Aggressive Bound Descent for Constraint Optimization

Thibault Falque
Exakis Nelite
CRIL, Univ Artois & CNRS

Christophe Lecoutre
CRIL, Univ Artois & CNRS

Bertrand Mazure
CRIL, Univ Artois & CNRS

Hugues Wattez
LIX CNRS, École Polytechnique, Institut Polytechnique de Paris

Abstract

Backtrack search is a classical complete approach for exploring the search space of a constraint optimization problem. Each time a new solution is found during search, its associated bound is used to constrain more the problem, and so the remaining search. An extreme (bad) scenario is when solutions are found in sequence with very small differences between successive bounds. In this paper, we propose an aggressive bound descent (ABD) approach to remedy this problem: new bounds are modified exponentially as long as the searching algorithm is successful. We show that this approach can render the solver more robust, especially at the beginning of search. Our experiments confirm this behavior for constraint optimization.

2012 ACM Subject Classification

Keywords and phrases Constraint Optimization, Backtracking Search

Digital Object Identifier 10.4230/LIPIcs.DPCP.2022.

1 Introduction

In Constraint Programming (CP), even if many related frameworks have been proposed since the 70's, it is usual to deal with either Constraint Satisfaction Problems (CSPs) or Constraint Optimization Problems (COPs). In addition to a set of (integer) variables to be assigned while satisfying a set of constraints, a COP instance involves an objective function to be optimized (i.e., a cost function to be minimized or a reward function to be maximized). Solving a COP instance not only requires to prove satisfiability (i.e., to find at least a solution), but also ideally to prove optimality or at least to find solutions of rather good quality (i.e., close to optimality).

Backtrack search is a classical complete approach for exploring the search space of a COP instance. Actually, this is equivalent to solve a series of CSP instances. Assuming an objective function f to be minimized, initially f is handled under the form of a constraint $f < \infty$, and each time a new solution of cost B is found, B is used as a new limit for the objective constraint, so as to become $f < B$ (hence, forming a tighter CSP instance). This way, any new found solution is guaranteed to have a better quality (lower cost) than the previous one, and optimality can be proved when the problem instance becomes unsatisfiable.

It is not unusual that optimization problems that come from industry are under-constrained, meaning that a lot of solutions exist with various qualities in different parts of the search space. In such situations, applying backtrack search may be penalized because the bound descent (i.e., the decreasing sequence of successively found bounds), can be very slow: the distance between any two successive bounds can be rather small. In this paper, we

propose an approach to attempt reducing this phenomenon by modifying the bounds in a more aggressive manner: as long as it is successful, new limits for the objective constraint are computed following an exponential growth. Of course, in case the limit renders the problem unsatisfiable or very difficult to solve, a restart mechanism allows us to resume search on safe tracks.

Even if other search techniques exist in the literature, like for instance the widely used meta-heuristic Large Neighborhood Search (LNS) [10], in this paper, we focus on depth-first backtracking search, equipped with solution (-based phase) saving that has been shown to be highly effective [12, 4].

This paper is organized as follows. In Section 2, technical background about CP is introduced. Next, in Section 3, Aggressive Bound Descent (ABD) is presented, and before concluding, experimental results are given in Section 4.

2 Technical Background

A *Constraint Network* (CN) consists of a finite set of variables and a finite set of constraints. Each variable x can take a value from a finite set called the *domain* of x, denoted by $\texttt{dom}(x)$. Each constraint c is specified by a relation over a set of variables. A *solution* of a CN is the assignment of a value to all variables such that all constraints are satisfied. A CN is *satisfiable* if it admits at least one solution, and the corresponding *Constraint Satisfaction Problem* (CSP) is to determine whether a given CN is satisfiable, or not.

A *Constraint Network under Optimization* (CNO) is a constraint network together with an objective function $\texttt{obj}$ that maps any solution to a value[1] in $\mathbb{R}$. Without any loss of generality, we shall consider that $\texttt{obj}$ must be minimized. A solution S of a CNO is a solution of the underlying CN; S is *optimal* if there is no other solution S' such that $\texttt{obj}(S') < \texttt{obj}(S)$. The usual task of the *Constraint Optimization Problem* (COP) is to find an optimal solution to a given CNO. Note that CNs and CNOs are also called CSP/COP instances.

