Travelling in the World of Local Searches in the Space of Partial Assignments*

Cédric Pralet and Gérard Verfaillie

LAAS-CNRS, Toulouse, France
{cpralet,gverfail}@laas.fr

Abstract. In this paper, we report the main results of a study which has been carried out about the multiple ways of parameterising a local search in the space of the partial assignments of a *Constraint Satisfaction Problem* (CSP), an algorithm which is directly inspired from the *decision repair* algorithm [1]. After a presentation of the objectives of this study, we present the generic algorithm we started from, the various parameters that must be set to get an actual algorithm, and some potentially interesting algorithm instances. Then, we present experimental results on randomly generated, but not completely homogeneous, binary CSPs, which show that some specific parameter settings allow such *a priori* incomplete algorithms to solve almost all the consistent and inconsistent problem instances on the whole constrainedness spectrum. Finally, we conclude with the work that remains to do if we want to acquire a better knowledge of the best parameter settings for a local search in the space of partial assignments.

1 Local search in the space of partial assignments

1.1 Depth-first tree search in constraint satisfaction

The basic algorithm, designed to solve *Constraint Satisfaction Problems* (CSP [2]), consists of a *depth first* search in the space of the partial assignments, organised into a tree the root of which is the empty assignment, leaves are complete assignments, and each node, except the root, results from the assignment of an unassigned variable in its father node. Variable and value heuristics are used to choose pertinently the next variable to assign and the value to assign to it [3]. Constraint propagation algorithms are also used to enforce any local consistency property, like for example *arc consistency*, at each node of the tree [4, 5]. These algorithms allow domains of the unassigned variables to be reduced and eventually wiped out. In the latter case, inconsistency of the associated subproblem is proven and another value for the last assigned variable is chosen. Producing and recording value removal explanations allow other forms of backtracking than this chronological one to be performed, like for example *conflict directed backjumping* [6].

* The study reported in this paper has been initiated when both authors were working at ONERA, Toulouse, France.

The main advantage of such a tree search is its completeness: if the problem instance is consistent, a solution is found; if not, inconsistency is established. Another advantage is its limited space complexity, which is only a linear function of the instance size. But its lack of scalability in terms of CPU-time prevents it from being used for solving large instances: at least for the hardest instances at the frontier between consistency and inconsistency, its time complexity is an exponential function of the instance size.

1.2 Local search in constraint satisfaction

On the contrary, local search algorithms perform an unordered search in the space of the complete assignments: each move goes from a complete assignment to another one in its neighbourhood, generally randomly chosen with a heuristic-biased probability distribution (choice of the variable to unassign and of the new value to assign to it). When solving CSPs, each neighbour assignment is evaluated via constraint checking [7]. Meta-heuristics, such as *simulated annealing*, *tabu search*, or others, allow various search schemes to be defined, including *restart* mechanisms [8].

A first advantage of local search is its limited space complexity, which is, as with depth first tree search, only a linear function of the instance size. Its main advantage is however its far better scalability in terms of CPU-time. But it suffers from at least three shortcomings:

- its incompleteness: if the problem instance is consistent, there is no guarantee that a solution will be found; if not, there is no mechanism which allows inconsistency to be established;
- its non deterministic behaviour: when random choice mechanisms are used, running twice the same algorithm on the same instance may give different results;
- its difficulty handling constraints: on the one hand, because local search is basically an optimisation method, it has some difficulty handling correctly together constraints and criterion; on the other hand, because only complete assignments are considered, constraint propagation is not used and is replaced by a simple constraint checking.

1.3 Hybridisations between tree search, local search, and constraint propagation

This landscape, which has been recognised for a long time [9], pushed researchers and practitioners to explore combinations of tree search, local search, and constraint propagation and to propose various hybridisation schemes.

Many of these schemes led to loose combinations: two kinds of search on the same problem in sequence or in parallel [10, 11]; a kind of search on a master subproblem and another kind on the complementary slave subproblem [12]. Stronger combinations have been however proposed with large neighbourhood local searches [13–15]: when neighbourhoods become large, they can no longer be enumerated, as they are in basic local search; combinations of tree search and constraint propagation become thus good candidates for exploring them.

