RAIRO Operations Research
Will be set by the publisher

ABOUT THE CHOICE OF THE VARIABLE TO UNASSIGN IN A DECISION REPAIR ALGORITHM

Cédric Pralet[1] and Gérard Verfaillie[1]

Abstract. The *decision repair* algorithm [8], which has been designed to solve *constraint satisfaction problems* (CSP), can be seen, either (i) as an extension of the classical *depth first tree search* algorithm with the introduction of a free choice of the variable to which to backtrack in case of inconsistency, or (ii) as a *local search* algorithm in the space of the partial consistent variable assignments. or (iii) as a hybridisation between *local search* and *constraint propagation*. Experiments reported in [10] show that some heuristics for the choice of the variable to which to backtrack behave well on consistent instances and that other heuristics behave well on inconsistent ones. They show also that, despite its a priori incompleteness, *decision repair*, equipped with some specific heuristics, can solve within a limited time almost all the consistent and inconsistent randomly generated instances over the whole constrainedness spectrum. In this paper, we discuss the heuristics that could be used by *decision repair* to solve consistent instances, on the one hand, and inconsistent ones, on the other hand. Moreover, we establish that some specific heuristics make *decision repair* complete.

Keywords: Constraint Satisfaction Problem, Depth First Tree Search, Local Search, Constraint Propagation, Backtrack, Heuristics, Completeness.

Mathematics Subject Classification.

14th October 2004.

[1] LAAS-CNRS, Toulouse, France; e-mail: cpralet@laas.fr & gverfail@laas.fr

1. Introduction

In [8], N. Jussien and O. Lhomme proposed an algorithm called *decision repair* (or *path repair* in previous papers) dedicated to the solving of *constraint satisfaction problems* (CSP) [4,9]. This algorithm can be assessed along various points of view:

- it can be seen as an extension of the classical *depth first tree search* algorithm, allowing the variable to which to backtrack to be freely chosen among the currently assigned variables in case of inconsistency of the current partial assignment;
- it can also be seen as a *local search* algorithm in the space of the partial locally consistent variable assignments, with two kinds of move: variable assignment in case of local consistency and variable unassignment in case of inconsistency;
- it can finally be seen as a hybrid algorithm which closely combines *local search* and *constraint propagation*, the same way as usual constraint solving algorithms closely combine *tree search* and *constraint propagation*.

In this algorithm, *value removal explanations*, which are produced and recorded by constraint propagation, are sets of variables the assignment of which forbids a value[1]. They play three roles:

- they allow an *inconsistency explanation* to be built when the domain of a variable becomes empty; this explanation is the union of the removal explanations of all the values of the variable the domain of which is empty; it means that the current assignment of all the variables involved in the explanation is inconsistent with the whole problem, and it can be exploited by choosing among these involved variables the one to unassign;
- they may allow an *empty inconsistency explanation*, that is a proof of inconsistency, to be built in case of inconsistency of the whole problem;
- they allow *decremental constraint propagation* to be performed efficiently in case of variable unassignment, without computing everything again from scratch.

With regard to *termination*, *correctness*, and *completeness*, its properties are the following:

- *decision repair* may not terminate, unless an arbitrary stopping criterion is implemented;
- it is correct: when solving a problem P, if it returns *yes* and a complete assignment A, P is consistent and A is solution of P; if it returns *no*, P is inconsistent;
- it is incomplete: when solving a problem P, consistent or not, it may never terminate; if the stopping criterion is activated, it returns *?*, which means *"I don't know"*.

[1] In the *decision repair* algorithm, value removal explanations are more generally sets of decisions (domain restriction, constraint adding, etc) which forbid a value. In this paper, we restrict ourselves to decisions that are value assignments.

Experiments reported in [10] lead to the following observations:

- *decision repair* may be far more efficient than usual local search algorithms, which search for a solution in the space of the complete variable assignments, and use no constraint propagation, only constraint checking;
- it may be more efficient than complete comparable algorithms such as *chronological backtracking* (usual *depth first tree search*), *conflict directed backjumping* [12], or *dynamic backtracking* [5];
- it can solve many inconsistent problems at the frontier or beyond the frontier between consistency and inconsistency; within a limited time, it can even solve more inconsistent problems than complete comparable algorithms can;
- the *heuristics* used for the choice of the variable to unassign in case of inconsistency that are efficient to produce solutions and prove problem consistency are not the same as the ones that are efficient to prove problem inconsistency.

These observations led us to consider two questions:

(1) At least concerning the choice of the variable to unassign in case of inconsistency, are there heuristics dedicated to consistent problems and other ones dedicated to inconsistent ones, and which are they?
(2) Knowing that *dynamic backtracking* is an instance of *decision repair* where the variable to unassign is systematically the most recently assigned one in the current inconsistency explanation, but is complete, do there exist other or weaker conditions on the way of choosing the variable to unassign that guarantee completeness?

This paper presents preliminary answers to both questions. It is structured as follows: in Section 2, we present the *decision repair* pseudo code and, as examples, two traces of execution on a consistent problem and an inconsistent one; in Section 3, we show that there is no reason to use different heuristics for the choice of the variable to assign and the value to assign to it according to whether the problem is consistent or not; in Section 4, we show that the situation is different for the choice of the variable to unassign: various reasons lead to different heuristics according to the supposed nature of the problem; in Section 5, we present results of experiments carried out on randomly generated, but not completely homogeneous, consistent or inconsistent problems, with unassignment heuristics dedicated to consistent problems and dedicated to inconsistent ones; in Section 6, we present two sufficient conditions for the completeness of the algorithm that generalise the proof of completeness of *dynamic backtracking*; finally, in Section 7, we list unanswered questions that deserve further attention.

2. The decision repair algorithm

2.1. Algorithm pseudo code

The algorithm we consider is derived from the one presented in [8], but can be seen as a more generic version where more parameters remain to be set. Its pseudo code is shown in Figure 1.

```
Data   : a CSP P and a starting assignment A.
         n is the number of variables in P.
begin
    bool ← Initial_Filter(P, A)
    repeat
        if bool then
            v ← Extend_Assignment(P, A)
            if v = n + 1 then return yes
            else bool ← Incremental_Filter(P, A, v)
        else
            v ← Repair_Assignment(P, A)
            if v = 0 then return no
            else bool ← Decremental_Filter(P, A, v)
    until Stop()
    return ?
end
```

Figure 1: A generic decision repair algorithm

The starting assignment A may be empty, partial, or complete. Function *Initial_Filter* uses constraint propagation to enforce any local consistency property on the problem P restricted by A. It returns $True$ if P is locally consistent and $False$ if not. Function *Extend_Assignment* chooses a variable v which is not assigned in the current assignment A and extends A by choosing a value a for v. It returns $n + 1$ if there is no such variable. It is the case when A is complete. Conversely, function *Repair_Assignment* chooses a variable v which is assigned in the current assignment A and repairs A by unassigning v. It returns 0 if there is no such variable. It is the case when A is empty or when inconsistency explanations are produced and recorded and the current inconsistency explanation is empty. Functions *Incremental_Filter* and *Decremental_Filter* use respectively incremental and decremental algorithms to enforce local consistency, without computing everything again from scratch. As does *Initial_Filter*, they return $True$ if the current subproblem is locally consistent and $False$ if not. Function *Stop* implements any stopping criterion.

2.2. Algorithm outputs

The algorithm ends with a proof of consistency of P and an associated solution (answer *yes*), with a proof of inconsistency of P (answer *no*), or with nothing in case of activation of the stopping criterion (answer *?*). See Figure 2 for a comparison with the outputs of classical tree or local searches. Note the complete symmetry of the algorithm with regard to extension and repair and to consistency and inconsistency.

	Consistent instances	**Inconsistent instances**
Tree search	*Yes*	*No*
Local search	*Yes* or *?*	*?*
Decision repair	*Yes* or *?*	*No* or *?*

Figure 2: Possible outputs on consistent and inconsistent problem instances.

2.3. Algorithm main parameters

The main parameters that remain to be set are the following ones:

- the local consistency property which is enforced at each step of the algorithm and the way of enforcing it incrementally or decrementally: forward checking, arc consistency, etc.
- the way value removal explanations are handled i.e., produced, recorded, combined, removed, etc.
- the variables and values that are affected by local consistency enforcing: all the variables or only the not assigned ones, all the values or only the currently not removed ones;
- the heuristics that are used in case of local consistency of the current partial assignment to choose the variable to assign and the value to assign to it;
- the heuristics that are used in case of inconsistency of the current partial assignment to choose the variable to unassign;
- the presence or absence of priority for assignment (resp. unassignment) to the variable that has been unassigned (resp. assigned) just before, when this assignment (resp. unassignment) immediately follows an unassignment (resp. assignment);
- finally, the stopping criterion which is used when no result (*yes* or *no*) has been produced.

