Guiding Backtrack Search by Tracking Variables During Constraint Propagation

Gilles Audemard, Christophe Lecoutre, Charles Prud'Homme

Guiding Backtrack Search by Tracking Variables During Constraint Propagation

Gilles Audemard
CRIL, Univ. Artois & CNRS, France

Christophe Lecoutre
CRIL, Univ. Artois & CNRS, France

Charles Prud'homme
TASC, IMT-Atlantique, LS2N-CNRS, France

Abstract

It is well-known that variable ordering heuristics play a central role in solving efficiently Constraint Satisfaction Problem (CSP) instances. From the early 80's, and during more than two decades, the dynamic variable ordering heuristic selecting the variable with the smallest domain was clearly prevailing. Then, from the mid 2000's, some adaptive heuristics have been introduced: their principle is to collect some useful information during the search process in order to take better informed decisions. Among those adaptive heuristics, `wdeg/dom` (and its variants) remains particularly robust. In this paper, we introduce an original heuristic based on the *midway* processing of failing executions of constraint propagation: this heuristic called `pick/dom` tracks the variables that are directly involved in the process of constraint propagation, when ending with a conflict. The robustness of this new heuristic is demonstrated from a large experimentation conducted with the constraint solver ACE. Interestingly enough, one can observe some complementary between the early, midway and late forms of processing of conflicts.

2012 ACM Subject Classification Computing methodologies → Discrete space search

Keywords and phrases Variable Ordering Heuristics, Variable Weighting

Digital Object Identifier 10.4230/LIPIcs.CP.2023.9

Funding This work has benefited from the support of the National Research Agency under France 2030, MAIA Project ANR-22-EXES-0009.

1 Introduction

Backtrack search remains a classical approach for solving instances of the Constraint Satisfaction Problem (CSP), and the related Constraint Optimization Problem (COP). It is based on depth-first exploration, which is conducted by instantiating variables in sequence and backtracking when dead-ends occur. For efficiently exploring the search space, a filtering process is performed at each step of the search so as to reduce the domains of the variables; typically most of the constraints guarantee the property known as (generalized) arc consistency [5, 1, 26, 14]. The order in which variables are chosen during the depth-first traversal of the search space is decided by a *variable ordering heuristic* H. At each internal node of the search tree built by the backtrack search algorithm, the next variable x is selected by H, and a value is assigned to x according to a *value ordering heuristic*. Choosing the right heuristics for solving a given constraint network is a key issue since different heuristics can lead to drastically different search trees.

In modern constraint solvers, three main principles are considered for guiding search (i.e., performing depth-first exploration):

- First, one should start by assigning variables that belong to the most difficult part(s) of the problem instance. This principle is derived from the recognition that there is no point in traversing the easy part(s) of an instance and then backtracking repeatedly when

it turns out that the first choices are incompatible with the remaining difficult part(s). Here the underlying *fail-first* principle is [13]: "To succeed, try first where you are most likely to fail".

- Second, value selection should be based on the *succeed-first* or *promise* principle, which comes from the simple observation that to find a solution quickly, it is better to move at each step to the most promising subtree, primarily by selecting a value that is most likely to participate in a solution.
- Third, when starting to build the search tree, one should pay attention to the initial variable/value choices that are particularly important. Indeed, bad choices near the root of the search tree may turn out to be disastrous because they lead to exploration of very large fruitless subtrees. To make good initial choices, one strategy is to select the first branching decisions with special care, perhaps calling sophisticated and expensive procedures for this purpose. Another relevant strategy is to restart search several times, ideally learning some information each time in order to refine search guidance.

The current response (in solvers) to following these principles is as follows:

- Generic *adaptive* variable ordering heuristics that learn from conflicts during exploration are usually employed; a classical such heuristic being `wdeg/dom` [3], possibly combined with a mechanism, called last-conflict reasoning (lc) simulating a certain form of intelligent backjumps [18].
- Promising attempts to select values (or pairs variable-value) according to some elegant mechanisms such as Belief Propagation [23] have been introduced, but, unfortunately, we are not aware of any generic robust value ordering heuristic. One exception may be BIVS (Bound-Impact Value Selector) [8], but controlling its computation cost remains a difficult question. Consequently, it is rather frequent that the first (smallest) value be the default choice. Interestingly, for optimization, it is highly recommended to use in priority the value present in the last found solution, which is a technique known as solution(-based phase) saving [27, 7], clearly in concordance with the *promise* principle (as initially mentioned in [6, 10]).
- To address the issue of heavy-tailed runtime distributions [11], the search is restarted regularly, following a geometric progression (or the Luby sequence). Besides, by collecting nogoods [17] along the leftmost branch of the search tree at the end of each run (i.e., just before restarting), we have the guarantee of never exploring again the same parts of the search space (which is a nice feature when exhibiting all distinct solutions of a CSP instance).

This is the context of our contribution. More specifically, we focus our interest on the first crucial component: variable ordering heuristics. Indeed, in this paper, we introduce an original heuristic called `pick/dom` that learns from conflicts by identifying the variables that are directly involved in the process of constraint propagation (when failing). We show how to implement it in the variable-oriented propagation scheme. The robustness of this new heuristic is demonstrated by conducting a large experimentation with the well-known constraint solver ACE [15].

2 Preliminaries

A *Constraint Network* (CN) is composed of a finite set of n variables $\mathcal{X}$, and a finite set of e constraints $\mathcal{C}$. Each variable x must be assigned a value from its current domain, denoted by $\texttt{dom}(x)$. Each constraint c represents a mathematical relation over an ordered set of variables,

called the *scope* of c, and denoted by $\texttt{scp}(c)$. The *arity* of a constraint c is the size of its scope. The *degree* of a variable x is the number of constraints of $\mathcal{C}$ involving x. A *solution* to a CN $P = (\mathcal{X}, \mathcal{C})$ is the assignment of a value to each variable of $\mathcal{X}$ such that all constraints of $\mathcal{C}$ are satisfied. A constraint network is *satisfiable* iff it admits at least one solution. The *Constraint Satisfaction Problem (CSP)* is to determine whether a given constraint network is satisfiable, or not. A classical approach for solving this NP-complete problem is to perform a depth-first search with backtracking, while filtering domains after each taken decision. This procedure builds a binary search tree $\mathcal{T}$: for each internal node ν of $\mathcal{T}$, a pair (x, v) is selected where x is a variable and v is a value in $\texttt{dom}(x)$. Then, two cases are considered: the assignment $x = v$ (positive decision) and the refutation $x \neq v$ (negative decision). The *future* variables of a constraint c, denoted by $\texttt{fut}(c)$, are the variables in $\texttt{scp}(c)$ at a given node of the search tree that have not been explicitly assigned by the search algorithm.