1.4 A stronger hybridisation scheme between local search and constraint propagation

Under the name of *decision repair*, a stronger hybridisation scheme has been proposed in [1]. It originated from the following observations: strong combination between search and deduction, such as exists between tree search and constraint propagation, is not possible with basic local search, because it manages only complete assignments; but, it is possible as soon as one considers a local search which manages partial assignments.

More precisely, *decision repair* starts from any assignment A (empty, partial, or complete). It applies any local consistency enforcing algorithm (constraint propagation) on the associated subproblem. In case of consistency, the neighbourhood is the set of assignments that result from the assignment of any unassigned variable in A. In case of inconsistency, it is on the contrary the set of assignments that result from the unassignment of any assigned variable in A. This procedure is iterated until a complete consistent assignment (a solution) has been found, inconsistency has been proven, or a time limit has expired.

1.5 Expected gains and costs

Expected gains and costs can be analysed from the local search and tree search starting points.

- From the local search starting point, constraint propagation from partial assignments may avoid either exploring uselessly locally inconsistent subproblems, or considering inconsistent values in locally consistent subproblems. But, because constraint propagation from a partial assignment is far more costly than constraint checking on a complete assignment, it is certain that the time taken by a local move will increase. In fact, we hope that, as it has been observed for the combination between tree search and constraint propagation, the decrease in number of moves will compensate for the increase in the cost of each move. To get this result, it is necessary to design constraint propagation algorithms which are both incremental and decremental, that is which support efficiently variable assignments as well as unassignments[1].
- From the tree search starting point, the main novelty is the complete freedom of choice of the variable to unassign. With *chronological backtracking*, the last assigned variable is systematically unassigned. With *conflict directed*

[1] With depth first tree search, we need only incremental algorithms which support variable assignments, because the search is performed in a tree. Any backtrack, even in case of backjumping, comes back to a previously visited partial assignment for which constraint propagation has been already performed and its results recorded for example on a stack. If we allow any assigned variable to be unassigned in case of backtrack, the search is performed no more in a tree, but in a neighbourhood graph. The partial assignment that results from a backtrack may have never been visited. Results of constraint propagation are not available and must be decrementally computed. This is what occurs with *dynamic backtracking* [16, 17], which is a particular case of *decision repair*.

backjumping [6], all the assigned variables from the last one to the last one involved in the current conflict are unassigned. With *dynamic backtracking* [16], which is no longer a tree search, only the last one involved in the current conflict is unassigned. With *partial order backtracking* [18, 19], still more freedom is available by following only a partial order between variables. In fact, *decision repair* is similar to *incomplete dynamic backtracking* [20]. As does *incomplete dynamic backtracking*, it lays down completeness for efficiency and scalability. Potentially, this freedom allows the main shortcoming of depth first tree search (the possibility of getting stuck in an inconsistent subtree because of wrong assignments of the first variables) to be fixed (see Figure 1). But, as it has been already said, the need for incremental and decremental constraint propagation algorithms may lead to more costly assignments and unassignments. We hope that the more intelligent search will compensate for that.

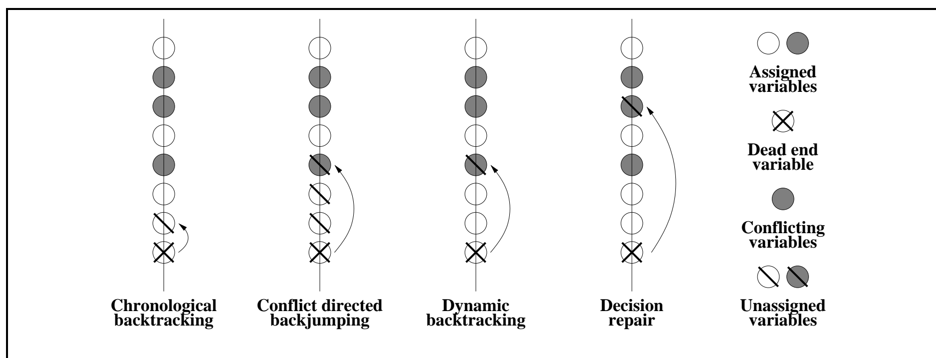

Fig. 1: Various forms of backtracking.