Backtrack search is a classical complete procedure for solving CSP/COP instances. It interleaves variable assignments (and refutations) and a mechanism called constraint propagation in order to filter the search space. Typically, as in MAC [9] that propagates constraints by maintaining the property of arc consistency, a binary search tree $\mathcal{T}$ is built: at each internal node of $\mathcal{T}$, (i) a pair (x, v) is selected where x is an unfixed variable and v is a value in $\texttt{dom}(x)$, and (ii) two cases (branches) are considered, corresponding to the assignment $x = v$ and the refutation $x \neq v$. The order in which variables are chosen during the depth-first traversal of the search space is decided by a *variable ordering heuristic*; a classical generic heuristic being $\texttt{dom/wdeg}$ [2]. The order in which values are chosen when assigning variables is decided by a *value ordering heuristic*; for COPs, it is highly recommended to use first the value present in the last found solution, which is a technique known as *solution (-based phase) saving* [12, 4].

Backtrack search for COP relies on CSP solving: the principle is to add 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. It means that any time a solution S is found with cost $B = \texttt{obj}(S)$, the objective constraint becomes $\texttt{obj} < B$. Hence, a sequence of better and better solutions is generated (SATisfiability is systematically proved with respect to the current limit of the objective constraint) until no more exists

[1] For simplicity of presentation, we consider that costs are given by integer values.

(UNSATisfiability is eventually proved with respect to the limit imposed by the last found solution), guaranteeing that the last found solution is optimal.

Restart policies play an important role in modern constraint solvers as they permit to address the heavy-tailed runtime distributions of SAT (Satisfiability Testing) and CSP/COP instances [5]. In essence, a restart policy corresponds to a function $\texttt{restart} : \mathbb{N}^+ \rightarrow \mathbb{N}^+$, that indicates the maximal number of "steps" allowed for the search algorithm at attempt, called *run*. It means that backtrack search piloted by a restart policy builds a sequence of binary search trees $\langle \mathcal{T}_1, \mathcal{T}_2, \ldots \rangle$, where $\mathcal{T}_j$ is the search tree explored at run j. Note that the *cutoff*, which is the maximum number of allowed steps during a run, may correspond to the number of backtracks, the number of wrong decisions [1], or any other relevant measure. In a *fixed* cutoff restart strategy, $\texttt{restart}(j)$ is constant whatever is the run j. In a *dynamic* cutoff restart strategy, `restart` increases the cutoff geometrically [13], which guarantees that the whole space of partial solutions will be explored.

3 Aggressive Bound Descent

When solving a COP instance, the *bound descent* is defined as the sequence $D = \langle B_1, B_2, \ldots \rangle$ of successive bounds identified by the search algorithm. At an extreme, this sequence contains only one value, the optimal bound. At another extreme, it contains a large sequence of values, each one being close to the previous one: the bound descent is said to be *slow*. This is the case when the mean value of the derived sequence of bound gains (or gaps) $G = \langle B_1 - B_2, B_2 - B_3, \ldots \rangle$ is small (close to 1).

Certainly, a slow bound descent indicates that there is some room for improvement about the way the backtrack search is conducted. Indeed, enumerating a lot of close solutions before reaching optimality involves solving many derived satisfaction problem instances, always on different, although related, backgrounds (objective constraint limits), and this may be penalizing. This is why we propose an *aggressive* policy of bound descent, ABD (policy) in short. Instead of setting the strict objective constraint limit to B when a new solution of cost B is found, we propose to set it to a possibly lower value B'.

A first and simple ABD policy could be to use a static difference between B and B': $B' = B - \Delta$ where Δ is a fixed positive integer value. However, this *static* policy clearly suffers from a lack of adaptability, and besides, setting the right value for Δ may be problem-dependent and not very easy to achieve. This is why we propose some *dynamic* ABD policies, inspired from studies concerning the sequences used by restart policies.