2.4. Two trace examples

To put things in a more concrete form, we show in Figures 4 and 5 possible traces of such an algorithm on a consistent graph colouring problem P_c and on an inconsistent one P_i (see Figure 3).

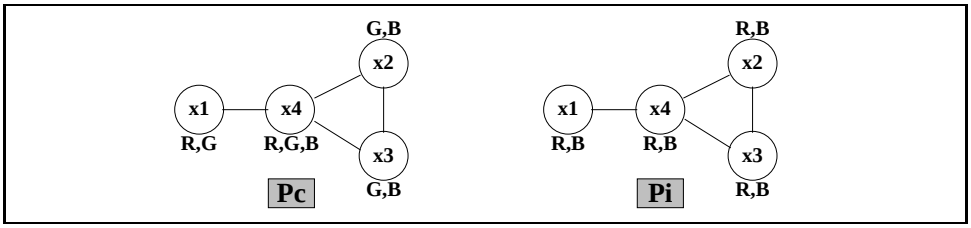

Figure 3: A consistent graph colouring problem P_c and an inconsistent one P_i.

For these examples, we consider a specific *decision repair* algorithm which behaves as follows. After each assignment of a variable v, *forward checking* is performed from v to the current domains of the not assigned variables. For each removed value, the singleton $\{v\}$ is recorded as a removal explanation. When the domain of a variable v' is wiped out, a variable v'' is chosen to be unassigned in the current inconsistency explanation, that is in the union of the value removal explanations in the domain of v'. After unassignment of v'', a value removal explanation is created for its previous value (the previous inconsistency explanation minus v''). Irrelevant value removal explanations, those that involve v'', are then removed. The associated values are restored. But forward checking must be performed again from the assigned variables to the current domains of the not assigned ones. For assignment, the variable of lowest index among the variables of smallest current domain is chosen. Values are tried in the order $\{R, G, B\}$ for P_c and $\{R, B\}$ for P_i. For unassignment, a variable is randomly chosen in the current inconsistency explanation.

Because *decision repair* in not a tree search but a local search in the space of partial assignments, its trace is only a sequence of states, each state being composed of a partial assignment and of a set of value removal explanations. In each state, initially forbidden values are pointed out in dark grey, current assignments by a small black square, and currently removed values by the indices of the variables that are involved in their removal explanation. Values the removal explanation of which is empty, those that are inconsistent whatever the assignment of the other variables is, are pointed out in light grey.

For example, on P_c (see Figure 4), in state s_4, the domain of variable x_4 is wiped out and all the other variables are involved in the inconsistency explanation. We assume that variable x_2 is chosen to be unassigned. This is what is done in state s_5. Value G is removed from the domain of x_2 with $\{x_1, x_3\}$ as an explanation. The value removal explanations in which x_2 was involved are forgotten and associated values restored: value G for x_3 and x_4. But forward checking the current domain of x_2 removes value B with $\{x_3\}$ as an explanation.

On P_i, in state s_6 (see Figure 5), the domain of variable x_4 is wiped out with $\{x_2\}$ as an explanation. Variable x_2 is unassigned and its previous value R is removed with an empty explanation. It is thus sure that value R for x_2 does not take part to any solution and can be removed from its domain. Similarly, when, in state s_{10}, the domain of variable x_3 is also wiped out with $\{x_2\}$ as an explanation, variable x_2 is unassigned and its previous value B is removed with

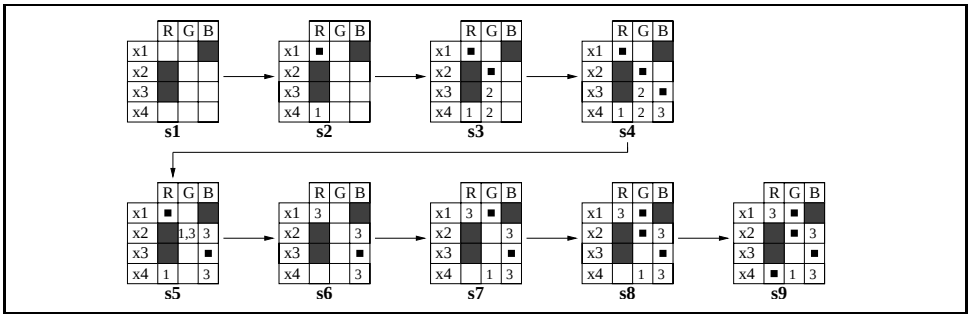

Figure 4: A possible execution of *decision repair* on P_c.

an empty explanation. It is thus sure that value B for x_2 does not take part to any solution and can be removed from its domain. Because the domain of x_2 is now empty, inconsistency of P_i is proven.

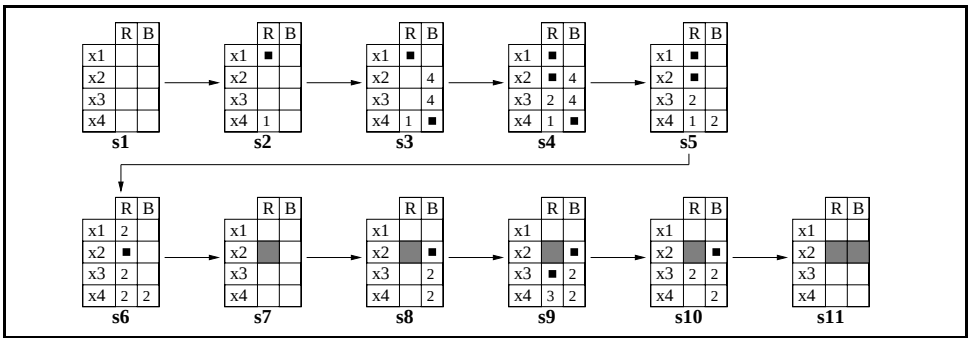

Figure 5: A possible execution of *decision repair* on P_i.

3. About the heuristics for the choice of the variable to assign and the value to assign to it

3.1. Choice of the variable to assign

The heuristics that are used to choose the next variable to assign in case of local consistency of the current partial assignment have two objectives:

- to reduce the *width* of the tree to explore[2]; this result is achieved by considering first the variables the current domain of which is the smallest;

[2]Speaking of a tree is abusive here, because *decision repair* does not explore a tree, but a graph nodes of which are partial assignments and edges of which connect two partial assignments when it is possible to go from one to the other via either an assignment, or an unassignment. But the graph of *decision repair* and the tree of a classical *backtrack* search have the same set of nodes: the set of possible partial assignments. The only difference is in the set of edges which is much larger with *decision repair* because of the larger freedom of unassignment: with *backtrack* search, unassignment must be performed in the inverse order of the asignment order.

indeed, if we have a set of variables with various domain sizes, the minimal enumeration tree in terms of number of internal nodes is obtained by ordering variables according to an increasing domain size;

- to reduce the *depth* of the tree to explore; this result is achieved by considering first the variables that are the most constrained, for example that are involved in the greatest number of constraints or in the tightest constraints, in order to prove subproblem inconsistency as soon as possible and thus to cut the enumeration tree as high as possible.

Both objectives can be combined in a heuristic such as *dom/deg*: to choose a variable the ratio of which between the size of the current domain and the degree in the constraint graph is the smallest [2].

This kind of heuristic is a sensible choice whatever the nature of the problem to solve is: consistent or not. In case of inconsistency, it reduces the size of the tree to explore (see Figure 6). In case of consistency, it reduces the size of the part of the tree to explore before finding a solution (see Figure 7).

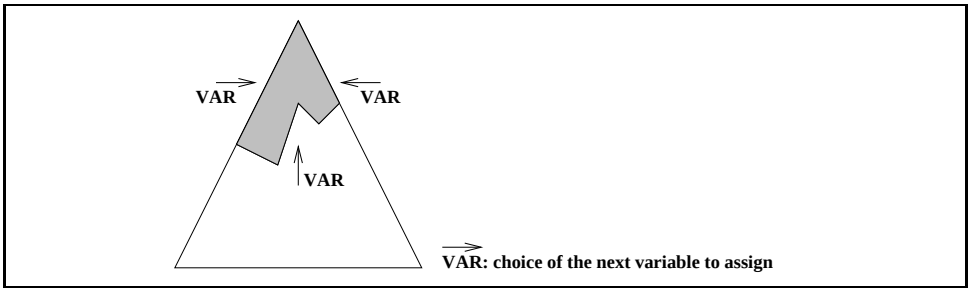

Figure 6: Objectives of the variable heuristics on inconsistent problems.