1.6 Study objectives and article organisation

In [1], N. Jussien and O. Lhomme proposed a specific instance of *decision repair*, called *tabu decision repair*, and evaluated it on open-shop scheduling problems. The objective of our study was slightly different: starting from a minimal *decision repair* scheme, we aimed at listing all the parameters that must be set to get an actual algorithm, and at exploring and evaluating *a priori* interesting algorithm instances. In other words, our main objective was not to exhibit an efficient instance, but to acquire knowledge about all the possible parameter settings and their relative efficiency, with the long-term objective of acquiring about local search in the space of partial assignments of CSPs the same level of knowledge as has been acquired about depth-first tree search.

After a presentation of the generic algorithm we started from in Section 2, we present its degrees of freedom in Section 3, some possible algorithm instances in Section 4, the results of the experiments we carried out in Section 5, and directions for future work in Section 6.

In this paper, we only consider satisfaction problems (no optimisation criterion) involving unary and binary constraints, although *decision repair* can deal with constraints of any arity.

2 A generic algorithm

We started from a very generic version of *decision repair* (see Figure 2) derived from the first algorithm presented in [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
```

Fig. 2: 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 (sets of assigned variables the assignments of which imply inconsistency) 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. 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 3 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 *?*

Fig. 3: Possible outputs on consistent and inconsistent problem instances.

To put things in a more concrete form, we show in Figures 5 and 6 possible traces of a *decision repair* algorithm on a consistent graph colouring problem P_c and on an inconsistent one P_i (see Figure 4).

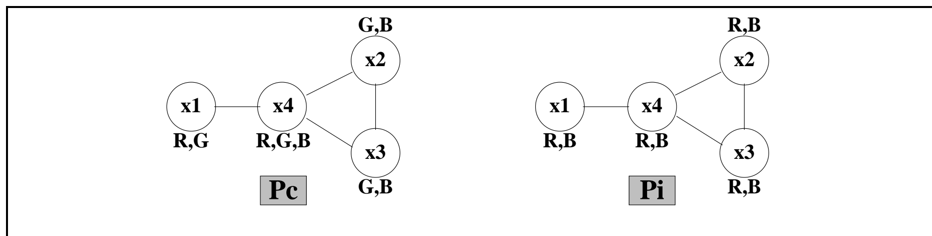

Fig. 4: A consistent graph colouring problem P_c and an inconsistent one P_i.

In both cases, after each assignment of a variable v, the algorithm performs forward checking from v to the current domains of the unassigned 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 unassigned 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 5), 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.

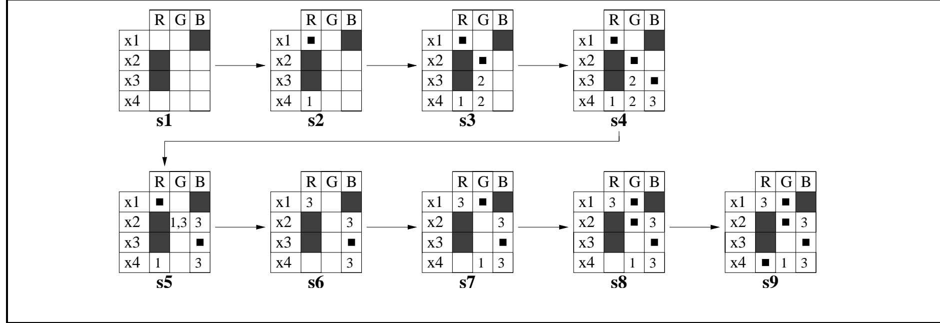

Fig. 5: A possible execution of *decision repair* on P_c.

On P_i, in state s_6 (see Figure 6), 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 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.

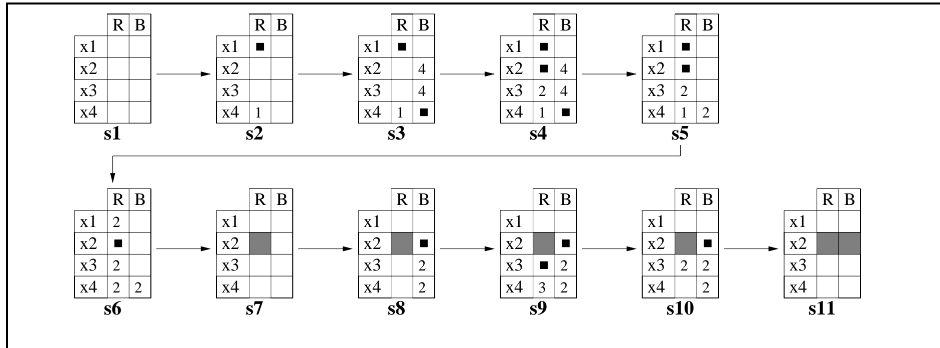

Fig. 6: A possible execution of *decision repair* on P_i.

3 Parameter settings

We can now list all the parameters that take place in such an algorithm:

1. the *local consistency property which is enforced* via constraint checking or constraint propagation at each step of the algorithm (functions *Initial_Filter*, *Incremental_Filter*, and *Decremental_Filter*). If a partial assignment is considered as consistent when all the completely assigned constraints are satisfied, we get the local search counterpart of *backward checking*. If it is considered as consistent when no domain of any unassigned variable is wiped out by forward checking, we get the local search counterpart of *forward checking* [4]. If it is considered as consistent when no domain of any unassigned variable is wiped out by *arc consistency* enforcing, we get the local search counterpart of *MAC* [5]. Other forms of local consistency may be obviously considered;