To define dynamic ABD policies, we first introduce a few general sequences of strictly positive integers, i.e., functions $\texttt{abd} : \mathbb{N}^+ \rightarrow \mathbb{N}^+$. Although detailed later, the parameter $i \geq 1$ of these sequences corresponds to the number of successive successful limit updates, i.e., successive aggressive updates of the objective constraint limit while keeping satisfiability. Specifically, four integer sequences are used in our study:

$$\texttt{exp}(i) = 2^{i-1} \tag{1}$$

$$\texttt{rexp}(i) = \begin{cases} 2^{k-1}, & \text{if } i = \frac{k(k+1)}{2} \\ 2^{i-\frac{k(k+1)}{2}-1}, & \text{if } \frac{k(k+1)}{2} < i < \frac{(k+1)(k+2)}{2} \end{cases} \tag{2}$$

$$\texttt{luby}(i) = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \\ \texttt{luby}(i - 2^{k-1} + 1), & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases} \tag{3}$$

$$\texttt{prev}(i) = \begin{cases} 1, & \text{if } i = 1 \\ G_{i-1} \times 2, & \text{else} \end{cases} \tag{4}$$

where, in Equation 4, G_i is the ith value of the sequence of bound gains, as defined earlier.

Equation 1 corresponds to the classical exponential function `exp` (using base 2). Derived from this simple exponential progression, `rexp` in Equation 2 corresponds to a regularly reinitialized `exp` sequence. The first values of this sequence are: $1, 1, 2, 1, 2, 4, \ldots$ When only considering the highest numbers produced by the first term (condition) of the equation, we obtain a slightly slower progression than the previous one: $O(2^{\sqrt{i}})$. Another sequence, commonly used in restart policies, is the Luby sequence [8], given by Equation 3. The first values of the Luby sequence are: $1, 1, 2, 1, 1, 2, 4, \ldots$ When considering again the highest numbers produced by the first term, we can observe that the progression is in $O(i)$. Finally, the last sequence, given by Equation 4, is based on the sequence of bound gains, and also follows an exponential progression.

Each sequence in $\Psi = \{\texttt{exp}, \texttt{rexp}, \texttt{luby}, \texttt{prev}\}$ allows us to define an eponymous ABD policy as follows.

3.1 ABD Policy

Let $\mathcal{T}$ be the current search tree built by the search algorithm (i.e., during the current run). Let D be the bound descent produced since the beginning of the current run, and let $\texttt{abd} \in \Psi$. The current run can meet three distinct situations:

1. the current run is stopped because the cutoff value is reached,
2. the current run is stopped because the search algorithms indicates that no more solution exists,
3. a new solution S is found.

First, we discuss the most interesting case: the third one. The ABD policy states that when a new solution S of cost B is found, B is appended to D, and the limit of the objective constraint is set to $B + 1 - \texttt{abd}(i)$, where $i = |D|$. In other words, the objective constraint becomes: $\texttt{obj} < B + 1 - \texttt{abd}(i)$; note that 1 is added to B because `abd` functions only return values greater than or equal to 1. Now, we give a general precise description (handling in particular the two first situations above) of how an ABD policy can be implemented within backtrack search.

3.2 Simple ABD Implementation

The function `solve`, Algorithm 1, aims at solving the specified CNO P while using the specified aggressive bound descent policy `abd`.

Algorithm 1 $\texttt{solve}(P, \texttt{abd})$

Output: $\underline{B_P}..\overline{B_P}$, runStatus
1 $\underline{B_P}..\overline{B_P} \leftarrow -\infty..+\infty$
2 **do**
3 $\quad P$, runStatus $\leftarrow \texttt{run}(P, \texttt{abd})$
4 **while** runStatus = CONTINUE
5 **return** $(\underline{B_P}..\overline{B_P}$, runStatus)

First of all, the lower and upper bounds, denoted by $\underline{B_P}$ and $\overline{B_P}$, of the objective function of P are respectively initialized to $-\infty$ and $+\infty$ (or any relevant values that can be pre-computed). During search, these bounds will be updated (but for the sake of simplicity, this

will not be explicitly shown in the pseudo-code). At line 2, the sequence of runs (restarts) is launched. Each time a new run is terminated, it returns the constraint network (possibly updated with some constraints or nogoods that have been learned; this will be discussed in more details later) together with a status information. The status takes one of the following values: CONTINUE if the solver is allowed to continue with a new run; COMPLETE if the last run has exhaustively explored the solution space; INCOMPLETE if the solver has reached the timeout limit without entirely exploring the search space. Finally, the function returns the best found bounds (in case, the optimality has been proved, we have $\underline{B_P} = \overline{B_P}$) and the final status of the search.