When an objective function (integer or real-valued function defined on a subset of variables of $\mathcal{X}$) is added to the constraint network, we obtain an instance of the *Constraint Optimization Problem (COP)*. Backtrack search for COP relies on an optimization strategy based on decreasingly updating the maximal bound (assuming minimization) whenever a solution is found; this is a kind of ramp-down strategy (related to Branch and Bound), whose principle is equivalent (still assuming a minimization problem) to adding a special objective constraint $\texttt{obj} < \infty$ to the constraint network (although it is initially trivially satisfied), and to update the limit of this constraint whenever a new solution is found.

3 Variable Ordering Heuristics

We provide in this section a quick overview of popular general-purpose search heuristics. The simple variable ordering heuristic `dom` [13], which selects variables in sequence of increasing size of domain, has long been considered as the most robust backtrack search heuristic. However, twenty years ago, *adaptive* heuristics were introduced: they take into account information collected along the part of the search space (tree) already explored.

In this paper, we shall mainly focus our attention to the popular adaptive heuristics based on constraint weighting (`wdeg`, `wdeg/dom`, `cacd`, `chs`), and failure rating (`frba/dom`). We will also refer to `impact`, `activity` and counting-based heuristics, which are defined as follows:

- `impact`, or `ibs` (Impact-Based Search), selects in priority the variable with the highest impact. The impact of a variable x gives a measure about the importance of x in reducing the search space [25]. The size of the search space of a CN P is the product of all current domain sizes:
$$\texttt{size}(P) = \prod_{x \in \mathcal{X}} |\texttt{dom}(x)|$$
The impact I of a variable assignment $x = a$ on P is computed as follows:
$$I(x = a) = 1 - \frac{\texttt{size}(P')}{\texttt{size}(P)}$$
where $P' = \phi(P|_{x=a})$ denotes the CN obtained after assigning x to a and running the filtering process ϕ (e.g., enforcing arc consistency). Note that if P' leads to a failure, then $I(x = a) = 1$. It is easy to see that this heuristic can be used for value selection as well.
- `activity`, or `abs` (Activity-Based Search), selects in priority the variable with the highest activity. The activity of a variable x is roughly measured by the number of times the domain of x is reduced during search [22]. This heuristic is motivated by the key role of propagation in constraint programming and relies on a decaying sum to forget the oldest statistics progressively. More formally, the activity $A(x)$ of a variable x is updated at

each (new) node of the search tree (after a decision has been taken by the solver followed by constraint propagation) regardless of the outcome (success or failure) by the following two rules:

- $A(x) = A(x) * \gamma$, where $0 \leq \gamma \leq 1$ is an age decay parameter, if the domain of x has not been affected (i.e., has not been reduced)
- $A(x) = A(x) + 1$ otherwise

The activities are initialized by making random probing in the search space.

- Counting-based search relies on computing the solution density of each variable-value assignment for a constraint in order to build an integrated variable-selection and value-selection heuristic [24]. Depending on the constraints, computing such information can carry a high computational cost although some mechanisms have been proposed to accelerate it [9].

Now, to introduce `wdeg` and `wdeg/dom`, we need to describe the way constraint propagation is run each time a decision is taken by the backtrack search algorithm. Algorithm 1 describes the constraint-oriented propagation scheme, which uses of a set of constraints for piloting propagation. This simplifies the presentation here, whereas later, we will introduce the new heuristic `pick/dom` in the context of the variable-oriented scheme. Initially, the set Q contains the whole set of constraints of the constraint network. Then, each constraint c in Q is picked in turn and a filtering process is applied from c: typically, this is for enforcing arc-consistency (or a partial form) by calling `filter`(`c`) at Line 4. The call to this function returns a subset of variables involved in c, denoted by X, whose domains have been modified (i.e., such that at least one value has been removed from each of these domains). By means of X, we can update Q so as to ensure constraint propagation is run until a fixed point is reached. If ever the domain of one variable of X is empty, it simply means that a conflict occurred (a dead-end has been identified) and so, a backtrack is required. This is triggered by the returned Boolean value `false`, after having called the function `incrementWeight` with the culprit constraint (responsible for the domain wipeout) passed as a parameter. In the initial paper [3], the principle of constraint weighing is very simple: the weight of the culprit constraint c is incremented by 1.

Algorithm 1 propagate($(\mathcal{X},\mathcal{C})$: CN): Boolean.

```
1  Q ← C
2  while Q ≠ ∅ do
3      pick and delete c from Q
4      X ← filter(c)   // X are variables in scp(c) with reduced domains
5      if ∃x ∈ X | dom(x) = ∅ then
6          incrementWeight(c)
7          return false                 // detected inconsistency
8      foreach c' ∈ C | c' ≠ c ∧ X ∩ scp(c') ≠ ∅ do
9          Q ← Q ∪ {c'}
10 return true
```

To summarize, each constraint c admits a weight, initially set to 1, which is incremented whenever a domain wipeout occurs while filtering c. Importantly, it was observed experimentally that it was more effective to consider only the future variables involved in a culprit constraint. Technically, instead of associating a global weight c.`weight` with each constraint

c, one can introduce a local weight $c.\texttt{weight}[x]$ to be associated with each variable x in $\texttt{scp}(c)$. Hence, when a conflict occurs, instead of incrementing the weight $c.\texttt{weight}$ of the culprit constraint, one can decide to increment the local weight $c.\texttt{weight}[x]$ of each future variable involved in $\texttt{scp}(c)$.