2. the *way local consistency is enforced* at the beginning of the search and at each step of the search after variable assignment or unassignment. Questions arise particularly about variable unassignments (function *Decremental_Filter*) and the way of avoiding propagating constraints again from scratch after each unassignment. If no information is recorded when propagating constraints, constraint graph information can be used to limit the work to do in case of unassignment [21, 22]. If the variable assignment order is recorded, it can also be used to limit this work. In [20], *conflict counts* associate with any value of any variable the number of current conflicting assignments and are used with *forward checking* to know at each step of the algorithm if a value of a variable belongs or not to its current domain (null or not null counter). Finally, if value removal justifications or explanations are recorded when propagating constraints, more value removals can be saved in case of unassignment [23–25, 17];
3. the *way these value removal explanations are handled* when they are produced and recorded. Can more than one removal explanation be recorded per value (functions *Incremental_Filter* and *Decremental_Filter*)? Must irrelevant removal explanations (inconsistent with the current assignment) be maintained (function *Decremental_Filter*)? When irrelevant removal explanations are maintained, how should the learning memory size be limited?
4. the *variables and values that are affected by constraint propagation* (functions *Initial_Filter*, *Incremental_Filter*, and *Decremental_Filter*). With depth first tree search, only the variables that are not assigned yet and the values in their domains that have not been removed yet are affected by constraint propagation. This is justified by the fact that the search goes always forward: more variables are assigned, more values are removed in the domains of the unassigned variables. The backward moves are not free and use a stack to recover previous states. With *decision repair*, the search goes freely forward and backward: any variable can go at any step from the unassigned to the assigned state and inversely; the same way, any value of any variable can go at any step from the present to the removed state and inversely. In

such conditions, it may be interesting both for the sake of efficiency and of information quality to perform constraint propagation on all the variables (unassigned and assigned) and all the values of the initial domains (present or removed). Such a systematic computing is obviously more costly, but can make the computing at each step easier and provide information for better heuristic choices (see below);

5. the *heuristics* that are used by function *Extend_Assignment to choose the variable to assign* in case of local consistency and *the value to assign to it.* All the studies that have been carried out about that in the context of depth first tree search [3] are *a priori* reusable in this larger local search context;
6. the *heuristics* that are used by function *Repair_Assignment to choose the variable to unassign* in case of inconsistency. Because this choice is the main novelty of *decision repair* with regard to depth first tree search, completely new heuristics must be designed and experimented for that. Various criteria should be *a priori* taken into account to judge the interest in unassigning a variable v of current value $a(v)$, initial domain $d(v)$, and current domain $d'(v)$: (i) the fact that v is involved in the current inconsistency, whatever the way this inconsistency is computed (constraint graph information, value