Black-Box Value Heuristics for Solving Optimization Problems with Constraint Programming

Augustin Delecluse
TRAIL, ICTEAM, UCLouvain, Belgium

Pierre Schaus
ICTEAM, UCLouvain, Belgium

Abstract

Significant research efforts have focused on black-box variable selection, with less attention given to value heuristics. An ideal value heuristic enables depth-first-search to prioritize high-quality solutions first. The Bound-Impact Value Selection achieves this goal through a look-ahead strategy, trying every value of the selected variable and ranking them based on their impact on the objective. However, this method is generally too computationally intensive for the entire search tree. We introduce two simple yet powerful modifications to improve its scalability. First, a lighter fix point computation involving only the constraints on the shortest path in the constraint graph between the variable and the objective. Second, a reverse look-ahead strategy optimistically fixes the objective variable to its minimum in order to prioritize the remaining values. These two ideas have been empirically validated on a range of academic problems and in the XCSP^3 competition, demonstrating significant improvements in scalability.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Programming, Value Selection, Look-Ahead, Optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2024.8

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/augustindelecluse/choco-solver`

Funding *Augustin Delecluse*: This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.A

1 Introduction

The Constraint Programming (CP) paradigm enables to solve combinatorial problems in a declarative way. The Holy Grail vision of CP is that the user simply has to state the problem [5]. However, for practical efficiency, this vision has been adjusted to the long-standing CP mantra: *CP = Model + Search*. The capability to program problem-specific searches remains crucial in CP to minimize the search tree size. Over time, the emphasis on search programming has diminished due to the development of effective black-box search strategies by the CP community [7, 4, 13, 20, 27]. A common search procedure involves a two-step decision-making process at each node: selecting an undecided variable and then choosing a value to assign from the domain on the left branch, which is removed upon backtracking. Despite extensive work on deriving efficient problem-independent variable selection heuristics based on the first-fail principle, few black-box strategies have been proposed for value selection. In the context of optimization problems, a simple yet effective generic method is the Bound-Impact Value Selector (BIVS) [12]. This look-ahead heuristic iterates over each value in the domain of a variable, successively assigning the variable to each of these values. After each

assignment is made, the fixpoint filtering by the constraints is computed. The impact on the objective—specifically, the lower bound increase in the case of a minimization problem—is then measured. The value that yields the to the smallest increase for minimization problem is selected for the left assignment.[1]

Although BIVS finds solutions of better quality compared to other naive strategies, such as picking the minimum value in the domain of a variable, its computational cost does not always compensate its benefits in guiding the search toward promising directions. This is especially true for problems with large domain sizes or when a large number of constraints are involved in the fixpoint computation. To overcome this shortcoming, one tradeoff is to consider only the smallest and largest values from the domain, instead of every value, especially if the domain size is too large. This is the default value selection performed in Choco-solver [26].

Our contribution introduces two ideas to reduce the computational cost of BIVS. The first, called the *Restricted Fixpoint* is to compute the fixpoint on a restricted set of constraints. Specifically, we measure the impact only on a fixpoint that involves the constraints along the shortests paths from the selected variable to the objective variable in a constraint graph, where node constraints are connected to node variables in their scope.

The second improvement, called *Reverse Look-Ahead* starts from the objective assignment. Instead of sequentially fixing the selected variable to each value, we suggest optimistically setting the objective variable to its lower bound (assuming minimization) and then computing the fixpoint. The selected value is one of those that remain. If infeasibility is detected, we increase the objective bound and continue until no domain is empty.

The two proposed improvements are designed to be non-intrusive, meaning that they can be implemented without necessitating modifications to the solver's architecture.

The paper is organized as follows. Section 2 presents first the main concepts necessary to understand BIVS, before introducing the two ideas to lower the cost of BIVS, namely the Restricted Fixpoint and Reverse Look-Ahead. Their performances are assessed in section 3.

2 Reducing BIVS Cost

The search heuristic is generally decomposed in two stages for efficiency reason: the variable is selected, then the value in the domain of the variable. For the variable selection, many ideas inspired by the *first-fail* principle "to succeed, try first where you are most likely to fail" [16] have been proposed such as choosing the variable with the smallest ratio of size of domain versus number of failures when running the fixpoint on them (`dom/wdeg`) [7, 4], the variable involved in the most recent failures [13, 20, 21], the variable whose assignment has the largest estimated search space reduction [27], the variable with the most frequent domain changes [24, 4] etc.