The heuristics `wdeg` and `wdeg/dom` are defined as follows:

- `wdeg` selects in priority the future variable with the highest "weighted degree". Each variable x is given a weighted degree, which is the sum of the weights over all constraints involving x and at least another future variable. For each future variable x, the score of x according to `wdeg` is:

$$\sum_{c \in \mathcal{C} : x \in \texttt{scp}(c) \wedge |\texttt{fut}(c)|>1} c.\texttt{weight}[\texttt{x}]$$

- `wdeg/dom` selects in priority the future variable with the highest ratio "weighted degree to current domain size". For each future variable x, the score of x according to `wdeg/dom` is:

$$\frac{\sum_{c \in \mathcal{C} : x \in \texttt{scp}(c) \wedge |\texttt{fut}(c)|>1} c.\texttt{weight}[\texttt{x}]}{|\texttt{dom}(x)|}$$

In classical forms of `wdeg` and `wdeg/dom`, counters are incremented by 1., which remains very simplistic and does not differentiate between constraints. This is why constraint weighting, in the so-called variant `cacd` [28], has been refined by exploiting as information both the "current arity" of the culprit constraint (i.e., the number of future variables) and the size of the current domains of the future variables. The increment values computed for the classical and `cacd` variants are precisely shown in Algorithm 2.

Algorithm 2 incrementWeight(c: Constraint).

```
1 foreach x ∈ fut(c) do
2     if variant cacd then
3         c.weights[x] ← c.weights[x] + 1/(|fut(c)|×(1+|dom(x)|))
4     else
5         c.weights[x] ← c.weights[x] + 1
```

Note that to break ties, which correspond to sets of variables that are considered as equivalent by the heuristic, one can use a second criterion. However, for adaptive heuristics, it is usual that the first encountered variable with the best score is selected.

Finally, we introduce two recent heuristics: the former, `chs` [12], exploits the history of search failures, while the latter, `frba/dom` [20], computes the proportion of failing assignments for each variable.

- `chs` (Conflict-History Search), selects in priority variables appearing in recent failures. All failures are registered with a timestamp. More precisely, `chs` maintains for each constraint c, a score $q(c)$ and updates it at every domain wipeout with an exponential recency weighted average:

$$q(c) = (1 - \alpha) \times q(c) + \alpha \times r(c)$$

where $\alpha = 0.4$ (decreasing as time goes by) and $r(c)$ is the reward given when a domain wipeout occurred. The reward is higher when the constraint frequently enters in conflict:

$$r(c) = \frac{1}{\#\texttt{Conflicts} - \texttt{Conflict(c)} + 1}$$

where #Conflicts is the total number of conflicts and Conflict(c) stores the last value of #Conflicts when c led to a failure. The conflict history score of a variable x which will be used in selecting the branching variable is given by:

$$\frac{\sum_{c \in \mathcal{C} : x \in \mathtt{scp}(c) \wedge |\mathtt{fut}(c)|>1} q(c) + \delta}{|\mathtt{dom}(x)|}$$

where δ is a positive real number close to 0 that avoids random selection at the beginning of search.

- frba/dom (Failure Rate Based Search), selects in priority the variables that most often lead to a conflict when assigning them. For this purpose, two counters are associated with each variable x: the former, #Conflicts(x), records the number of times a failure has been observed just by propagating an assignment involving x, and the latter, #Assigns(x), the number of times the variable x was assigned. The failure rate of a variable x is then:

$$fr(x) = \frac{\#\mathtt{Conflicts(x)}}{\#\mathtt{Assigns(x)}}$$

In addition, similarly to the factor used for chs, we compute:

$$a(x) = \frac{1}{\#\mathtt{Conflicts} - \mathtt{Conflict(x)} + 1}$$

where #Conflicts is the total number of conflicts and Conflict(x) stores the last value of #Conflicts when x being assigned led to a failure.
The failure rate score of a variable x by frba/dom is then:

$$\frac{fr(x) + a(x)}{|\mathtt{dom}(x)|}$$

4 Variable Tracking in Conflicting Propagation

In this section, we introduce the principle of Variable Tracking in Conflicting Propagation (VTCP). More specifically, we introduce a new variable ordering heuristic called pick/dom, whose principle is to track the variables that are used to trigger filtering operations during constraint propagation. However, it is important to note that this tracking is only used to update counters (called "pick degrees") associated with variables when constraint propagation ends with a conflict. We show how to implement such variable tracking within the variable-oriented propagation scheme.

To record information about tracked variables, we just need to associate a counter pick[x] with each variable x of the CN. Initially, this counter ("pick degree") is set to 0. According to the selected mode (see below) used to update these counters, recorded values may be in real or integer forms. The heuristic pick/dom is simply defined as follows:

- pick/dom selects in priority the future variable with the highest ratio "pick degree to current domain size". For each future variable x, the score of x according to pick/dom is:

$$\frac{\mathtt{pick[x]}}{|\mathtt{dom}(x)|}$$

In the rest of this section, we show how pick degrees are computed.

In ACE, constraint propagation follows the variable-oriented scheme (as initially introduced in [21]): the set Q contains variables. The principle is that whenever a value is removed

from the domain of a variable x, this variable is added to Q. In a first step, by ignoring any statement related to L and Δ_x, we can rather easily recognize the variable-oriented propagation scheme in Algorithm 3: as long as there is a variable in Q, one of them, x is picked, and we execute the filtering algorithms (propagators) attached to all constraints involving x, while updating Q when necessary.

Algorithm 3 propagate($(\mathcal{X}, \mathcal{C})$: CN): Boolean.

```
1  L = ⟨⟩
2  Q ← 𝒳
3  while Q ≠ ∅ do
4      pick and delete x from Q
5      Δ_x ← 0
6      for c ∈ 𝒞 | x ∈ scp(c) do
7          X ← filter(c, Δ_x)            // Δ_x is updated during call
8          if ∃y ∈ X | dom(y) = ∅ then
9              append (x, Δ_x) to L
10             incrementPick(L)
11             return false               // detected inconsistency
12         Q ← Q ∪ X
13     if Δ_x > 0 then
14         append (x, Δ_x) to L
15 return true
```

Algorithm 4 incrementPick(L: Sequence).

```
1  foreach i ranging from 1 to |L| do
2      (x_i, Δ_{x_i}) ← L[i]
3      switch VARIANT do
4          case 0 do
5              increment ← 1
6          case 1 do
7              increment ← Δ_{x_i}
8          case 2 do
9              increment ← 100 × Δ_{x_i} / Σ_{j=1}^{|L|} Δ_{x_j}
10         case 3 do
11             increment ← (n − depth)/n × 100 × Δ_{x_i} / Σ_{j=1}^{|L|} Δ_{x_j}
12     pick[x_i] ← pick[x_i] + increment
```

Now, let us consider first the structure L, which is a list, initially empty, keeping track of any variable x that plays a role (i.e., triggers some effective filtering) during propagation, together with an information indicating the degree Δ_x of this role. Notice that, since L is a list, the same variable may occur several times. Concerning the local variable Δ_x, it is initialized to 0 in Line 5. When the loop starting at Line 6 ends, Δ_x indicates how many

values have been deleted from the domain of x in the different calls to Function `filter` at Line 7. In practice, it is possible to handle Δ_x in a non-intrusive way by introducing a global variable whose value is incremented whenever a value is deleted (whatever the domain is). If at Line 13, the value of Δ_x is 0, it means that no filtering/reduction was performed at all since the time x was picked. This is then a useless "pick", which is the reason why we do not update the structure L at Line 14. Importantly, the list L is only exploited if Algorithm 3 returns `false` (because a conflict is detected). Before returning `false`, the last picked variable is added to L (because we have the guarantee of some filtering) and the function `incrementPick` is called in order to update some picked degrees.