removal explanations ...), (ii) the number of constraints in which v is involved (its degree in the constraint graph), (iii) the number of values in the domains of the other variables that are inconsistent with $a(v)$ (in the consistency graph), (iv) the quality of $a(v)$ with regard to any static heuristic ordering of $d(v)$, (v) the size of $d(v)$ and $d'(v)$, (vi) the removal explanations already produced and recorded for the values in $d(v) - d'(v)$, (vii) the rank of v in the current assignment order, (viii) the value removal explanations that would be destroyed and forgotten in the domains of the other variables in case of unassignment of v .... Note that we can build from these criteria unassignment heuristics which aim at building a consistency proof (a solution) by removing bad choices (for example, to unassign a variable the value of which is inconsistent with the greatest number of values in the domains of the other variables) and other ones which aim at building an inconsistency proof (for example, to unassign a variable the domain size of which is the smallest, or a variable for which the number of destroyed value removal explanations in the domains of the other variables would be the smallest in case of unassignment, in order to destroy as less as possible the proof in progress);
7. the presence or absence of priority for assignment (resp. unassignment) to the variable that has been unassigned (resp. assigned) just before, when an assignment (resp. unassignment) immediately follows an unassignment (resp. assignment). See for example [26]. Such priorities limit the choice of the variable to assign or to unassign in these situations, but favour the production of inconsistency proofs. When both priorities are present, we say that the associated algorithm perseveres.
8. finally, the *stopping criterion* which is used by function *Stop* when no result *yes* or *no* has been already produced: CPU-time, number of assignment extensions or repairs ...

4 Some algorithm instances

This very generic algorithm scheme includes many known instances.

- *Depth first tree search* with *chronological backtracking* is an instance of *decision repair* where the last assigned variable is systematically chosen to be unassigned in case of inconsistency. In such conditions, the current state of the algorithm is a partial assignment equipped with an assignment order and the neighbourhood graph becomes a tree. A stack can be used to avoid performing constraint propagation again in case of unassignment. Completeness is guaranteed without the help of any inconsistency explanation.
- *Dynamic backtracking* [16, 17] is another instance where value removal explanations are produced and recorded as long as they remain relevant (consistent with the current assignment) and where, in case of inconsistency, the last assigned variable involved in the current inconsistency explanation is systematically chosen to be unassigned. The neighbourhood graph is no more a tree. A stack cannot be used anymore. But completeness is guaranteed.
- *Partial order backtracking* [18, 19] is an extension of *dynamic backtracking* where, in case of inconsistency, an assigned variable, involved in the current inconsistency explanation and following a partial order between variables which results from the previously recorded value removal explanations, is chosen to be unassigned. Completeness is still guaranteed.
- *Incomplete dynamic backtracking* [20] is another instance where constraint propagation is performed via *forward checking* on all the variables (assigned or not) and all their values (removed or not), where *conflict counts* associate with any value of any variable the number of current conflicting assignments, and where the choice of the variable to unassign[2] in case of inconsistency is free (either random or following a given heuristics). As a consequence of this complete freedom, completeness is no more guaranteed.