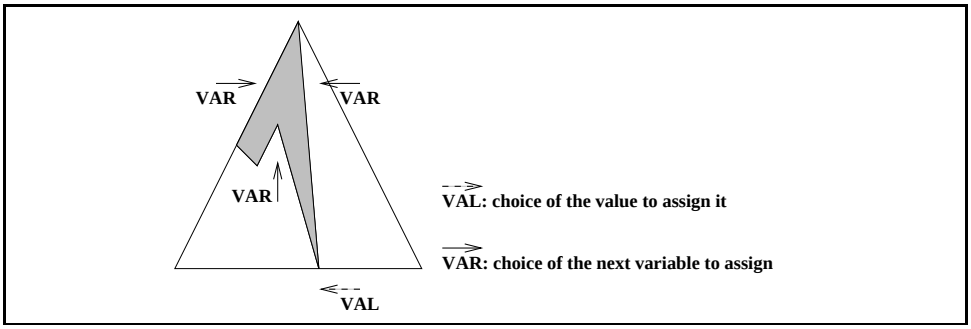

Figure 7: Objectives of the variable and value heuristics on consistent problems.

3.2. Choice of the value

If we consider now the heuristics that are used to choose the value to assign to the variable that has been chosen for assignment, they have only one objective:

- to reduce the *part* of the tree to explore before finding a solution (see Figure 7); this result is achieved by considering first the values that can the most likely extend the current partial assignment into a solution, in order to prove problem consistency as soon as possible.

In case of inconsistency, such heuristics have no utility, because no solution exists. It results in no gain, but no cost, unless it needs too much computing. In case of consistency, it may result in some gain. Thus, it is a sensible choice whatever the nature of the problem to solve is: consistent or not.

4. About the heuristics for the choice of the variable to unassign

The landscape is different if we consider the heuristics to use to choose the variable to unassign in the current inconsistency explanation: a topic which has not been really explored until now because usual tree search algorithms do not offer such a freedom.

If the problem is consistent, the objective is to build a solution as quickly as possible. In case of inconsistency of the current partial assignment, the objective is thus to identify and to undo bad choices in order to allow the algorithm to make better ones.

Conversely, if the problem is inconsistent, the objective is to build a proof of inconsistency, that is an empty inconsistency explanation, which contains no variable assignment. This result can be achieved if and only if a variable has an empty domain and the removal explanations of all its values are themselves empty (see for an example Figure 5 in Section 2.4). The objective is thus to empty domains with an as small as possible union of the value removal explanations. Note that this result is implicitly achieved in usual *depth first tree search* when all the values of the first assigned variable have been tried without any success. With *decision repair*, we want to achieve the same objective using a less ordered search and explicit value removal explanations.

If we try to list the heuristics that could be used to build a solution, we get the following ones:

(1) to choose *randomly* among the assigned variables, in order to diversify the search, as usually done in local searches [1];
(2) to choose a variable of *maximum current domain size*, because it offers more space for value choice;
(3) to choose a variable of *minimum degree* in the constraint graph, for similar reasons;
(4) to choose a variable that is involved in the *greatest number of value removal explanations*, because its unassignment allows many values to be restored in the domains of the other variables, unless these values must be maintained removed because of other assignments;

(5) more generally, to choose a variable such that the subproblem resulting from its unassignment is the *least constrained*, using any measure of constrainedness;
(6) to choose the *least recently assigned* one, because first choices are generally the least informed, and thus the most questionable;
(7) if a note can be associated with any value of any variable, measuring for example its likelihood of taking part in a solution, to choose a variable the assignment of which has the *worst note*;
(8) in the same conditions, to choose a variable the assignment of which is the *most doubtful*, for example such that the difference between the note of the current assignment and the one of the best assignments in the current domain (alternative choices) is the smallest.

Note that heuristics 2, 3, 4, and 5 aim at recovering space for future assignments and that heuristics 6, 7, and 8 aim at undoing bad or doubtful choices.

If we try now to list the heuristics that could be used to build a proof of inconsistency, we get the following ones:

(1) to choose a variable of *minimum current domain size*, because we are with such a variable closer to an inconsistency, the explanation of which is empty or not;
(2) to choose a variable of *maximum degree* in the constraint graph, for similar reasons;
(3) to choose a variable such that the subproblem resulting from its unassignment is the *most constrained*, using any measure of constrainedness;
(4) to choose the *most recently assigned* one, because last choices remove generally fewer values than first ones do and thus undoing them compels the algorithm to destroy fewer value removal explanations, if we make the assumption of an algorithm which removes irrelevant value removal explanations, that are inconsistent with the current partial assignment, in order to save memory space;
(5) to choose a variable that is involved in the *smallest number of value removal explanations*, in order to destroy the fewest of them (with the same assumption about the algorithm behaviour), each explanation being eventually weighted by the cost of its production (a production via backtrack is a priori more costly than a direct production via local consistency enforcing) divided by the number of involved variables (a small explanation is more valuable than a large one);
(6) to choose a variable such that, after unassignment, the number of variables involved in the union of its value removal explanations, which can be seen as an inconsistency explanation in progress, eventually divided by the number of removed values, is the smallest, in order to get closer to an empty inconsistency explanation.

Note that heuristics 1, 2, and 3 aim at producing inconsistency as soon as possible, whereas heuristics 4 and 5 aim at maintaining as much as possible existing

value removal explanations, and heuristic 6 aims at producing small inconsistency explanations.

Note also that heuristics 1, 2, 3, 4, and 5 for inconsistent problems are the exact opposites of heuristics 2, 3, 5, 6, and 4 for consistent problems. If we know nothing about the nature of the problem to solve (consistent or not), this suggests a strategy which would consist to run in parallel two searches with two different goals (to build a solution and to build an inconsistency proof) and thus two different heuristics for the choice of the variable to unassign. But, if we conjecture that the problem is consistent (resp. inconsistent), this suggests another strategy which would consist first to try to build a solution (resp. an inconsistency proof) and then to switch the search goal, or to work towards both in parallel, if the previous one has not been achieved by a given deadline.

5. Experiments

Although we carried out experiments on many problems and instances, with many algorithmic variants, we report in this paper only the ones that have been carried out on randomly generated, but not completely homogeneous, binary CSPs[3], with a limited number of algorithmic variants, some of them dedicated to consistent instances and others dedicated to inconsistent ones.

5.1. Problem instances

We considered binary CSPs, randomly generated with the usual four parameters: number of variables n, domain size d (the same for all the variables), graph connectivity p_1, and constraint tightness p_2 (the same for all the constraints), but we broke their homogeneity by partitioning the set of variables into nc clusters of the same size and by introducing a graph connectivity p_1 inside each cluster (the same for all the clusters) and a lower one p_{1c} between clusters.

The experimental results that are shown in Figures 8 and 9 have been obtained with $n = 50$, $d = 15$, $nc = 5$, $p_1 = 30$, $p_{1c} = 20$, $p_2 = \frac{100 \cdot np}{d^2} = \frac{100 \cdot np}{225}$, and np varying between 81 and 93 by step of 1 around the complexity peak, that is p_2 varying between 36 and 41.3. 10 instances have been generated for each value of np. For $np \leq 87$ ($p_2 \leq 38.7$), nearly all the generated instances were consistent and we only considered them (no inconsistent instances). Inversely, for $np \geq 88$ ($p_2 \geq 39.1$), nearly all the generated instances were inconsistent and we only considered them (no inconsistent instances).

[3]The choice of focusing experiments on this kind of problems is that classical random CSPs are too homegeneous to exhibit the benefits of methods that turn to advantage the explicit or implicit structure of the instance to solve. For example, all the variables in a classical randomly generated instance have more or less the same degree. Although less systematic, the experiments we carried out on Latin square completion problems, another kind of structured problems [7], produced results that are similar to the ones we show in this section.

5.2. Algorithms

The algorithms we compared are *backtrack* (BT), *conflict directed backjumping* (CBJ), *dynamic backtracking* (DBT), *min conflicts* (MC), and three variants of *decision repair*: DR(rand), DR(mostdoubt), and DR(mindestroy).