The way picked degrees are updated is shown in Algorithm 4. Four modes (denoted by values ranging from 0 to 3) are possible. In mode 0, the picked degree of any occurrence of a variable present in L is incremented by 1. In mode 1, the increment is given by Δ_x, the impact of x after having been picked. In mode 2, each time constraint propagation is run, 100 points are shared according to the relative impacts of the variables present in L. In mode 3, a coefficient is applied to 100, depending on the current depth of the solver. As a first extreme case, the current depth is 0, which means that we are at the root node, and so, 100 points are spread. As a second extreme case, the current depth is n, meaning that we are a leaf, and 0 point is shared. The rationale is that we give more importance to nodes near the top of the search tree.

5 Experimental Results

In our experiments, we have compared general-purpose variable ordering heuristics based on constraint weighting, failure rating and variable tracking during conflicting propagation, with the constraint solver ACE [15]. More specifically, we have compared the four variants of `pick/dom` with `wdeg-cacd` [28], `wdeg/dom` [3] and `chs` [12], as well as the recently introduced `frba/dom`. From now on, for simplicity, `pick/dom`, `wdeg-cacd` and `frba/dom` will be referred as `pick` (while appending the mode in subscript text), `cacd` and `frba`, respectively. Note that `ibs` and `abs` are not retained in our experiments because they are usually outperformed when used in ACE. Concerning the value ordering heuristic, it systematically chooses the smallest value in domains.

We have considered two benchmarks, denoted by `xcsp-csp` and `xcsp-cop`, which are respectively composed of all CSP and COP instances selected for the main tracks of the XCSP^3 competitions (In 2019, instances were randomly selected from existing series, and there were no competitions held in 2020 and 2021) organized in 2017, 2018 [16] and 2022 [2] (most of them generated by the Python library PyCSP^3 [19]). They correspond to two full sets of 942 and 1,034 instances in format XCSP^3 [4], for exactly 77 and 50 problems, respectively. A time limit of 1,200 seconds was given per instance.

Ranking. Results will be partly analyzed from the scoring function used for the 2022 XCSP^3 competition. For self-containedness, we recall it now. The number of points won by a solver S is decided as follows:

- for CSP, this is the number of times S is able to solve an instance, i.e., to decide the satisfiability of an instance (either exhibiting a solution, or indicating that the instance is unsatisfiable)
- for COP, this is, roughly speaking, the number of times S gives the best known result, compared to its competitors. More specifically, for each instance I:

- if I is unsatisfiable, 1 point is won by S if S indicates that the instance I is unsatisfiable, 0 otherwise,
- if S provides a solution whose bound is less good than another one (found by another competing solver), 0 point is won by S,
- if S provides an optimal solution, while indicating that it is indeed the optimality, 1 point is won by S,
- if S provides (a solution with) the best found bound among all competitors, this being possibly shared by some other solver(s), while indicating no information about optimality: 1 point is won by S if no other solver proved that this bound was optimal, 0.5 otherwise.

5.1 Global Overview of Results

We start this experimental section with a global overview of the obtained results. The scores of tested heuristics on `xcsp-csp` and `xcsp-cop` are given in Table 1 and Table 2. First, let us make some comments on CSP. Here, the default version of ACE is the best one (`cacd`). The heuristics are relatively close, and the differences mainly come from the number of (solved) SAT instances. On our benchmark, the heuristic `frba` appears to be the worst one. Concerning the `pick` variants, they are quite close, even if $\texttt{pick}_3$ is the one that is most often the fastest heuristic.

Table 1 Ranking on `xcsp-csp`. For each heuristic, we report the number of SAT/UNSAT instances, the total number of solved instances and the number of times a heuristic is the fastest for solving an instance. We also report the result for the virtual best solver/heuristic (VBS).

Heuristic	Solved	SAT	UNSAT	Fastest
VBS	676	481	195	676
`cacd`	**646**	**457**	**189**	411
`chs`	636	449	187	390
$\texttt{pick}_3$	632	441	**189**	**442**
$\texttt{pick}_0$	631	442	188	439
$\texttt{pick}_1$	630	441	**189**	427
`wdeg/dom`	626	442	183	377
$\texttt{pick}_2$	625	437	188	434
`frba`	618	431	187	391

For COP, all `pick` variants are the most efficient heuristics: indeed, the gap with the other heuristics is significant. The variant $\texttt{pick}_3$ is the best one in terms of the number of proved optima and the number of found best bounds. The heuristics based on constraint weighting got quite similar results, and even if the heuristic `frba` appears to be also outperformed, it is important to report that this heuristic is very good in proving optima.

In order to provide a deeper analysis of the results of our experiments, we propose, in the next section, to focus only on the following three heuristics:

- `cacd` as a robust representative of the heuristics based on constraint weighting. The results of `cacd`, `wdeg/dom` and `chs` are quite similar on COP but `cacd` appears to be the most efficient heuristic on CSP. In addition, this is the default search strategy of ACE.
- `frba` as a recent proposed heuristic based on failure rating.