- Even *min conflicts* [7] can be considered as an instance of *decision repair* where constraint checking is only performed on complete assignments (all the partial assignments are assumed to be consistent), where the explanation for the inconsistency of a complete assignment is the union of the variables of the unsatisfied constraints, where, in case of inconsistency, a variable is randomly chosen in the current inconsistency explanation to be unassigned, and where a new value is randomly chosen among those that minimise the number of unsatisfied constraints if they were selected. In absence of inconsistency explanation recording, completeness is not guaranteed.

Beyond these known algorithms and without any exhaustivity intention, new potentially interesting algorithm instances could be considered round specific unassignment heuristics.

[2] In fact, *incomplete dynamic backtracking* allows several variables to be unassigned at the same time. The number of unassigned variables at each backtrack step is a new algorithm parameter. This option could be introduced in the generic *decision repair* scheme.

- A first obvious option consists in choosing the variable to unassign randomly in the current inconsistency explanation, as in *min conflicts*, whatever the way this explanation is built. Such a heuristic we refer to as *UHrand* could be profitably integrated into a global *randomisation* and *restart* scheme, inspired from [27].
- A second option consists in choosing the variable to unassign randomly among the variables of the current inconsistency explanation the assignment of which is the most doubtful. If a static heuristic value is associated with each value of each variable, we can for example consider that the doubtful nature of an assignment is measured by the difference between the heuristic values of the first and of the second value. Such a heuristic we refer to as *UHmostdoubt* aims at undoing quickly doubtful choices in order to produce solutions.
- A third option consists in choosing the variable to unassign randomly among the variables of the current inconsistency explanation that are the least involved in value removal explanations in the domains of the other variables. Such a heuristic we refer to as *UHmindestroy* aims at keeping recorded as long as possible value removal explanations in order to build inconsistency proofs. It can be improved by giving higher weights to removal explanations that have been built by backtrack than to removal explanations that have been directly built by constraint propagation, because the first ones generally needed more work to be produced, and by giving higher weights to smaller removal explanations. It can be also improved by performing constraint propagation on all the variables and not only on the unassigned ones in order to get more precise heuristic information.

5 Experiments

We carried out experiments on many problems and instances, with many algorithmic variants, but we report in this paper only the ones that have been carried out on randomly generated, but not completely homogeneous, binary CSPs, with a limited number of algorithmic variants: the ones that appeared to be potentially more efficient than the classical tree or local search algorithms.

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 7 and 8 have been obtained with $n = 50$, $d = 10$, $nc = 5$, $p_1 = 30$, $p_{1c} = 20$, and p_2 varying between 30 and 40 around the complexity peak. 10 instances have been generated for each value of p_2. Note that, for $p_2 \leq 32$, all the generated instances are consistent,