Except MC, all these algorithms perform forward checking. Except MC and BT, all of them compute and record value removal explanations, and maintain only those that remain relevant. Forward checking is only performed on the unassigned variables and on their current domains. Except MC, all of these algorithms give priority for assignment (resp. unassignment) to the variable that has been unassigned (resp. assigned) just before (see Section 2.3). Except for MC, the assignment heuristic consists in choosing an unassigned variable of smallest ratio between its current domain size and its degree in the constraint graph. Except for MC and the second variant of DR (DR(mostdoubt)), the value heuristic is random. The three variants of DR have in common the choice of the variable to unassign inside the current inconsistency explanation, but differ in the way of making this choice[4]:

- DR(rand) makes it randomly with a uniform probability distribution over the current inconsistency explanation;
- DR(mostdoubt) uses the results of the pre-computing, for each value of each variable, of the number of values it removes in the domains of the other variables; values are ordered in each domain according to an increasing value of this number; the first value in each current domain is chosen for assignment; the doubtful nature of an assignment is measured by the difference between the number associated with the first value and the one associated with the second one; a variable the assignment of which is the most doubtful (such that the difference is the smallest) in the current inconsistency explanation is chosen for unassignment, with random tie-breaking; such a heuristic clearly aims at producing solutions by removing doubtful assignment choices;
- DR(mindestroy) uses weights that are associated with each assigned variable; when assigned, the weight of a variable v is initialised with the number of values its assignment removes in the domains of the not assigned variables; when unassigned, its weight is reset to 0 and its previous weight is equally distributed among the variables in the current inconsistency explanation that are different from v; a variable the weight of which is the smallest in the current inconsistency explanation is selected, with random tie-breaking; this heuristic clearly aims at producing inconsistency proofs by keeping recorded as long as possible value removal explanations which result either from constraint propagation or from backtrack and may have been generated thanks to a lot of computation.

[4]The ways of choosing the variable to unassign inside the current inconsistency explanation used in these three variants are, among all we have experimented, the ones that exhibit a behaviour that is significantly better than the one of classical algorithms at least on a significant constraint thightness interval: before, at, or after the consistency/inconsistency frontier.

5.3. Experimental results

Because all the considered algorithms involve random choices, each of them has been run 20 times on each instance. Each run has been given a maximum CPU-time of 1200 seconds. A CPU-time of 1200 seconds is associated with any run which did not finish by the deadline. Figure 8 reports the median CPU-time as a function of p_2, on consistent and then on inconsistent instances. Figure 9 reports the percentage of runs which did not finish by the deadline, among all the runs on all the instances associated with each value of p_2, on consistent and then on inconsistent instances. These results allow us to make the following observations.

- if MC may be efficient on consistent instances, it becomes quickly inefficient when approaching the consistency/inconsistency frontier; it is, as other classical local search algorithms, unable to solve inconsistent instances;
- BT, CBJ, and DBT present the same usual behaviour with a peak of complexity at the consistency/inconsistency frontier; results of CBJ and DBT are clearly better than those of BT on consistent and inconsistent instances; on consistent instances, CBJ performs better than DBT, but situation is opposite on inconsistent instances;
- DR(rand) and DR(mostdoubt) produce practically identical results, with only a small advantage to the second one; their global behaviour is similar to that of MC; although they are the most efficient on consistent instances until the consistency/inconsistency frontier, they are, as MC and other classical local search algorithms, unable to solve inconsistent instances;
- although it is basically a local search algorithm in the space of partial assignments, and thus a priori incomplete, DR(mindestroy) presents the same behaviour as do complete algorithms such as BT, CBJ, and DBT; moreover, it is the most efficient on inconsistent instances until the consistency/inconsistency frontier.

6. Sufficient conditions of completeness

These experimental results show that the behaviour of some instances of *decision repair*, such as DR(mindestroy), is similar to the one of complete algorithms such as *backtrack*, *conflict directed backjumping*, and *dynamic backtracking*. It exhibits the same *easy-hard-easy* pattern and is clearly different from the one of incomplete algorithms such as *min conflicts*. Hence, a question arises: are these instances in fact complete ?

One can first note that *dynamic backtracking*, which is an instance of *decision repair* where the variable to unassign is systematically the most recently assigned one in the current inconsistency explanation, has been proven to be complete [5]. Moreover, other instances of *decision repair* where only partial order conditions

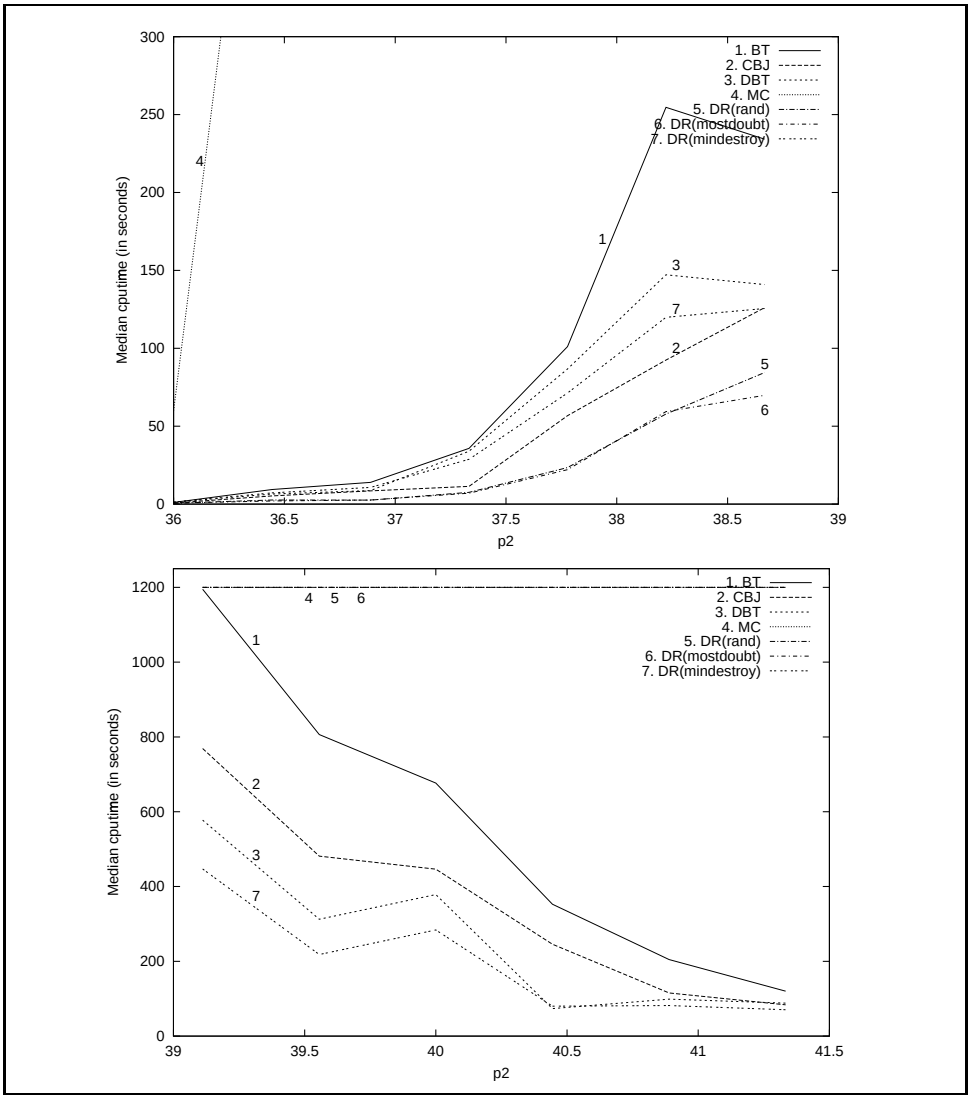

Figure 8: Median CPU-time on randomly generated consistent (top) and inconsistent (bottom) problem instances. Note that scales are different for sake of readability.

must be met between the variable to unassign and the other variables in the current inconsistency explanation (*partial order* and *general partial order backtracking*) have been proven to be complete too [3, 6]. One the other hand, *incomplete dynamic backtracking*, which can be seen as a variant of *decision repair* where the variable to unassign is chosen either randomly or using any heuristic is clearly incomplete [11].