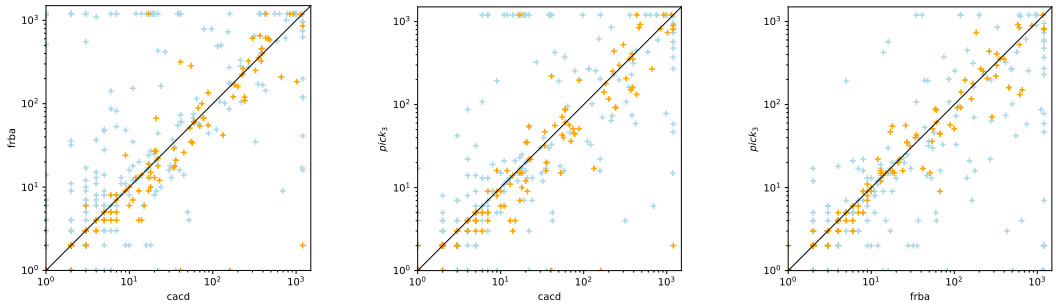
Table 2 Ranking on `xcsp-cop`. For each heuristic, we report the number of proved optima, the number of times the best bound has been found and the score as computed for the 2022 XCSP[3] competition. We also report the results for the virtual best solver/heuristic (VBS).

Heuristic	Score	Optimum	Best Bound
VBS	959.00	372	956
`pick`$_3$	**624.50**	**347**	**627**
`pick`$_2$	618.00	343	621
`pick`$_1$	617.50	336	621
`pick`$_0$	612.50	341	617
`wdeg/dom`	569.50	312	583
`cacd`	557.50	315	570
`frba`	557.00	341	560
`chs`	554.50	324	563

- `pick`$_3$ as the best variant of variable tracking in conflict propagation. Actually, it is the best heuristic on COP, able to find, most often, best bounds and also to prove them. On CSP, it is also the most efficient variant of VTCP and the fastest one.

5.2 Comparing Best Heuristics

In this section, we compare the three selected heuristics, namely, `cacd`, `frba` and `pick`$_3$. We start this comparison on the `xcsp-csp` benchmark with Figure 1, which shows some classical scatter plots (permitting to compare two algorithms with rather good precision).



(a) `cacd` vs `frba`. **(b)** `cacd` vs `pick`$_3$. **(c)** `frba` vs `pick`$_3$.

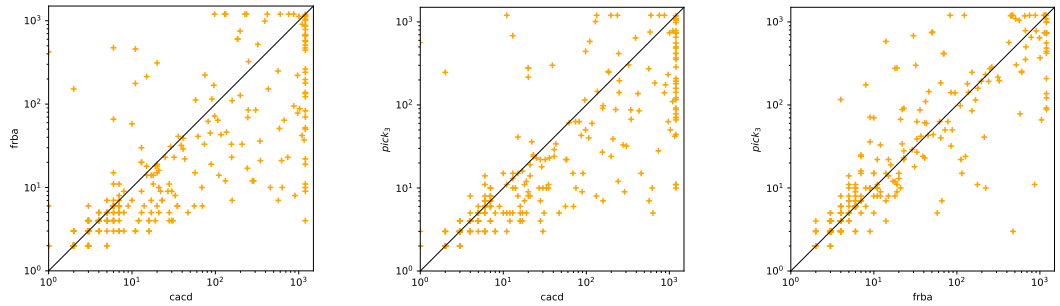
Figure 1 Scatter plots for CSP instances. Each dot represents an instance: its value on the x-axis (resp. y-axis) represents the time needed by the heuristic labelling the x-axis (resp. y-axis) to solve it. Blue (resp. Orange) dots corresponds to SAT (resp. UNSAT) instances. The dots below the diagonal represent then the instances where the y-axis heuristic is faster than the x-axis one.

Here, it is clear that `frba` is less efficient than `cacd` and `pick`$_3$. Even if `cacd` is the best heuristic on CSP (see Table 1), notably on the hardest instances, an instance-by-instance comparison between `cacd` and `pick`$_3$ looks less obvious, as the dots are uniformly distributed over both parts separated by the diagonal.

As an alternative for visualizing these results, a Venn diagram is depicted in Figure 4a. In such a diagram, each circle represents the instances solved by a heuristic. An overlapping region represents a set of instances solved in an *equivalent manner* by two heuristics, or

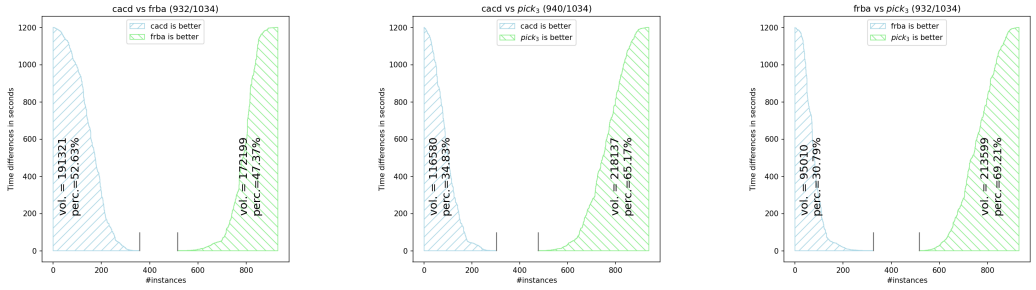
three in the case of the central region. For CSP, two heuristics are equivalent if they require the same amount of time (with a tolerance of one second) to find the same result (SAT or UNSAT). A region with no overlap emphasizes the instances that are better solved by a single heuristic. Here, the ranking is clear: $\texttt{pick}_3$ is the winner, followed by `cacd` and finally `frba`. To summarize the results for CSP: although `cacd` is the most robust heuristic when considering the number of solved instance (with a timeout set to $1,200$ seconds), $\texttt{pick}_3$ is usually the fastest heuristic.

Next, we consider the `xcsp-cop` benchmark. A first analysis can be made from the scatter plots in Figure 2. Each plot is based only on the instances whose optimality has been proved by at least one of the two compared heuristics. On the one hand, one can see in Figure 2a and Figure 2b that `frba` and $\texttt{pick}_3$ are better at proving optimality than `cacd`. On the other hand, Figure 2c does not show any dominance between `frba` and $\texttt{pick}_3$.



(a) `cacd` vs `frba`. **(b)** `cacd` vs $\texttt{pick}_3$. **(c)** `frba` vs $\texttt{pick}_3$.

Figure 2 Scatter plots for COP instances. Each dot represents an instance: its value on the x-axis (resp. y-axis) represents the time needed by the heuristic labelling the x-axis (resp. y-axis) to prove its optimum. The dots below the diagonal represent then the instances where the y-axis heuristic is faster than the x-axis one.