that, for $p_2 \geq 34$, all of them are inconsistent, and that, for $p_2 = 33$, only one generated instance out of 10 is inconsistent.

5.2 Algorithms

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

Except MC, all of 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 current domains. Except for the fourth variant of DR (DR(mindestroy,avar)), it is only performed on the unassigned variables. Except MC, all of these algorithms persevere. Except for MC, the assignment heuristics 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 four variants of DR have in common the choice of the variable to unassign inside the current inconsistency explanation. However, they differ mainly in the way of making this choice.

- DR(rand) uses *UHrand* as an unassignment heuristic (see Section 4).
- DR(mostdoubt) uses *UHmostdoubt* as an unassignment heuristic (see Section 4) and *VHmineff* as a static value heuristic. This value heuristic results from 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.
- DR(mindestroy,uvar) uses *UHmindestroy* as an unassignment heuristic (see Section 4).
- DR(mindestroy,avar) differs from DR(mindestroy,uvar) only in that forward checking is performed on all the variables.

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 120 seconds. A CPU-time of 120 seconds is associated with any run which did not finish by the deadline. Figure 7 reports the median CPU-time as a function of p_2. Figure 8 reports the number of runs which did not finish by the deadline, among the 200 $(20 \cdot 10)$ for each value of p_2. 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 similar and both clearly better than those of BT.

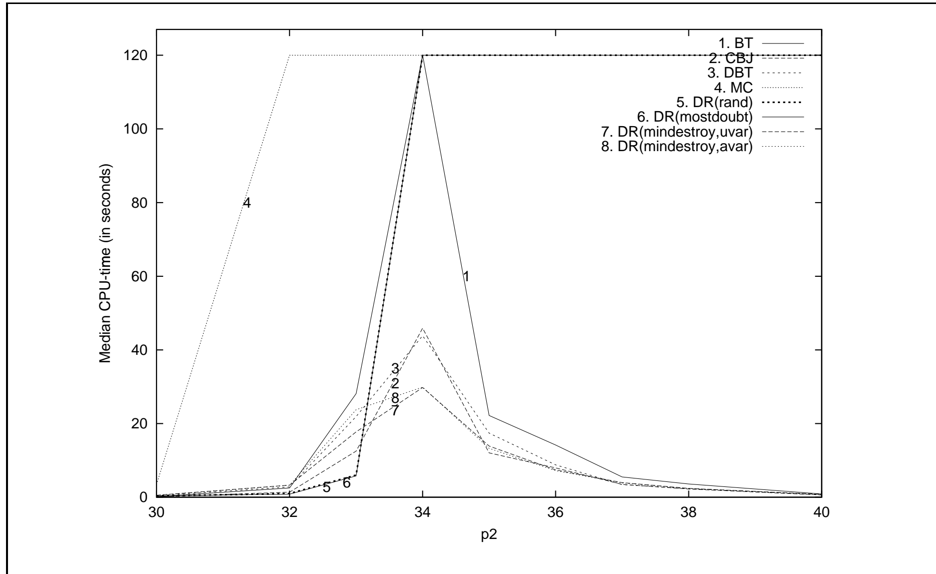

Fig. 7: Median CPU-time on randomly generated problem instances.

- DR(rand) and DR(mostdoubt) produce quasi identical results and present a behaviour which is similar to that of MC. Although they are the most efficient on consistent instances at the consistency/inconsistency frontier, they are, as MC and other classical local search algorithms, unable to solve inconsistent instances.
- Although they are basically local search algorithms in the space of partial assignments, and thus *a priori* incomplete, DR(mindestroy,uvar) and DR(mindestroy,avar) present the same behaviour as do complete algorithms such as BT, CBJ, and DBT. Moreover they are the most efficient on inconsistent instances at the consistency/inconsistency frontier and allow the greatest number of instances to be solved by the deadline. Note a small advantage for DR(mindestroy,uvar) when compared with DR(mindestroy,avar) in terms of CPU-time on consistent instances at the consistency/inconsistency frontier: performing forward checking on all the variables is too costly. On the contrary, note a small advantage for DR(mindestroy,avar) when compared with DR(mindestroy,uvar) in terms of number of solved instances at this frontier: performing forward checking on all the variables provides the algorithm with a better unassignment heuristic.

6 Future work

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

- Are there unassignment heuristics that can better solve consistent instances and other ones that can better solve inconsistent ones? If we know nothing

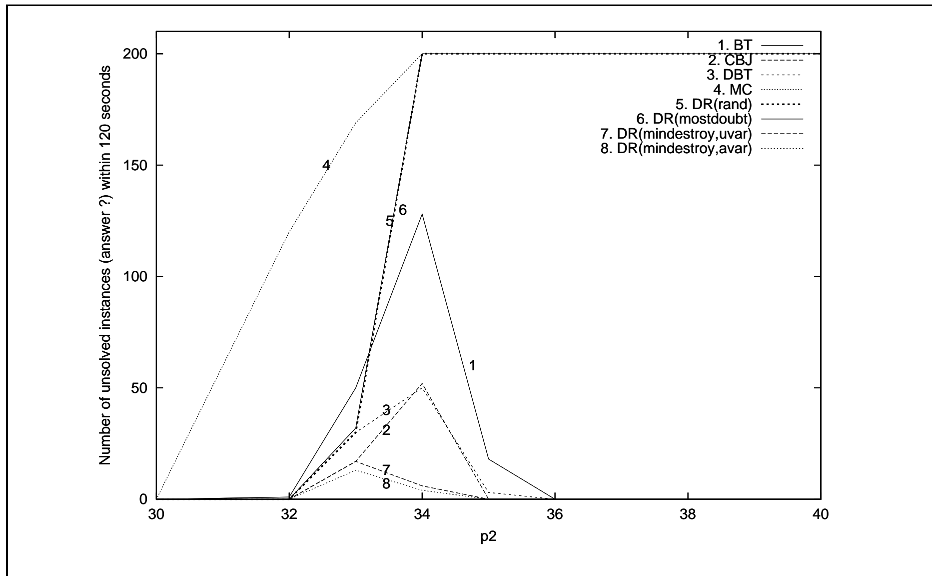

Fig. 8: Number of unsolved problem instances within 120 seconds.

about consistency or inconsistency, would it be profitable to run two algorithms concurrently, one with the objective of building a solution and the other one with the objective of building an inconsistency proof?