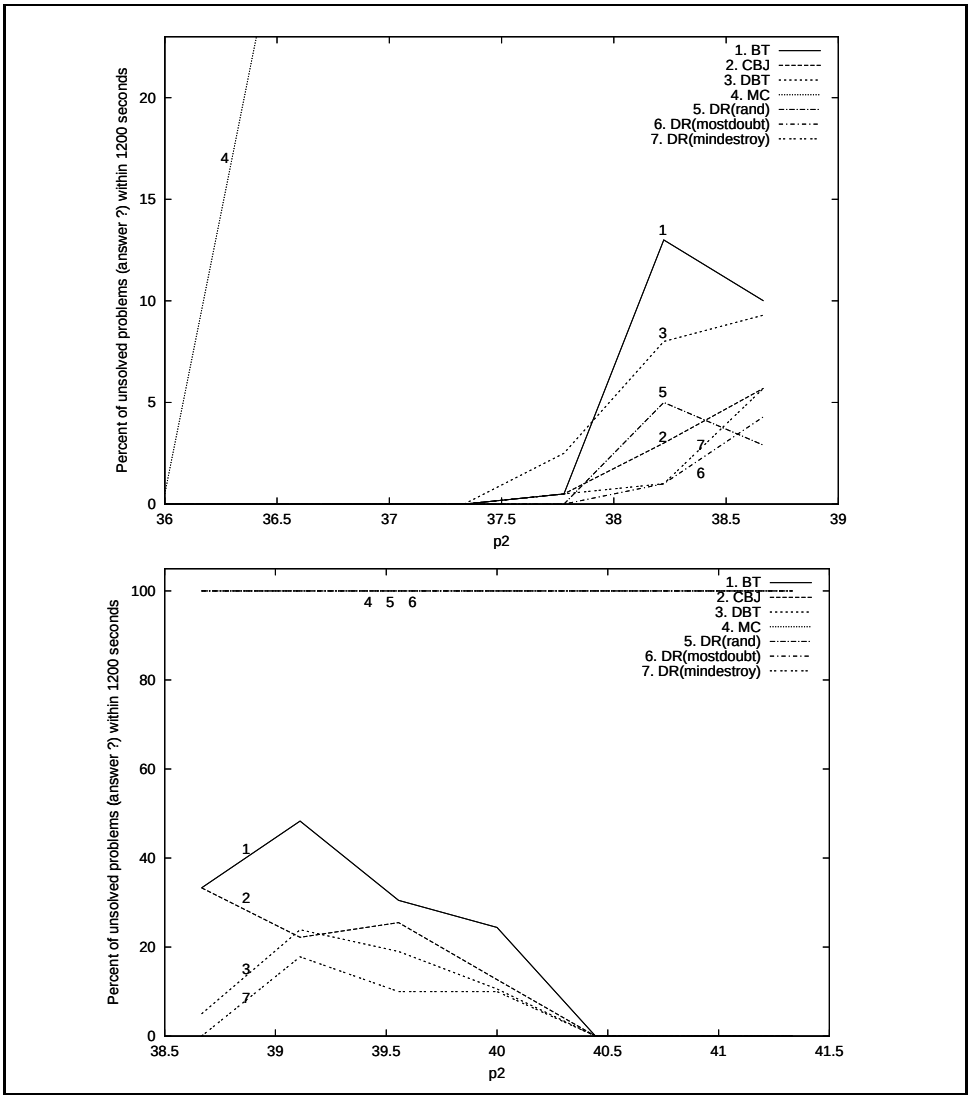

Figure 9: Number of unsolved consistent (top) and inconsistent (bottom) problem instances within 1200 seconds. Note that scales are different for sake of readability.

This suggests that some freedom in the choice of the variable to unassign is not incompatible with completeness, but that some restrictions of this freedom are necessary to guarantee it. In this section, we consider two kinds of restrictions that are sufficient to guarantee completeness and generalise the result of completeness of *dynamic backtracking*. Note that these restrictions are sufficient but not necessary to guarantee completeness, and that other restrictions could offer the same guarantee.

First, we assume that a weight $w(v,k)$ can be associated with each variable v at each step k of the algorithm. A step is associated with each passing through the main *repeat* loop of the algorithm with a locally consistent or inconsistent partial assignment (variable *bool* set to *true* or *false*; see Figure 1). Then, we assume that these weights meet the following initialisation and updating rules:

(1) at $k = 0$, all the weights are initialised to 0;
(2) if a variable v is assigned at step k, its weight at step $k+1$ is set to $\alpha(v,k)$ and the weights of all the other variables are not modified;
(3) if a variable v is unassigned at step k, its weight at step $k+1$ is set to 0, the weight of each assigned variable v', involved or not in the current inconsistency explanation, is updated by adding $\beta(v',k)$ to it, but the weights of the not assigned variables are not modified.

Note that, at each step k, the weights of the not assigned variables are null. The quantity $\alpha(v,k)$ represents the importance or the confidence that can be associated with the assignment of a variable v at step k. The quantity $\beta(v',k)$ represents the change in the importance or the confidence associated with the assignment of a variable v', resulting from the unassignment of another variable v at step k, in other words the influence of v's unassignment on v''s weight.

We assume that the *decision repair* algorithm chooses systematically the variable to unassign among the ones of smallest weight in the current inconsistency explanation, that is among the ones the assignment of which is the least important or reliable. We assume also that, if there are more than one variable of smallest weight in the current inconsistency explanation, the one to unassign is freely chosen, for example randomly with a uniform distribution. In such conditions, Theorems 6.1 and 6.2 can be proven.

Theorem 6.1. *If (1)* $\forall k, \forall v, [\alpha(v,k) \geq 0] \wedge [\beta(v,k) \geq 0] \wedge [w(v,k) \leq wMax]$ *and (2)* $\forall k, \forall v, \forall v', [v \neq v'] \wedge [v, v' assigned\ at\ step\ k] \Rightarrow [|w(v,k) - w(v',k)| \geq dMin > 0]$*, then decision repair is complete.*

In other words, if α and β are positive or null, if weights have an upper bound $wMax$, and if the distance between the weights of two assigned variables has a lower bound $dMin > 0$, then *decision repair* is complete.

Using a more symbolic approach, Theorem 6.1 could be reformulated as follows: if each variable is valued using a finite and totally ordered valuation set E, if the valuation of each not assigned variable is the minimum element of E, if the valuation of a variable cannot decrease unless it is unassigned, and if two assigned variables cannot be equally valued, then *decision repair* is complete.

One can observe that *dynamic backtracking* meets these conditions by simulating it using the following values for α and β: if v is assigned at step k, $\alpha(v,k)$ is equal to the number of not assigned variables at step k; if v is unassigned at step k, $\beta(v,k)$ is equal to 1 for all the assigned variables the weight of which is smaller than v's weight and equal to 0 for all the other assigned variables. It is easy to show that the weight of each assigned variable is an integer between 1 and the number n of variables, that two assigned variables cannot have the same weight, and that the variable to unassign is systematically the most recently assigned in

the current inconsistency explanation. But *dynamic backtracking* can be generalised without forsaking completeness by using any other criterion (different from the number of not assigned variables) to set $\alpha(v,k)$ when assigning v at step k.

It also possible to use any criterion as a first criterion and the assignment order as a second criterion by using the following values for α and β: if v is assigned at step k, if the valuation of v (using the first criterion) is equal to any number e between 1 and $eMax$, and if m is the number of already assigned variables the valuation of which (using the same first criterion) was equal to e, then $\alpha(v,k)$ is equal to $e - \frac{m}{n}$; if v is unassigned at step k, $\beta(v',k)$ is equal to $\frac{1}{n}$ for all the assigned variables v' the valuation of which is equal to v's valuation and the weight of which is smaller than v's weight, and equal to 0 for all the other assigned variables. Conditions of Theorem 6.1 are still met, with $wMax = eMax$ and $dMin = \frac{1}{n}$. Completeness is thus guaranteed.

Theorem 6.2. *If (1)* $\forall k, \forall v, [0 \leq \alpha(v,k) \leq \alpha Max] \wedge [0 \leq \beta(v,k) \leq f(w(v,k))]$, *$f$ being a continuous, positive, and strictly increasing function, and*
(2) $\forall k, \forall v, [[v$ *unassigned at step* $k] \wedge [|IE(k)| > 1]] \Rightarrow [\beta(v',k) \geq \beta Min > 0]$ *for at least one variable v' of minimum weight in $IE(k) - \{v\}$, then decision repair is complete.*

In other words, if α and β are positive or null, if α has an upper bound αMax, if $\beta(v,k)$ has an upper bound of the form $f(w(v,k))$, f being a continuous, positive, and strictly increasing function, and if, when a variable v is unassigned at step k and is not alone in the current inconsistency explanation $IE(k)$, the weight of one of the variables in $IE(k) - \{v\}$ of minimum weight, is increased by at least $\beta Min > 0$, then *decision repair* is complete.