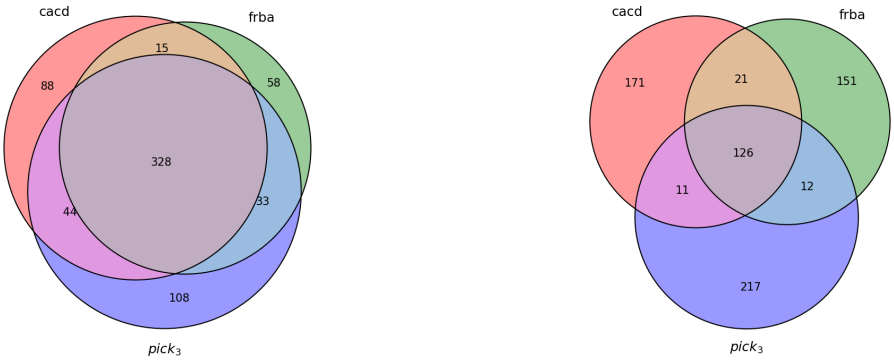


(a) `cacd` vs `frba`. **(b)** `cacd` vs $\texttt{pick}_3$. **(c)** `frba` vs $\texttt{pick}_3$.

Figure 3 Plots for COP Instances. Above each figure, the number of instances (partially) solved by at least one of the two heuristics as well as the total number of instances is indicated. When a heuristic is better than the other, this contributes to its area, representing the difference in resolution time. The *vol.* value indicates the volume of this surface, the *perc.* value indicates the ratio of the volume to the sum of the two volumes. The difference between two heuristics takes into account the best bounds found and possibly the proof of optimality.

Figure 3 allows a complementary analysis. Each graphics shows a pairwise comparison with two areas. When a heuristic is better than the other, the difference in solving time is

computed. An heuristic a is better than a heuristic b if it proves optimality whereas b does not, or if it finds a better bound than b, or, finally, if they find the same bound and it is faster than b. In the two first cases, the resolution time of b is set to the time limit, namely $1,200$. Each computed time difference contributes to the area of the best heuristic. Hence, a larger area means that the attached heuristic offers better performances than the other one. Each area is annotated with *vol.* and *perc.* which respectively denote the volume of the area and the ratio, in percentage, of the volume to the sum of the two volumes. Finally, strictly equivalent instances are indicated between the 2 vertical bars. In Figure 3a, one can observe that `cacd` is slightly more robust than `frba`. Interestingly, in Figure 3b and Figure 3c, the competitiveness of $\texttt{pick}_3$ is clearly visible: the area of $\texttt{pick}_3$ is almost twice as large as that of `cacd` or `frba`.



(a) Venn Diagram on CSP Instances.

(b) Venn Diagram on COP Instances.

Figure 4 Venn diagrams for `cacd`, `frba` and $\texttt{pick}_3$ on CSP and COP instances. Each circle represents the instances solved by a heuristic. An overlapping region represents a set of instances solved in an equivalent manner by two heuristics, or three in the case of the central region. Two heuristics are considered as being equivalent if they found the same result in the same amount of time. For CSP, this is the time for proving (un)satisfiability. For COP, this is for getting the best bound or proving optimality. A region not overlapped emphasizes the instances that are better solved by a single heuristic.

These results are confirmed by the Venn diagram in Figure 4b. The circles, and especially the one for $\texttt{pick}_3$, stand out from the centre. This Venn diagram also suggests that each heuristic would be more suitable for certain problems. This is confirmed in Table 3. First, `frba` performs particularly well on five problems, namely Cutstock, DC, ItemsetMining OpenStacks and TravelingSalesman. Indeed, it is able to close more instances and provides better bounds than `cacd` and $\texttt{pick}_3$. In turn, `cacd` behaves better on six problems: CyclicBandwith, Ramsey, StillLife and Taillard, and to a lesser extent PseudoBoolean and QueenAttacking. As for $\texttt{pick}_3$, the set of problems where it dominates is clearly larger: Auction, BinPacking, ChessBoardColoration, CoinsGrid, EchelonStock2, FAPP, GraphColoring, MultiAgentPathFinding, NurseRostering, OPD, QuadraticAssignment, RCPSP, RLFAP, Spot, SteelMillSlab, SumColoring TAL, TravelingTournament, Triangular and Warehouse. This is just under half of the problems. For some problems the gap is quite large, namely BinPacking, FAPP or RCPSP.

Finally, we show in Table 4 some details for some chosen (representative and singular)

instances of some problems. This may be helpful for testing and reproducing the results we have obtained.

Table 3 Details per problem (COP). For each problem, the number of instances is displayed, and for each heuristic, the couple 'number of proved optima : number of best bounds found' is provided. Best results are highlighted (when at least one heuristic is outperformed by the other(s)).

Problem	cacd	frba	pick$_3$	Problem	cacd	frba	pick$_3$
AirCraft (13)	4:**10**	**5**:6	4:**10**	NursingWork (12)	1:2	1:**4**	1:**4**
Auction (16)	2:6	2:4	2:**14**	OPD (17)	8:9	**9**:14	**9**:**15**
BACP (24)	4:24	4:24	4:24	OpenStacks (16)	12:16	**14**:16	12:16
BinPacking (51)	1:17	0:25	**2**:**33**	PeacableA (14)	4:**9**	4:**9**	4:8
BusScheduling (10)	1:5	1:5	1:**8**	PizzaVoucher (10)	4:7	**6**:8	5:**10**
CVRP (10)	0:2	0:2	0:**8**	PseudoB (30)	13:**23**	13:18	13:22
ChessBoard (17)	**3**:13	2:13	**3**:**14**	Quadratic (36)	6:26	6:14	6:**27**
ClockTriplet (10)	2:**9**	2:6	2:**9**	QueenAtt (17)	7:**11**	5:3	**8**:6
CoinsGrid (10)	2:7	2:4	**3**:**9**	Rack (4)	2:4	**3**:4	**3**:4
CrosswordDes (13)	3:**5**	3:3	3:**5**	Ramsey (17)	**5**:**17**	4:16	**5**:14
CutStock (17)	6:9	**8**:9	7:9	RCPSP (43)	31:34	34:35	**35**:**43**
CyclicBand (12)	**4**:**9**	2:3	2:6	RLFAP (50)	10:22	**12**:32	**12**:**48**
DC (26)	7:14	10:**25**	**12**:18	Spot (10)	2:7	2:6	2:**8**
EchelonStock2 (10)	0:9	0:7	0:**10**	SteelMill (17)	2:2	2:2	2:**7**
FAPP (18)	2:2	2:7	**3**:**13**	StillLife (47)	**15**:**46**	14:21	13:29
FastFood (17)	17:17	17:17	17:17	SumColoring (14)	4:8	4:7	4:**10**
Filters (8)	8:8	8:8	8:8	Taillard (51)	5:**17**	5:6	5:13
GolombRuler (47)	**19**:31	18:27	18:**35**	TAL (20)	9:16	**10**:15	**10**:**18**
GraphCol (28)	14:**24**	15:23	**17**:**24**	TemplateDes (15)	**12**:**13**	**12**:**13**	11:12
ItemsetMining (15)	3:10	**7**:**12**	6:10	TravelingTour (14)	2:7	2:2	2:**10**
Knapsack (31)	22:31	**26**:31	**26**:31	TravelingSale (29)	3:16	3:**17**	3:15
LowAuto (31)	12:21	12:**23**	**14**:19	Triangular (10)	0:3	**1**:5	**1**:**9**
Mario (10)	10:10	10:10	10:10	VRP (17)	0:**9**	0:1	0:**9**
MultiAgentP (20)	4:6	8:9	**9**:**17**	WarOrPeace (10)	6:10	6:10	6:10
NurseRost (41)	2:16	**3**:3	**3**:**29**	WareHouse (9)	0:3	0:0	0:**9**

