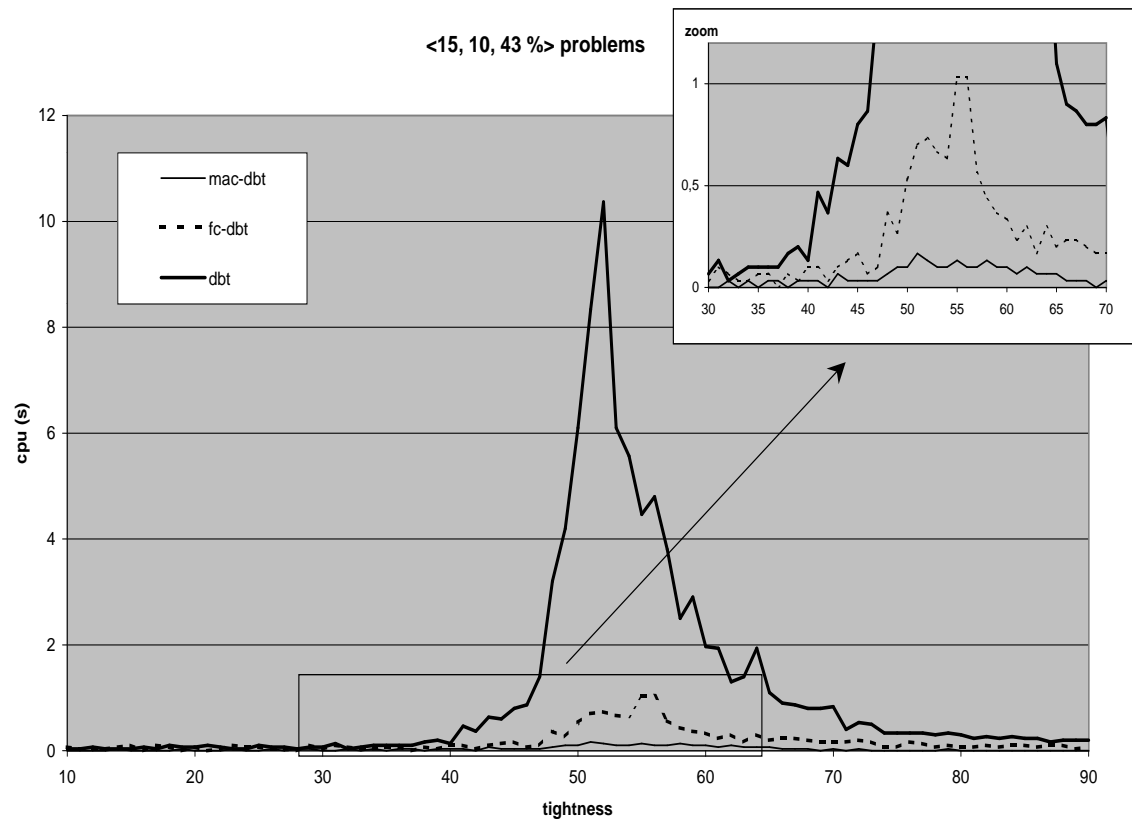


Fig. 3. Comparing dbt, fc-dbt and mac-dbt

mac-dbt vs mac

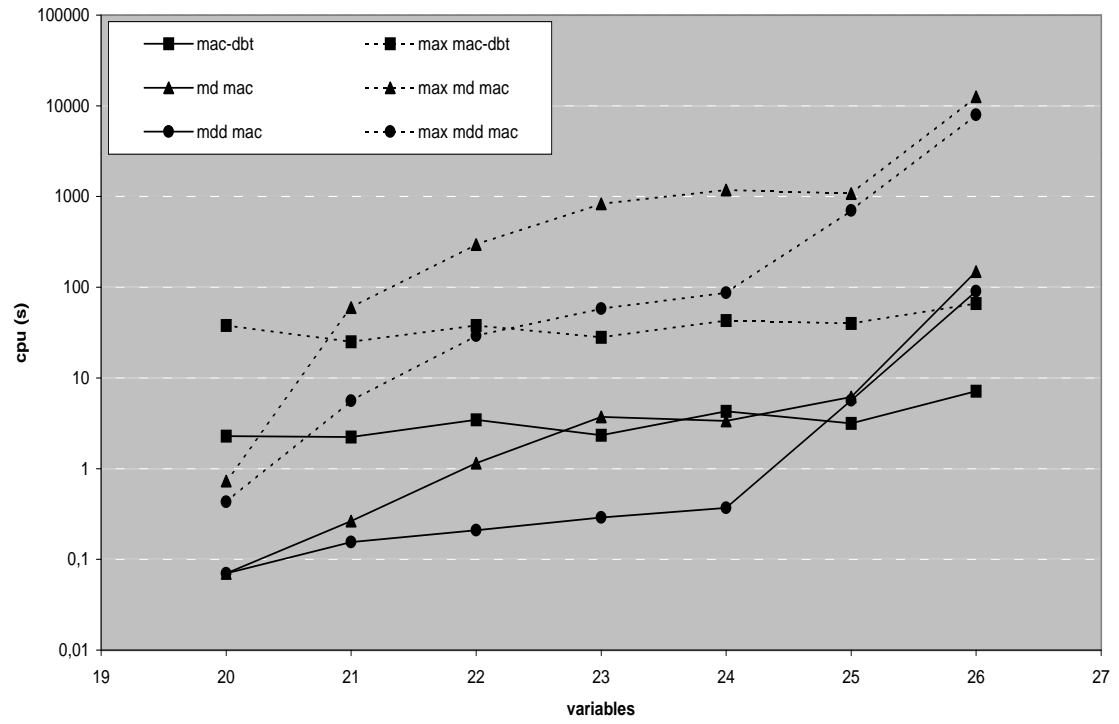


Fig. 4. Comparing mac-dbt and mac

We report in figure 4 the average and maximum cpu time in seconds to solve series of 500 problems in each point for both `mac-dbt` and `mac` using the `min dom/deg` and `mindom` variable dynamic ordering heuristic (resp. `mdd mac` and `md mac` in figure 4). We can see that `mac-dbt` remains stable as the number of variables arises meanwhile `mac` becomes prohibitive. Moreover, even if at the beginning the average results for `mac` with the `min dom/deg` heuristic are better than those of `mac-dbt`, note that for size 26, the average result of `mac` is even worse than the worst case of `mac-dbt` on the same instances. No results are given on larger problems because on some instances involving 27 variables `mac` required more than 4 hours to provide answers.

6 Conclusion and Further works

In this paper we have described the integration of arc-consistency in `dbt`, leading to `mac-dbt`. `mac-dbt` is to `dbt` what `mac` is to `sb`. This integration relies mainly on recording eliminating explanations.

`mac-dbt` shows very good results compared to `dbt` or `fc-dbt`: propagation improves efficiency. Experiments have shown that `mac-dbt` was able to solve very large problems and that it remains very stable as the size of the problems arises and even outperforms `mac` on structured problems even if for particular cases `mac-dbt` explores a greater part of the search tree than a standard backtracking based version [1].

Our current works include adapting our ideas to other consistency techniques such as `quick` [8] leading to `quick-dbt`. We deeply think that explanations, the key feature of `mac-dbt`, can be very useful in the CSP community. We have shown here their use for developing new search techniques. Another new search technique using explanations has been presented in [15] showing very good results on scheduling problems. Explanations could also be very useful in other fields such as explaining inconsistencies when debugging constraint programs or enabling constraint relaxation and so enhancing interactivity in solving over constrained problems.

References

1. Andrew B. Baker. The hazards of fancy backtracking. In *12th National Conf. on Artificial Intelligence, AAAI94*, pages 288–293, Seattle, WA, USA, 1994.
2. Roberto Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve exceptionnaly hard SAT instances. In *CP'96*, 1996.
3. Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.
4. Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.
5. Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problem. In *CP'96*, Cambridge, MA, 1996.

6. C. Bliet. Generalizing partial order and dynamic backtracking. In *Proceedings of AAAI*, 1998.
7. Romuald Debruyne. Arc-consistency in dynamic CSPs is no more prohibitive. In *8th Conference on Tools with Artificial Intelligence (TAI'96)*, pages 299–306, Toulouse, France, 1996.
8. Romuald Debruyne. *Local consistencies for large scale CSP*. PhD thesis, Université de Montpellier II, December 18 1998. In French.
9. Romuald Debruyne and Christian Bessière. From restricted path consistency to max-restricted path consistency. In *CP'97*, pages 312–326, Linz, Austria, October 1997.
10. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
11. Matthew L. Ginsberg and David A McAllester. Gsat and dynamic backtracking. In *International Conference on the Principles of Knowledge Representation (KR94)*, pages 226–237, 1994.
12. Narendra Jussien. *Relaxation de Contraintes pour les problèmes dynamiques*. 1. thèse, Université de Rennes I, 24 October 1997.
13. Narendra Jussien and Christelle Guret. Improving branch and bound algorithms for open shop problems. In *Conference of the International Federation of Operational Research Societies (IFORS'99)*, Beijing, China, August 1999.
14. Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csp. In *European Conference on Artificial Intelligence*, pages 224–228, Brighton, United Kingdom, August 1998.
15. Narendra Jussien and Olivier Lhomme. The path-repair algorithm. In *CP99 Post-conference workshop on Large scale combinatorial optimisation and constraints*, Alexandria, VA, USA, October 1999.
16. Patrick Prosser. MAC-CBJ: maintaining arc-consistency with conflict-directed backjumping. Research Report 95/177, Department of Computer Science – University of Strathclyde, 1995.
17. Jean-Charles Régim. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique*. Thèse de doctorat, Université de Montpellier II, 21 December 1995. In French.
18. Daniel Sabin and Eugene Freuder. Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, May 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).
19. Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
20. R. M. Stallman and G. J. Sussman. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.
21. Gérard Verfaillie and Thomas Schiex. Dynamic backtracking for dynamic cps. In Thomas Schiex and Christian Bessière, editors, *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, Amsterdam, August 1994.