In Theorem 6.2, the condition of Theorem 6.1 of bounded weights is replaced by conditions of bounded α and β: a unique bound for α and a bound which depends on the current weight of the concerned variable for β. Moreover, the condition of Theorem 6.1 of a minimum distance between the weights of two assigned variables is replaced by the condition of a minimum increase in the weight of at least one variable of minimum weight in $IE(k) - \{v\}$, when v is unassigned at step k and is not alone in $IE(k)$. Note that v has been already chosen to be unassigned among the variables of minimum weight in $IE(k)$. Its weight at step $k+1$ is then set to 0. Moreover, if $|IE(k)| > 1]$, the weight of at least one variable of minimum weight in $IE(k) - \{v\}$ is increased by at least $\beta Min > 0$. Note also that two assigned variables may have now the same weight.

Let us consider for example the following way of setting α and β: if v is assigned at step k, $\alpha(v,k)$ is equal to the number of values the removal of which results from v's assignment; if v is unassigned at step k, if $IE(k)$ is the current inconsistency explanation, then $\beta(v',k)$ is equal to $\frac{w(v,k)}{|IE(k)-1|}$ for all the variables v' in $IE(k)$ that are different from v, and equal to 0 for all the other assigned variables. In case of assignment, the more removed values, the higher weight. In case of unassignment, the weight of the unassigned variable is equally distributed among the other variables involved in the current inconsistency explanation. It is lost if the unassigned variable is alone in the current inconsistency explanation. Finally, the

variable to unassign is roughly speaking chosen among the variables in the current inconsistency explanation that remove either directly (via constraint propagation) or indirectly (via backtrack) the smallest number of values in the domains of other variables. This is the heuristic we use in the algorithm DR(mindestroy) the experimental results of which are shown in Section 5. This name is justified by the fact that this heuristic tries to maintain as much as possible the removals and removal explanations that have been built so far, in order to build a problem inconsistency proof. It is easy to show that conditions of Theorem 6.2 are met by this heuristic, with $\alpha Max = (n-1) \cdot d$ (maximum number of values removals resulting from one assignment, if n is the number of variables and d the maximum domain size), f the identity function (because, in the best case, there is only one variable in the current inconsistency explanation other than the unassigned variable v and because v's weight is the smallest one in the current inconsistency explanation), and $\beta Min = \frac{1}{n-1}$ (because, in the worst case, a weight of 1 is equally distributed among all the other variables). Completeness is thus guaranteed.

To prove both theorems, it must be first noted that there are only two termination cases for the *decision repair* algorithm: either a complete consistent assignment which is a proof of consistency, or an empty inconsistency explanation which is a proof of inconsistency. So, to prove algorithm completeness, it suffices to prove that it terminates. Termination proofs can be found in appendices A and B.

It must be however stressed that, although completeness is guaranteed, the worst-case solving time of *decision repair*, equipped with such heuristics, is not the same as the one of *dynamic backtracking*. It is in fact higher. If a search state is characterised by a partial assignment and a set of value removals from the domains of the not assigned variables, because of its termination property, no search state is visited twice by *dynamic backtracking*. The same property holds with *decision repair*, when heuristics satisfy conditions of Theorems 6.1 or 6.2, but with a search state now characterised by a partial assignment, a set of weights associated with the assigned variables, and a set of value removals from the domains of the not assigned variables. This implies that the worst-case solving time is roughly speaking multiplied by the maximum number of weight combinations. This number may be huge and depends on the values of $wMax$ and $dMin$ in Theorem 6.1, and on the value of αMax and on the function f in Theorem 6.2.

In fact, we hope that in spite of its higher worst-case solving time, *decision repair* will be able to exploit its larger freedom in the choice of the variable to unassign and to exhibit globally lower mean value and variance of the solving time. Experimental results of Section 5 show that this may be the case.

7. Looking further

Beyond these first results, many questions remain unanswered and need further theoretical and experimental studies. Among them:

- Can we define other or weaker sufficient conditions of completeness for *decision repair* algorithms? Can we better define necessary conditions of completeness?
- What can be the actual benefit of a strategy which would consist to run in parallel two searches: one aiming at building a solution, another one aiming at building an inconsistency proof? Could both searches benefit to each other, for example by exchanging locally consistent partial assignments or value removal explanations?
- What is the influence of the level of local consistency, checked on each partial assignment, on the efficiency of this kind of local search (forward checking, arc consistency ...), and, beyond that, the precise influence of all the parameters listed in section 2.3?

This study has been initiated when both authors were working at ONERA, Toulouse, France. Many thanks to Narendra Jussien, from École des Mines de Nantes, France, and Olivier Lhomme, from ILOG, Sophia-Antipolis, France, for the *decision repair* algorithm, and particularly to Narendra Jussien and Hadrien Cambazard, from École des Mines de Nantes too, for fruitful discussions about this algorithm.

References

[1] E. Aarts and J. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[2] C. Bessière and J.C. Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?). In *Proc. of the 2nd International Conference on Principles and Practice of Constraint Programming (CP-96, LNCS 1118)*, pages 61–75, Cambridge, MA, USA, 1996.

[3] C. Bliek. Generalizing Partial Order and Dynamic Backtracking. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 319–325, Madison, WI, USA, 1998.

[4] R. Dechter and R. Mateescu. Mixtures of Deterministic-Probabilistic Networks and their AND/OR Search Space. In *Proc. of the 20th International Conference on Uncertainty in Artificial Intelligence (UAI-04)*, Banff, Canada, 2004.

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

[6] M. Ginsberg and D. McAllester. GSAT and Dynamic Backtracking. In *Proc. of the 4th International Conference on the Principles of Knowledge Representation and Reasoning (KR-94)*, pages 226–237, Bonn, Germany, 1994.

[7] C. Gomes and B. Selman. Problem Structure in the Presence of Perturbations. In *Proc. of the 14th National Conference on Artificial Intelligence (AAAI-97)*, Providence, RI, USA, 1997.

[8] N. Jussien and O. Lhomme. Local Search with Constraint Propagation and Conflict-based Heuristics. *Artificial Intelligence*, 139:21–45, 2002.

[9] A. Mackworth. Constraint Satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. John Wiley & Sons, 1992.

[10] C. Pralet and G. Verfaillie. Travelling in the World of Local Searches in the Space of Partial Assignments. In *Proc. of the International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR-04)*, pages 240–255, Nice, France, 2004.

[11] S. Prestwich. Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search. *Annals of Operations Research*, 115:51–72, 2002.
[12] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problems. *Computational Intelligence*, 9(3):268–299, 1993.

Appendix A. Proof of Theorem 6.1

Proof. Theorem 6.1 can be proven as follows. Let us consider the solving of a problem pb at a step k, with n variables, sd values (sum of the domain sizes), a maximum weight w, and the unique assigned variable v of maximum weight (if there is no assigned variable, either one variable will be assigned the next step, or problem inconsistency has been proven). Two cases may occur:

(1) either v is never unassigned; the solving of pb is similar to the solving of pb without v, that is with $n-1$ variables;
(2) or v is unassigned; let k' be the first step at which v is unassigned; two sub-cases may occur:
   (a) either v is alone in the current inconsistency explanation; v's current value can be definitively removed from v's domain and the solving of pb amounts to its solving minus one value, that is with $sd-1$ values;
   (b) or v is not alone in the current inconsistency explanation $IE(k')$; because v has been chosen to be unassigned, its weight $w(v,k')$ is the smallest in $IE(k')$; there exists another variable $v' \in IE(k')$ the weight of which $w(v',k')$ is greater than $w(v,k')$; because there is a minimum distance $dMin > 0$ between the weights of two assigned variables, $w(v',k') \geq w(v,k') + dMin$; because the weight of a variable cannot decrease unless it is unassigned, $w(v,k') \geq w(v,k)$ and $w(v',k') \geq w(v,k) + dMin = w + dMin$; the maximum weight has been strictly increased between steps k and k'; the solving of pb amounts to its solving with a maximum weight at least equal to $w + dMin$.

Because the number of variables, the number of values, and the number of possible weights are finite, one faces three possible cases in a finite number of steps:

(1) a problem with only one variable: *decision repair* obviously terminates when solving it;
(2) a problem with an empty domain: *decision repair* terminates too;
(3) a problem pb' at step k' with a maximum weight w' such that no greater weight is possible ($w' > wMax - dMin$); let v' be the unique assigned variable of maximum weight; the same reasoning can be applied; either v' is never unassigned; we get pb' minus one variable; or v' is unassigned and is alone in the current inconsistency explanation; we get pb' minus one value; or v' is unassigned at step k'' and is not alone in the current inconsistency explanation; the maximum weight has been increased by

$dMin$ between steps k' and k''; this is impossible and this last case cannot occur.