6 Discussion

The three different ways of exploiting conflicts for guiding search, as experimented in the last section, show somewhat complementary behaviors. This can be explained by the fact that information is extracted at different moments: at the very beginning of the process conducting to a conflict (i.e., at the time of the decision), during constraint propagation, or at the time the last propagator (filtering algorithm) is solicited. One can then refer to such approaches as *early* (E), *midway* (M) and *late* (L) operational treatment of conflicts. This is illustrated in Figure 5 where a new decision $x = a$ is taken, when solving a CN P, in the continuity of two previously taken decisions $v = a$ and $w \neq b$. In our scenario, running constraint propagation ϕ on (the current state of) P after having assigned the value a to x, i.e., $\phi(P|x = a)$, reveals a new conflicting (dead-end) situation (denoted by $\bot$). The early processing of this new conflict consists in considering the variable x involved in the decision as the main culprit. This is the principle behind the heuristic `frba/dom`. The midway processing of this conflict consists in considering all variables having played a role (i.e., having been picked) during propagation as having contributed to the failure. This is the principle underlying the heuristic `pick/dom`. The late processing of this conflict consists in considering the last constraint (here, c^2_w) provoking a domain-wipeout (i.e., removing the last value of a domain) as the object of interest. This is the principle of constraint weighting, as in `wdeg/dom`.

One related heuristic, which is based on some form of midway strategy, is `abs`. However, VTCP and `abs` collect information according to different strategies. `abs` updates activity

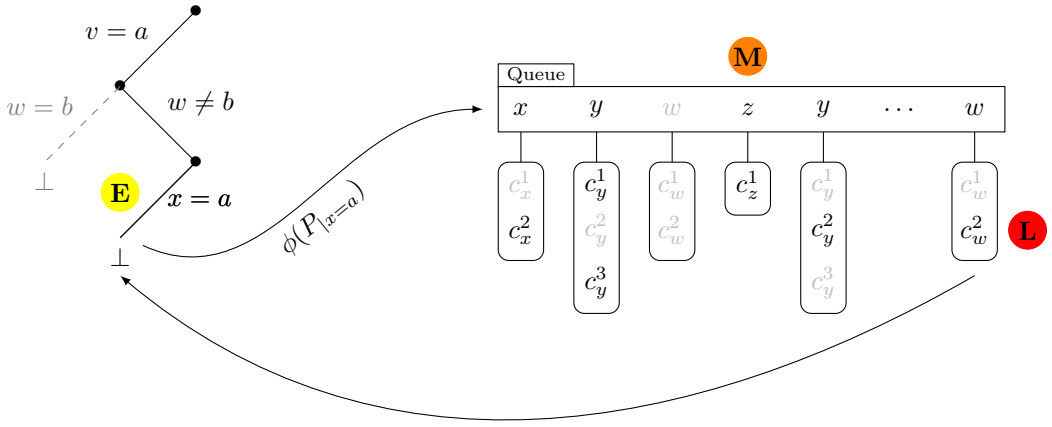


Figure 5 Illustration of pivotal moments for collecting information about conflicts: this correspond to early (E), midway (M) and late (L) processing of conflicts.

counters at each node, whereas VTCP only considers conflicting nodes. `abs` systematically updates the activity counters of each variable, whereas VTCP only increments the counters of variables at the origin of calls to effective propagators.

Importantly, we do believe that the experimental results we have obtained are significant, for several reasons. First, they can be reproduced in an open-source constraint solver (with a repository available in GitHub). Second, the number of models and instances used for our experiments is very large, involving more than 120 problems of various nature. Third, ACE is a competitive constraint solver and (although not officially engaged) showed good performances in (notably the COP main track) of the 2022 XCSP^3 competition.

Finally, it is true that the importance of variable tracking in conflicting propagation looks more limited when solving CSP instances. Actually, solving a (satisfiable) CSP instance involves one single phase: finding a solution, whereas solving a COP instance involves two subsequent phases: moving down towards an optimal solution, and proving optimality. We believe that `pick/dom` is rather efficient for the first phase of COP solving (for the second phase, a learning mechanism like in Picat [29] or OR-Tools becomes central).

7 Conclusion

In this paper, we have introduced a new way of exploiting conflicts during backtrack search so as to build a well-informed variable ordering heuristic. In contrast to existing heuristics relying on the early and late treatments of failing nodes, this new heuristic, `pick/dom`, consists in a midway processing of conflicts by tracking variables during constraint propagation. The robustness of `pick/dom` has been demonstrated from a vast experimentation campaign involving more than 120 problems (and around $2,000$ instances). Interestingly, the three different forms of exploiting conflicts, based on different moments when to collect information, entail somewhat complementary behaviours of the solver. This opens some perspectives for building a still more robust solver by combining these conflict-based heuristics in a clever way. Identifying features or properties of problem instances (e.g., tree width, backbone size, presence of strong communities, structure of variable arrays, etc.) which are favorable for a certain form of conflict processing is an issue which would deserve to be addressed.