The function `run`, Algorithm 2, performs a search run, following the restarts and `abd` policies. Before going further, we need to introduce the notion of *safe/unsafe* run solving: when the solver is asked to aggressively decrease its objective limit, we may enter a part of the search space that is UNSAT. If unsatisfiability is proved during the current run, this may be due to our aggressive approach, and consequently, we have to address this issue. This is discussed below.

Algorithm 2 `run`$(P, \texttt{abd})$

```
   Output: (P, status)
1  i ← 1
2  Σ_i ← ∅
3  do
4  |  Δ ← abd(i)
5  |  i ← i + 1
6  |  Σ_i, status ← search_next_sol(P, Σ_{i−1}, Δ)
7  while status = SAT
8  safe ← Δ = 1
9  if status = TIMEOUT then
10 |  return (P, INCOMPLETE)
11 if status = UNSAT & safe then
12 |  return (P, COMPLETE)
13 if status = UNSAT & ¬safe then
14 |  return (P ⊕ nld(Σ_{i−1}), CONTINUE)
15 if status = CUTOFF_REACHED & ¬safe then
16 |  return (P ⊕ nld(Σ_{i−1}), CONTINUE)
17 if status = CUTOFF_REACHED & safe then
18 |  return (P ⊕ nld(Σ_i), CONTINUE)
```

The function starts by initializing a counter i to 1. It corresponds to the number of times we have tried to find a new solution during the current run. At Line 2, Σ_i is the sequence of decisions taken along the rightmost branch of the current run, just before starting the next attempt to find a new solution; this way, we can keep searching from the very same place (in practice, we simply resume search after is was stopped). Initially, we start from no taken decisions at all (and so, Σ_1 is the empty set). From a practical point of view, as we shall see, only the two last sequences Σ_i and Σ_{i-1} will be useful (to deal with the safe and unsafe solving cases). Then, the algorithm iteratively performs runs as long as new solutions can be found. At each turn of the loop, the next bound gap Δ is computed by soliciting the `abd` policy, i is incremented (lines 4 and 5). To perform a part of the search, the function `search_next_sol` is called, while considering the specified sequence of decisions

to start from, and the specified bound gap. The gap Δ is used by `search_next_sol` to compute a temporarily upper bound B' which replaces the current upper bound $\overline{B_P}$: we have $B' = \overline{B_P} - \Delta + 1$, forcing then the objective constraint to be $f < B'$ during this call to `search_next_sol`. If a new solution of cost B (necessarily, $B < B'$) is found by `search_next_sol`, the call is stopped, and the objective constraint is updated to safely become $f < B$. Otherwise, the call is stopped (because the cutoff or timeout limits are reached), and the objective constraint is updated to $f < \overline{B_P}$ (getting back the previous safe upper bound). To summarize, this function implicitly updates the optimization bounds before returning the new sequence of decisions (the exact place where the search has stopped) and a (local) status. The local status is either SAT, in which case the run can be continued with a new loop iteration, or a value among UNSAT, CUTOFF_REACHED, and TIMEOUT.

The current run necessarily executes some statements starting from Line 8. At this line, a Boolean is set, informing us whether the current run was safe or not, regarding the last computed Δ value. When the global timeout limit is reached, the constraint network and the status INCOMPLETE are returned (lines 9 and 10). When the local status notifies UNSAT, we have two cases to consider. If the current search was performed in safe mode, COMPLETE can be returned because the search space is guaranteed to have been completely explored. Otherwise, CONTINUE is returned with the constraint network P possibly integrating some new constraints (nogoods). The notation $P \oplus nld(\Sigma_{i-1})$ indicates that all nogoods that can be extracted from the last but one sequence of decisions (see [6]) are added to P; this is valid because this sequence was the one corresponding to the last found solution. When the local status notifies CUTOFF_REACHED, we can also continue, while considering the adjunction of some restart nogoods, from either Σ_i or Σ_{i-1}.