Consequently, *decision repair* terminates in a finite number of steps.

□

Appendix B. Proof of Theorem 6.2

Some lemmas are necessary to prove Theorem 6.2. All these lemmas can be deduced from conditions of Theorem 6.2.

Lemma B.1. *If decision repair does not terminate, then* $\exists k_0 \geq 0$ *such that no definitive value removal occurs after step* k_0.

Proof. Let us assume the negation of this lemma: $\forall k \geq 0, \exists k' \geq k$ such that a value removal occurs at step k'. Then, $\exists k'' \geq k'$ such that a value removal occurs at step k'' and so on infinitely. Because the number of values in the problem is finite, this is impossible. This lemma is thus true. □

Lemma B.2. *If no definitive value removal can occur, then, at each backtrack at any step* k*, the current inconsistency explanation* $IE(k)$ *contains at least two variables.*

Proof. If the current inconsistency explanation $IE(k)$ was limited to one variable v, the value that is currently assigned to v could be definitively removed. In fact, if variables in $IE(k)$ are ordered according to an increasing weight, with any heuristic order between variables of same weight, then the first one is chosen to be unassigned and its weight is set to 0. Moreover, the weight of the second one is increased by at least βMin. □

Lemma B.3. *If no definitive value removal can occur, then the maximum weight over all the variables cannot decrease.*

Proof. Let us consider a step k. In case of assignment at step k, no variable weight decreases. Thus, the maximum weight does not decrease. In case of unassignment at step k, only the weight of the unassigned variable decreases. The only case where the maximum weight could decrease is the case where the unassigned variable v is alone of maximum weight. But, such a situation cannot occur. Indeed, according to Lemma B.2, the current inconsistency explanation contains at least another variable v'. If v was alone of maximum weight, v''s weight would be strictly smaller than v's weight and v' would have been chosen to be unassigned. Thus, the maximum weight does not decrease, in case of unassignment too. □

Lemma B.4. *If no definitive value removal can occur, then, after* a^n *backtracks, the maximum weight over all the variables is greater than or equal to* $a \cdot \beta Min$.

Proof. Before each backtrack b, let $W(b)$ be the vector of the variable weights, ordered according to decreasing values. The maximum weight $wmax(b)$ is the first element of this vector.

Before each backtrack b, let us consider another artificial vector $W_0(b)$ of same dimension, which would be bound by the following initialisation and updating rules: all the components of $W_0(0)$ are null; $W_0(b+1)$ is obtained from $W_0(b)$ by adding βMin to the smallest component of $W_0(b)$ and by reordering it according to decreasing values. In fact, βMin is always added to the last component of $W_0(b)$. If u_0 points out this updating rule, we write $W_0(b+1) = u_0(W_0(b))$. It is easy to show that the number of backtracks that are necessary to go from $W_0(0)$ to the vector $(a \cdot \beta Min, 0, \ldots, 0)$, with $a \cdot \beta Min$ for the first component and 0 for all the other ones, is smaller than or equal to a^n.

Moreover, if we consider the lexicographic comparison operator $\succ$, it can be shown that $\forall b, W(b) \succeq W_0(b)$. This is obviously true for $b = 0$, because weights are all positive or null. Let us assume that this is true for b. $W(b+1)$ results from $W(b)$ (1) by setting to 0 one component the location of which is i (the one associated with the unassigned variable), (2) by adding at least βMin to another component the location of which is j, with $j < i$, because the weight of the unassigned variable is one of the smallest in the current inconsistency explanation, (3) by increasing some null components due to the assignments that may occur between b and $b+1$, and finally (4) by reordering the resulting vector. Let $u_0(W(b))$ be the vector that would result from the application of the artificial updating rule u_0 to $W(b)$. $u_0(W(b))$ results from $W(b)$ by adding βMin to the last component and reordering the resulting vector. It is easy to show that $W(b+1)$ is lexicographically greater than or equal to $u_0(W(b))$, because $u_0(W(b))$ results from the adding of βMin to the last component of $W(b)$ and $W(b+1)$ results from the adding of at least βMin to a component of $W(b)$ which is not the last and thus is greater than or equal to the last. Because we assume that $W(b) \succeq W_0(b)$, we have $u_0(W(b)) \succeq u_0(W_0(b)) = W_0(b+1)$. Because we proved that $W(b+1) \succeq u_0(W(b))$, we have $W(b+1) \succeq W_0(b+1)$.

Because vector W is always lexicographically greater than or equal to vector W_0, the number of backtracks that are necessary to get $wmax(b) \geq a \cdot \beta Min$ is smaller than or equal to a^n. Because, according to Lemma B.3, $wmax$ cannot decrease, we get $wmax(a^n) \geq a \cdot \beta Min$.

□

Lemma B.5. *If no definitive value removal can occur, then the maximum weight over all the variables after b backtracks is greater than or equal to $(b^{1/n} - 1) \cdot \beta Min$.*

Proof. First, $\forall b \geq 0, \exists a$ such that $a^n \leq b \leq (a+1)^n$. From Lemma B.4, we have $wmax(a^n) \geq a \cdot \beta Min$. Because $b \geq a^n$, from lemma B.3, we have $wmax(b) \geq wmax(a^n) \geq a \cdot \beta Min$. Because $(a+1)^n > b$, we have $(a+1) > b^{1/n}$ and $a \geq b^{1/n} - 1$. From that, we get $wmax(b) \geq (b^{1/n} - 1) \cdot \beta Min$. □

Lemma B.6. *If no definitive value removal can occur, then the maximum weight over all the variables after k steps of the algorithm is greater than or equal to $((k/2 - n)^{1/n} - 1) \cdot \beta Min$.*

Proof. If we consider now the steps of the algorithm, with either assignment or unassignment, we can observe that the minimum number of backtracks in k steps

is equal to $k/2 - n$, because, in the worst case, the starting assignment is empty and the ending assignment is complete. Consequently, from Lemmas B.5 and B.3, if $wmax(k)$ is the maximum weight over all the variables at step k, we get $wmax(k) \geq ((k/2 - n)^{1/n} - 1) \cdot \beta Min$. □

Lemma B.7. *If no definitive value removal can occur, then* $\forall A, \exists k_1$ *such that the weight of at least one variable is strictly greater than* A *at step* k_1.

Proof. This a direct consequence of Lemmas B.6, by taking k_1 such that $((k_1/2 - n)^{1/n} - 1) \cdot \beta Min > A$ □

Lemma B.8. *Let* g *be the function that associates with any positive number* x *the value* $x + f(x)$.
$\forall k_1 \geq 0$, *if* $\exists k_0, 0 < k_0 \leq k_1$, *such that* $w(v, k_1) > g^{k_0 - 1}(\alpha Max)$, *then*
(1) $\forall k, 0 \leq k \leq k_0$, $g^{k_0}(w(v, k_1 - k)) \geq w(v, k_1)$ *and*
(2) $\forall k, 1 \leq k \leq k_0$, v *is not unassigned at step* $k_1 - k$.

Proof. Roughly speaking, this lemma allows us to use the weight of a variable v at step k_1 (in fact, a lower bound on this weight) to deduce some information about what happened with v between steps $k_1 - k_0$ and k_1 (with k_0 function of the lower bound).

First, note that, because f is a continuous, positive, and strictly increasing function, the same properties hold for g. Moreover, $\forall x \geq 0, g(x) \geq x$.

We establish this lemma via a recurrence on k_0.

For $k_0 = 1$, the starting assumption becomes $w(v, k_1) > \alpha Max$. v cannot have been unassigned at step $k_1 - 1$ because, if it had been the case, $w(v, k_1)$ would be null. v cannot have been assigned at step $k_1 - 1$ too because, if it had been the case, $w(v, k_1)$ would be less than or equal to αMax. Thus, a variable different from v has been assigned or unassigned at step $k_1 - 1$. In case of assignment, $w(v, k_1) = w(v, k_1 - 1)$. In case of unassignment, $w(v, k_1) \leq g(w(v, k_1 - 1))$. Because $g(w(v, k_1 - 1)) \geq w(v, k_1 - 1)$, in both cases, $w(v, k_1) \leq g(w(v, k_1 - 1))$. Consequently, properties (1) and (2) hold for $k = 1$. Moreover, property (1) holds for $k = 0$ because $g(w(v, k_1)) \geq w(v, k_1)$.