Table 4 Details per instance. For each selected instance and each heuristic, the best bound, the time (in seconds) to find it (TB) and the time (in seconds) to prove the optimum (TO, if proven) are provided. Best results are highlighted. DC-A, DC-B, DC-C and DC-D stands for DC-midori-xor4-d1-t0-r05-v128, DC-midori-xor4-d1-t0-r08-v128, DC-skinny-xor0-d0-t0-r09-v64-z0, and DC-midori-xor4-d1-t0-r09-v128, respectively. Some other names have been shortened for more visibility.

	cacd			frba			pick$_3$		
Instance	Bound	TB	TO	Bound	TB	TO	Bound	TB	TO
DC-A	**5**	733	739	**5**	9	**23**	5	21	28
DC-B	41	59	-	**8**	174	-	**8**	316	**484**
DC-C	**41**	5	-	**41**	9	-	**41**	3	**1,087**
DC-D	53	753	-	**9**	699	-	43	179	-
Fapp-m2s-21-0500	180,499K	1,195	-	**63,832K**	870	**881**	177,187K	1,008	-
Fapp-m2s-02-0250	221,762K	1,191	-	**221,520K**	1,158	-	221,591K	1,197	-
Fapp-m2s-test03-0400	453,213K	1,199	-	**449,289K**	1,192	-	450,098K	1,200	-
QueenAttacking-09	**0**	94	**98**	1	119	-	**0**	439	442
QueenAttacking-11	2	245	-	-	-	-	**1**	708	**736**
QueenAttacking-13	**11**	**158**	-	-	-	-	-	-	-
StillLife-11-14	**81**	264	-	79	652	-	**81**	625	-
StillLife-wastage-12	**76**	12	**872**	**76**	23	-	**76**	549	-
StillLife-wastage-37	**681**	1,189	-	619	1,182	-	585	398	-
Auction-cnt-d100	829K	200	-	825K	1,099	-	**849K**	1,197	-
Auction-sum-d100	840K	69	-	824K	77	-	**854K**	102	-
Auction-sum-d500	**3,368K**	186	-	3,264K	224	-	3,344K	89	-
CVRP-A-n32-k5	1,095	15	-	1,006	257	-	**835**	492	-
CVRP-A-n36-k5	1,050	77	-	1,039	1,112	-	**892**	74	-
CVRP-A-n34-k5	-	-	-	915	57	-	**813**	385	-
NurseRostering-01	**607**	1	-	**607**	1	11	**607**	1	**10**
NurseRostering-02	1,024	5	-	928	62	-	**833**	383	-
NurseRostering-19	68,621	1,200	-	-	-	-	**60,805**	1,197	-
Rlfap-graph-05-opt	2,882	883	-	**221**	2	**10**	**221**	62	**70**
Rlfap-graph-06-opt	46,647	1,198	-	34,004	1,198	-	**8,882**	769	-
Rlfap-scen-06-opt	11,211	1,166	-	10,102	491	-	**3,389**	124	-
Triangular-10	**20**	89	-	**20**	4	**138**	**20**	0	142
Triangular-22	50	46	-	50	161	-	**52**	167	-
Triangular-38	95	10	-	**97**	224	-	**97**	337	-
SteelMillSlab-m2s-3-0	68	1,059	-	-	-	-	**43**	995	-
SteelMillSlab-m2s-3-2	306	794	-	-	-	-	**171**	1,177	-
SteelMillSlab-m2s-4-0	161	1,109	-	-	-	-	**94**	1,186	-

References

1 K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
2 G. Audemard, C. Lecoutre, and E. Lonca. Proceedings of the 2022 XCSP3 competition. *CoRR*, abs/2209.00917, 2022. doi:10.48550/arXiv.2209.00917.
3 F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
4 F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016. URL: http://arxiv.org/abs/1611.03398, arXiv:1611.03398.
5 R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
6 R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1988.
7 E. Demirovic, G. Chu, and P. Stuckey. Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers. In *Proceedings of CP'18*, pages 99–108, 2018.
8 J.-G. Fages and C. Prud'homme. Making the first solution good! In *Proceedings of ICTAI'17*, pages 1073–1077, 2017.
9 S. Gagnon and G. Pesant. Accelerating counting-based search. In *Proceedings of CPAIOR'18*, pages 245–253, 2018.
10 P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, pages 31–35, 1992.
11 C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
12 D. Habet and C. Terrioux. Conflict history based heuristic for constraint satisfaction problem solving. *Journal of Heuristics*, 27(6):951–990, 2021.
13 R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
14 C. Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/Wiley, 2009.
15 C. Lecoutre. ACE, a generic constraint solver. *CoRR*, abs/2302.05405, 2023. doi:10.48550/arXiv.2302.05405.
16 C. Lecoutre and O. Roussel. Proceedings of the 2018 XCSP3 competition. *CoRR*, abs/1901.01830, 2019. arXiv:1901.01830.
17 C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 1:147–167, 2007.
18 C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Reasonning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.
19 C. Lecoutre and N. Szczepanski. PyCSP3: modeling combinatorial constrained problems in Python. *CoRR*, abs/2009.00326, 2020. arXiv:2009.00326.
20 H. Li, M. Yin, and Z. Li. Failure based variable ordering heuristics for solving CSPs. In *Proceedings of CP'21*, 2021.
21 J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
22 L. Michel and P. Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Proceedings of CPAIOR'12*, pages 228–243, 2012.
23 G. Pesant. From support propagation to belief propagation in constraint programming. *Journal of Artificial Intelligence Research*, 66:123–150, 2019.
24 G. Pesant, C.-G. Quimper, and A. Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43:173–210, 2012.
25 P. Refalo. Impact-based search strategies for constraint programming. In *Proceedings of CP'04*, pages 557–571, 2004.
26 F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

27 J. Vion and S. Piechowiak. Une simple heuristique pour rapprocher DFS et LNS pour les COP. In *Proceedings of JFPC'17*, pages 39–45, 2017.

28 H. Wattez, C. Lecoutre, A. Paparrizou, and S. Tabary. Refining constraint weighting. In *Proceedings of ICTAI'19*, pages 71–77, 2019.

29 N.-F. Zhou. An XCSP3 Solver in Picat. In *XCSP3 Competition 2022 Proceedings*, 2022.