Once the variable to branch on has been selected, a value must be assigned to it. In opposition to the *first-fail* principle, the *succeed-first* principle is used instead [14, 10]. The idea is to select the value being most likely to participate in a solution. Compared to variable selection, only a few generic selectors have been proposed for picking a value, which is why choosing the smallest value from the domain of a variable remains a popular default choice [19, 29]. Two exceptions are Belief Propagation (BP) [25], aimed at driving the search towards areas with a large number of solutions, or Bound-Impact Value Search (BIVS) [12], choosing the value with the best impact on the objective. Even though those two methods

[1] This idea is similar to *Strong Branching* in Mixed-Integer Linear Programming solvers [2].

do reduce a lot of the search space, their computational cost is too large to make them the default choice in most solver during the whole search procedure. Moreover, BP is better suited for constraint satisfaction problems rather than optimization problems. Research on machine learning for value selection heuristics [22, 9] is limited by extensive data requirements for lengthy training phases. While Objective Landscapes methods [18] collecting pre-search impacts work well in stable settings like scheduling, they falter in dynamic scenarios where impacts fluctuate with other assignments. An effective alternative is *phase-saving*, selecting values from previous solutions [11], though it relies on a default heuristic when no prior value is applicable.

Our work revisits the BIVS algorithm, detailed in Algorithm 1, to reduce its computational cost while maintaining its effectiveness in finding high-quality solutions. BIVS examines each value v in the domain of the selected variable $x \in \mathcal{X}$ (line 3), assigning v to x (line 5) and running the fixpoint algorithm (line 6). If the fixpoint is error-free and improves the objective's lower bound ($\lfloor \mathcal{D}(obj) \rfloor$), the value is considered for retention (line 7). The process is encapsulated within `saveState`($\mathcal{X}$, $\mathcal{C}$) and `restoreState`($\mathcal{X}$, $\mathcal{C}$) operations (lines 4 and 10), ensuring that the solver's state is reset before each new trial[2]. While this method is effective in steering the search towards high-quality solutions, the time complexity for selecting a value for variable x in the BIVS algorithm is $\Theta(\mathcal{F} \cdot |\mathcal{D}(x)|)$, with $\mathcal{F}$ as the fixpoint algorithm's complexity. As [4] notes, controlling this high computation cost is challenging. We propose two methods to reduce this complexity: one by lowering the cost of the fixpoint algorithm and the other by reducing the frequency of its calls. However, these changes do not guarantee identical outcomes as the original BIVS.

Algorithm 1 Bound-Impact Value Selector (assuming minimization), adapted from [12].

Input : $\mathcal{X}$: variables, $\mathcal{C}$: constraints, x: branching variable, obj: objective variable
Ouput : bestV, the value to assign to the variable x.

```
1  bestV ← ⌊D(x)⌋
2  bestBound ← ∞
3  for v ∈ D(x) do
4  |  saveState(X, C)
5  |  D(x) ← {v}
6  |  success ← fixpoint(X, C)
7  |  if success and ⌊D(obj)⌋ < bestBound then
8  |  |  bestBound ← ⌊D(obj)⌋
9  |  |  bestV ← v
10 |  restoreState(X, C)
11 return bestV
```

2.1 Restricted Fixpoint Computation

The Restricted Fixpoint (RF) approach assesses the impact on the objective by focusing solely on a limited set of constraints. Specifically, it considers only the constraints that are located on the shortest paths from the selected variable to the objective variable within a bipartite variables-constraints graph. In this graph, variables and constraints are nodes,

[2] See [23] for state saving and restoration with a trail.

connected by edges if the variable is within the scope of the constraint. For each variable, we precompute all constraints on these paths. This method reduces computation costs as it involves fewer constraints, but it may be less informative and risk missing potential failures.

▶ **Example 1.** Consider the Traveling Salesman Problem (TSP), where a salesman needs to visit a set of cities, each city being visited exactly once, while minimizing the traveled distance. Given n cities and a distance matrix $d \in \mathbb{Z}^{n \times n}$, a commonly used CP model is minimize $TotD$ such that $\texttt{Circuit}(S)$, $D_i = \texttt{Element}(d_{i,*}, S_i)\ \forall i \in \{0..n-1\}$, and $TotD = \texttt{Sum}(D)$. It introduces two variables per city: S_i is the visit occurring after the city i in the tour, and D_i the distance between city i and its successor S_i. The circuit constraint enforces every city to be visited within a single tour. The distance D_i between a city i and its successor S_i corresponds to the S_i-th entry within the line i in the distance matrix d, enforced with an element constraint. The sum of traveled distance $TotD$ is the objective to minimize. A TSP with 4 cities is shown in Figure 1 and the graph in Figure 3. When selecting values for S_0 using RF with BIVS, only 2 constraints are considered per iteration, compared to 6.

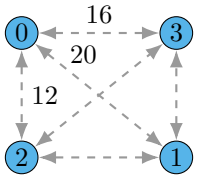

Figure 1 A TSP instance with 4 cities to visit. The distances are shown on the edges.

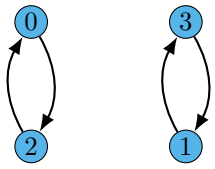

Figure 2 When fixing $\mathcal{D}(TotD)$ to $\{48\}$, all successors points towards their nearest city. This violates the Circuit constraint.