We conclude this section with two remarks. Firstly, the algorithm is introduced within the context of a light nogood recording scheme (only, nogoods that can be extracted from the rightmost branch, when the search is temporarily stopped, are considered). However, it is possible to adapt it to other learning schemes, by keeping track of the exact moment where a nogood (clause) is inferred; this is purely technical. Secondly, there is a specific case concerning unsatisfiability: if ever we encounter a situation where $B' \leq \underline{B_P}$ when trying to set a new temporarily upper bound B' during the current run, the sequence is reinitialized by forcing back $i = 1$ and B' is recomputed.

3.3 Related Work

As a related approach, let us mention domain splitting (e.g., see [11]) whose role is to partition the domain of the variable selected by the variable-ordering heuristic, and to branch on the resulting sub-domains. When the objective function is simply represented by a stand-alone variable (whose value must be minimized or maximized), a domain splitting approach can be tuned to simulate an exponential aggressive bound descent. However, no control is possible as no mechanism allows us to abandon too optimistic choices, contrary to ABD which is well integrated within a restart policy. Besides, when the objective function has a more general form than a variable (like, for example a sum or a minimum/maximum value to be computed), there is no more direct correspondence (and systematically introducing an auxiliary variable for representing the objective can be very intrusive, and even source of inefficiency, for the solver). Finally, note that the issue of avoiding long sequences of slowly improving solutions has been addressed in MIP by introducing primal heuristics, which aim at finding and improving feasible solutions early in the solution process.

4 Experimental Results

This section presents some experimental results concerning the ABD approach on a wide range of optimization problems. To conduct the experiments, we have used the constraint solver `ACE`, which is the new avatar of AbsCon[2]. The default option settings of the solver were used: $\texttt{wdeg}^{\texttt{ca.cd}}$ [15] as variable-ordering heuristic, `lexico` as value-ordering heuristic, last-conflict (lc) [6] as a lazy manner to simulate intelligent backtracking, `solution-saving` [12, 4] for simulating a form of neighborhood search, and a geometric restart policy [14] with a base run cutoff set to 10 wrong decisions and a common ratio fixed to 1.1. We notably study the impact of certain factors of the most promising sequences. All experiments have been launched on 2.66 GHz Intel Xeon CPU, with 32 GB RAM, while the timeout was set to $1,200$ seconds.

4.1 About Scoring

First, we introduce the scoring method we used for assessing the experimental results. For the sake of simplicity, we continue to consider having only minimization problems.

Given a set $\mathcal{I}$ of instances and a set $\mathcal{S}$ of solvers, $b^t_{i,s}$ corresponds to the best bound (i.e., the lowest one) obtained by the solver $s \in \mathcal{S}$ on instance $i \in \mathcal{I}$ at time $t \in [0, ..., T]$ and T is the timeout. We consider the *default solver* $\texttt{def} \in \mathcal{S}$ to be the the original solver using its default behavior (and so, using no ABD policy). For example, $b^t_{i,\texttt{def}}$ is simply the bound obtained by the default solver on instance i at time t.

We also have a Boolean $e^t_{i,\mathcal{S}}$ whose value is *true* when a solution has been found at time t by at least one solver of $\mathcal{S}$. We can now define two specific values:

$$\texttt{min}^t_i = \begin{cases} \min_{s \in S} b^t_{i,s}, & \text{if } e^t_{i,\mathcal{S}} \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

$$\texttt{max}^t_i = \begin{cases} \max_{s \in S} b^t_{i,s}, & \text{if } e^t_{i,\mathcal{S}} \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

where $b^t_{i,s}$ is equal to $\texttt{min}^t_i$ if $e^t_{i,\{s\}}$ is false.

The two previous expressions respectively correspond to the smallest (best) and highest (worst) bounds obtained by a solver on a given instance i at time t.