- What must be the unassignment heuristics to allow inconsistent instances to be solved? Can we define sufficient conditions on these heuristics to guarantee algorithm termination without any stopping criterion, and thus algorithm completeness? Can we, still better, define necessary conditions?
- 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 3?

Acknowledgements Many thanks to Narendra Jussien and Olivier Lhomme for the *decision repair* algorithm and particularly to Narendra for fruitful discussions about this algorithm.

References

1. Jussien, N., Lhomme, O.: Local Search with Constraint Propagation and Conflict-based Heuristics. Artificial Intelligence **139** (2002)
2. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press Ltd. (1993)
3. Dechter, R., Pearl, J.: Network-based Heuristics for Constraint Satisfaction Problems. Artificial Intelligence **34** (1987)
4. Haralick, R., Elliot, G.: Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artificial Intelligence **14** (1980)

5. Sabin, D., Freuder, E.: Contradicting Conventional Wisdom in Constraint Satisfaction. In: Proc. of ECAI (1994)
6. Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problems. Computational Intelligence **9** (1993)
7. Minton, S., Johnston, M., Philips, A., Laird, P.: Minimizing Conflicts: a Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. Artificial Intelligence **58** (1992)
8. Aarts, E., Lenstra, J., eds.: Local Search in Combinatorial Optimization. John Wiley & Sons (1997)
9. Langley, P.: Systematic and Nonsystematic Search Strategies. In: Proc. of AAAI (1992)
10. Kask, K., Dechter, R.: GSAT and Local Consistency. In: Proc. of IJCAI (1995)
11. Schaerf, A.: Combining Local Search and Look-Ahead for Scheduling and Constraint Satisfaction Problems. In: Proc. of IJCAI (1997)
12. Zhang, J., Zhang, H.: Combining Local Search and Backtracking Techniques for Constraint Satisfaction. In: Proc. of AAAI (1996)
13. Pesant, G., Gendreau, M.: A View of Local Search in Constraint Programming. In: Proc. of CP (1996)
14. Shaw, P.: Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In: Proc. of CP (1998)
15. Lobjois, L., Lemaître, M., Verfaillie, G.: Large Neighbourhood Search using Constraint Propagation and Greedy Reconstruction for Valued CSP Resolution. In: Proc. of the ECAI Workshop on "Modelling and Solving with Constraints" (2000)
16. Ginsberg, M.: Dynamic Backtracking. Journal of Artificial Intelligence Research **1** (1993)
17. Jussien, N., Debruyne, R., Boizumault, P.: Maintaining Arc-Consistency within Dynamic Backtracking. In: Proc. of CP (2000)
18. Ginsberg, M., McAllester, D.: GSAT and Dynamic Backtracking. In: Proc. of KR (1994)
19. Bliek, C.: Generalizing Partial Order and Dynamic Backtracking. In: Proc. of AAAI (1998)
20. Prestwich, S.: Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search. Annals of Operations Research **115** (2002)
21. Berlandier, P., Neveu, B.: Maintaining Arc Consistency through Constraint Retraction. In: Proc. of ICTAI (1994)
22. Georget, Y., Codognet, P., Rossi, F.: Constraint Retraction in CLP(FD): Formal Framework and Performance Results. Constraints : An International Journal **4** (1999)
23. Bessière, C.: Arc-Consistency in Dynamic Constraint Satisfaction Problems. In: Proc. of AAAI (1991)
24. Debruyne, R.: Arc-Consistency in Dynamic CSPs is no more Prohibitive. In: Proc. of ICTAI (1996)
25. Fages, F., Fowler, J., Sola, T.: Experiments in Reactive Constraint Logic Programming. Journal of Logic Programming **37** (1998)
26. Sabin, D., Freuder, E.: Understanding and Improving the MAC Algorithm. In: Proc. of CP (1997)
27. Gomes, C., Selman, B., Kautz, H.: Boosting Combinatorial Search Trough Randomization. In: Proc. of AAAI (1998)