Let us assume that the lemma is true for any value $k_0 \leq k_1 - 1$ and let us try to establish it for $k_0 + 1$. The starting assumption becomes $w(v, k_1) > g^{k_0}(\alpha Max)$. Because $\forall x \geq 0, g(x) \geq x$, we have $g^{k_0}(\alpha Max) \geq \alpha Max$ and thus $w(v, k_1) > \alpha Max$. Exactly the same way as for $k_0 = 1$, we can infer that v has not been unassigned at step $k_1 - 1$ and that $w(v, k_1) \leq g(w(v, k_1 - 1))$. Because of the starting assumption $(w(v, k_1) > g^{k_0}(\alpha Max))$, we have $g(w(v, k_1 - 1)) \geq w(v, k_1) > g^{k_0}(\alpha Max)$. Because g is an strictly increasing function, we get $w(v, k_1 - 1) > g^{k_0 - 1}(\alpha Max)$. This is the starting assumption with k_0 and $k_1 - 1$ and we can use the recurrence assumption.

With this recurrence assumption, we have (1) $\forall k, 0 \leq k \leq k_0, g^{k_0}(w(v, k_1 - 1 - k)) \geq w(v, k_1 - 1)$ and (2) $\forall k, 1 \leq k \leq k_0$, v is not unassigned at step $k_1 - 1 - k$.

This can be rewritten as follows: (1) $\forall k, 1 \leq k \leq k_0 + 1, g^{k_0}(w(v, k_1 - k)) \geq w(v, k_1 - 1)$ and (2) $\forall k, 2 \leq k \leq k_0 + 1$, v is not unassigned at step $k1 - k$.

Because it has been established above that $w(v,k_1) \leq g(w(v,k_1-1))$ and because g is an increasing function, a new form of (1) can be deduced: (1) $\forall k, 1 \leq k \leq k_0+1, g^{k_0+1}(w(v,k_1-k)) \geq g(w(v,k_1-1)) \geq w(v,k_1)$.

If we add that (1) is true for $k=0$ because $g^{k_0}(w(v,k_1)) \geq w(v,k_1)$ and that (2) has been already established for $k=1$ (v cannot have been unassigned at step k_1-1), we get (1) $\forall k, 0 \leq k \leq k_0+1, g^{k_0+1}(w(v,k_1-k)) \geq w(v,k_1)$, and (2) $\forall k, 1 \leq k \leq k_0+1$, v is not unassigned at step k_1-k.

The lemma is true for k_0+1 and is thus true for any value of k_0. □

Lemma B.9. *If no definitive value removal can occur and if $\forall A, \exists k_1$ such that the weights of at least i variables ($i < n$) are strictly greater than A at step k_1, then $\forall A', \exists k_1'$ such that the weights of at least $i+1$ variables are strictly greater than A' at step k_1'.*

Proof. Let us consider any value $k_0 > 0$ and any value $A_0 > g^{k_0-1}(\alpha Max)$. According to the assumption of this lemma, $\exists k_1$ such that the weights of i variables are strictly greater than A_0 at step k_1. Let V_1 be this set of variables and V_1' its complement ($V_1' = V - V_1$). $\forall v \in V_1$, we have $w(v,k_1) > A_0 > g^{k_0-1}(\alpha Max)$. This means that the assumption of Lemma B.8 holds for every variable $v \in V_1$. Lemma B.8 allows us to infer that $\forall v \in V_1$, (1) $\forall k, 0 \leq k \leq k_0,\ g^{k_0}(w(v,k_1-k)) \geq w(v,k_1)$ and (2) $\forall k, 1 \leq k \leq k_0$, v is not unassigned at step k_1-k. Because g is a continuous strictly increasing function and because A_0 can be chosen strictly greater than $g^{k_0}(0)$, (1) can be written as follows: (1) $\forall k, 0 \leq k \leq k_0,\ w(v,k_1-k) \geq (g^{-1})^{k_0}(w(v,k_1))$. Because $w(v,k_1) > A_0$, (1) can be written again as follows: (1) $\forall k, 0 \leq k \leq k_0,\ w(v,k_1-k) \geq (g^{-1})^{k_0}(A_0)$. Two cases are now possible.

In the first one, the weight of at least one variable $v' \in V_1'$ is greater than or equal to the weight of at least one variable in $v \in V_1$ at a step $k_1-k, 0 \leq k \leq k_0$. We have $w(v',k_1-k) \geq w(v,k_1-k)$. Because $w(v,k_1-k) \geq (g^{-1})^{k_0}(w(v,k_1))$, we have $w(v',k_1-k) \geq (g^{-1})^{k_0}(w(v,k_1))$ too. Because $w(v,k_1) > A_0$, we have $w(v',k_1-k) \geq (g^{-1})^{k_0}(A_0)$.

In the second one, the weight of all the variables in V_1' remains strictly smaller than the weight of all the variables in V_1 at each step $k_1-k, 0 \leq k \leq k_0$. Because no variable in V_1 is unassigned between k_1-k_0 and k_1-1, at most $n \cdot (d-1)$ removal explanations involving only variables in V_1 are produced between both these steps. If more had been produced, the domain of a variable would have been wiped out, a backtrack would have occur, an inconsistency explanation involving only variables in V_1 would have been produced, and one variable in V_1 should have been unassigned. Consequently, there exists an interval of at least $(k_0/2-n)/(n \cdot (d-1))$ consecutive backtracks with no removal explanations involving only variables in V_1 ($k_0/2-n$ is the minimum number of backtracks in k_0 steps). In this interval, each time a backtrack occurs, the associated inconsistency explanation, which is the union of the removal explanations in the domain of the variable the domain of which has been wiped out, contains at least one variable in V_1'. In fact, it contains at least two variables in V_1', because the variable to be unassigned cannot not be chosen in V_1 and, if there were not another variable in V_1' in the inconsistency explanation, a removal explanation would be created involving only variables in V_1.

Consequently, the weight of a variable in V_1' is increased by at least βMin at each backtrack in this interval. From Lemma B.5, limited to variables in V_1', we can infer that the maximum weight in V_1' after these $(k_0/2-n)/(n\cdot(d-1))$ consecutive backtracks is greater than or equal to $[[(k_0/2-n)/(n\cdot(d-1))]^{1/(n-i)}-1]\cdot\beta Min$, which is greater than or equal to $[[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min$. This means that $\exists k, 0\leq k\leq k_0, \exists v'\in V_1'$ such that $w(v', k_1-k)\geq[[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min$.

We have chosen A_0 such that $A_0 > g^{k_0-1}(\alpha Max)$. But we can choose it such that moreover $A_0 > g^{k_0}([[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min)$.

Because $A_0 > g^{k_0}([[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min)$ implies that $(g^{-1})^{k_0}(A_0) > [[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min$, both cases can be merged. This allow us to say that, in both cases, $\exists k, 0\leq k\leq k_0, \exists v'\in V_1'$ such that $w(v', k_1-k)\geq[[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min$.

Because $\forall v\in V_1, \forall k, 0\leq k\leq k_0, w(v, k_1-k)\geq(g^{-1})^{k_0}(A_0) > [[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min$, we can now say that $\forall k_0, \exists k_1'$ such that the weights of $i+1$ variables are strictly greater than $[[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min$ at step k_1'.

Because $[[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min$ is an increasing function of k_0, $\forall A', \exists k_0$ such that $[[(k_0/2-n)/(n\cdot(d-1))]^{1/n}-1]\cdot\beta Min > A'$. Thus $\forall A', \exists k_1'$ such that the weights of $i+1$ variables are strictly greater than A' at step k_1' . □

Proof. All these lemmas allow us now to prove Theorem 6.2. Let us assume that *decision repair* does not terminate. According to Lemma B.1, $\exists k_0\geq 0$ such that no definitive value removal occurs after step k_0. According to Lemma B.7, $\forall A, \exists k_1 > k_0$ such that the weight of at least one variable is strictly greater than A at step k_1. Then, according to Lemma B.9, we can infer that $\forall i, 1\leq i, \leq n, \forall A, \exists k_1 > k_0$ such that the weights of at least i variables are strictly greater than A at step k_1. This applies for $i=n$. Thus, $\forall A, \exists k_1 > k_0$ such that the weights of all the variables are strictly greater than A at step k_1. But, αMax is an upper bound on the minimum weight over all the variables because, in case of a partial assignment, the weight of at least one variable is null and because, in case of a complete assignment, the weight of the last assigned variable is smaller than or equal to αMax. Setting $A > \alpha Max$ results in a contradiction. □