Then, we can compute a reward for a pair (i, s):

$$r^t_{i,s} = \begin{cases} 0, & \text{if } \neg e^t_{i,\{s\}} \\ 1 - \frac{b^t_{i,s} - \texttt{min}^t_i}{\texttt{max}^t_i - \texttt{min}^t_i}, & \text{if } \texttt{max}^t_i \neq \texttt{min}^t_i \\ 1, & \text{otherwise} \end{cases} \tag{7}$$

The reward function (Equation 7) corresponds to a classical *min-max normalization* with possible missing values. In case a solver has not found any solution, its reward is 0, and in case the smallest and highest bounds are equal (meaning that the solver has found the

[2] `https://github.com/xcsp3team/ace`

unique known solution at time t), its reward is 1. Otherwise, *min-max* is employed by taking the complement to 1 of the result (because the *min-value* is the best).

The mean reward (based on r) of each solver s at time t is defined as follows:

$$R_s^t = \frac{1}{|\mathcal{I}|} \times \sum_{i \in \mathcal{I}} r_{i,s}^t - r_{i,\texttt{def}}^t \tag{8}$$

This last equation will be useful to draw plot-lines (e.g., Figure 1) representing the average progression over time of any solver compared to the default solver. Because of the way it is defined, the default solver corresponds to $y = 0$. Hence, solvers with a curve situated above $y = 0$ can be considered as being better than the default solver, contrary to those with a curve below $y = 0$.

4.2 Constraint Optimization

Our approach has been evaluated on a wide range of optimization problems coming from the XCSP distribution [3, 7]. We have used a benchmark corresponding to the entire set of COP instances from the XCSP18 competition, resulting in 22 problems and 362 instances.

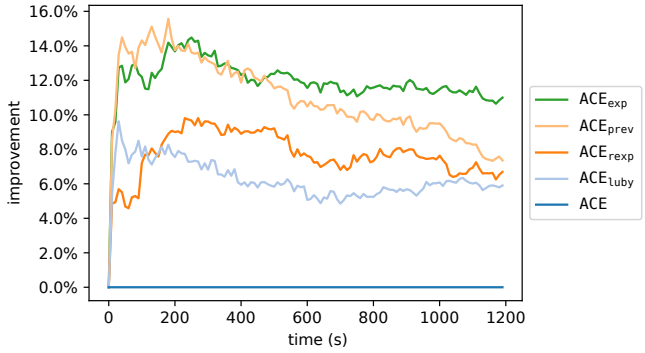

Figure 1 Comparison of ABD Policies from Ψ on the optimization solver `ACE`

Figure 1 shows, against time, a direct comparison of ABD policies with respect to the default solver (remember that default `ACE` is represented by $y = 0$). We can first observe a neat interest in using ABD Policies during the 200 first seconds: an improvement from 8% (for the worst policies) to 14-16% (for $\texttt{ACE}_{\texttt{exp}}$ and $\texttt{ACE}_{\texttt{prev}}$). We can also see that $\texttt{ACE}_{\texttt{exp}}$ stays at (an improvement level of) around 12%, while $\texttt{ACE}_{\texttt{prev}}$ decreases to 8%.

5 Conclusion

In this paper, we have introduced ABD (Aggressive Bound Descent) which is a technique modifying aggressively the limit of objective constraints. By taking risks of running periodically solvers in unsafe parts of the search space, we show that interesting experimental performance can be obtained on constraint optimization problems. It allows us to get more quickly better bounds than the default backtrack search approach on some strongly structured problems. Nevertheless, we believe that some refinements of ABD could be studied, as for example, exploiting more the history of bound gains, or identifying the relevant sequences of limit gaps to be used according to the structure of the problems.

References

1 C. Bessiere, B. Zanuttini, and C. Fernandez. Measuring search trees. In *Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints*, pages 31–40, 2004.

2 F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.

3 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`.

4 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.

5 C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1):67–100, 2000.

6 C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Reasonning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

7 C. Lecoutre and N. Szczepanski. PyCSP^3: Modeling combinatorial constrained problems in Python. Technical report, CRIL, 2020. Available from `https://github.com/xcsp3team/pycsp3`.

8 M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

9 D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.

10 P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of CP'98*, pages 417–431, 1998.

11 W.J. van Hoeve and M. Milano. Postponing branching decisions. In *Proceedings of ECAI'04*, pages 1105–1106, 2004.

12 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.

13 T. Walsh. Search in a small world. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, page 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

14 T. Walsh. Search in a small world. In *Proceedings of IJCAI'99*, pages 1172–1177, 1999.

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