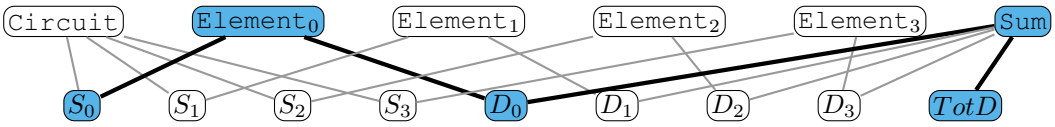

Figure 3 Constraints and variables of the TSP instance from Figure 1. Variables and constraints in blue are located on the shortest path to the objective $TotD$ when considering variable S_0.

2.2 Reverse Look-Ahead

The Reverse Look-Ahead (RLA) strategy reduces the number of calls to the fixpoint computation by restricting optimistically the domain of the objective and observing the effects on the variable to $\mathcal{D}(x)$, rather than fixing x directly. It is similar to the Destructive Lower Bound used in scheduling [17] and can also tighten bounds on the objective.

Algorithm 2 outlines RLA. A value δ controls the domain size of the objective variable during the fixpoint computation, starting with a value of 1 to fix the objective to its minimum value. Several iterations may be performed, each increasing δ until the fixpoint computation succeeds. At this point, the minimum value from the domain of x is returned at line 11. The value of δ doubles at each iteration, resulting in an exponential evolution. Note that when the fixpoint computation fails, the lower bound of the domain of the objective variable can be safely increased (line 14).

Algorithm 2 Reverse Look-Ahead (assuming minimization)

Input : $\mathcal{X}$: variables, $\mathcal{C}$: constraints, x: branching variable, obj: objective variable
Ouput : *success*: boolean indicating node expandability, v: assigned value for x

```
1  δ ← 1
2  success ← true
3  while success do
4  |  saveState(X, C)
5  |  ⌈D(obj)⌉ ← min(⌈D(obj)⌉, ⌊D(obj)⌋ − 1 + δ)
6  |  success ← |D(obj)| > 0
7  |  if success then
8  |  |  if fixpoint(X, C) then
9  |  |  |  v ← ⌊D(x)⌋
10 |  |  |  restoreState(X, C)
11 |  |  |  return (true, v)
12 |  |  else
13 |  |  |  restoreState(X, C)
14 |  |  |  ⌊D(obj)⌋ ← ⌊D(obj)⌋ + δ
15 |  |  |  success ← |D(obj)| > 0
16 |  |  |  δ ← δ * 2
17 |  else
18 |  |  restoreState(X, C)
19 return (false, 0)
```

▶ **Example 2.** Consider the same situation as Example 1, shown in Figure 1. The initial fixpoint yields $\mathcal{D}(D_i) = \{12, 16, 20\}\ \forall i \in \{0..n-1\}$ and $\mathcal{D}(TotD) = \{48, \ldots, 60\}$. When using RLA to choose a value for S_0, the following iterations occur:

1. The fixpoint is triggered with $\mathcal{D}(TotD) = \{48\}$. This means that the successor of every city must be the closest city, violating the `Circuit` constraint (cf Figure 2). The iteration fails and the lower bound of the objective is now set to 49 for the sub-tree to consider.
2. $\delta = 2$ and $\mathcal{D}(TotD) = \{49, 50\}$. Similarly, this fails and sets the lower bound to 51.
3. $\delta = 4$ and $\mathcal{D}(TotD) = \{51, \ldots, 54\}$. The fixpoint proceeds without failure, resulting in $\mathcal{D}(S_0) = \{2, 3\}$. Value 2, the nearest neighbor, is picked and the state is restored while keeping the lower bound of the objective to 51. Finally, 2 is returned (lines 9 to 11).

The time complexity of RLA is $\Omega(\mathcal{F})$ in the best case, if only one iteration needs to be performed, and $\mathcal{O}(\mathcal{F} \cdot \log_2 |\mathcal{D}(obj)|)$ in the worst case. Moreover, RF can also be used with RLA, meaning that the complexity of the fixpoint can be lowered.

▶ **Example 3.** We reuse the model from Example 2. Initially, RLA restricts $\mathcal{D}(TotD) = \{48\}$ and runs the RF. This scenario suggests all successors should be nearest neighbors, which is infeasible given the `Circuit` constraint (see Figure 2). However, when considering only the shortest path constraints (the `Sum` constraint and one `Element` constraint, cf Figure 3), the failure is missed. Consequently, the domain of S_0 becomes $\{2\}$ (its closest neighbor), and 2 is returned. Compared to the scenario in Example 2, only one iteration has been performed (being less costly) but the heuristic itself could not tighten the bounds of the objective.

3 Experiments

To assess the performances of our methods, we consider two main settings. The first one analyzes three classical discrete problems easily modeled with CP. The second one reports the performances on various optimization problems, using the XCSP^3 2023 competition [8, 3].

All experiments were conducted using two Intel(R) Xeon(R) CPU E5-2687W. The implementation was done in Java in the Choco-Solver (version 4.10.5), a state-of-the-art general purpose constraint programming solver [26]. In all settings, instances needing more than 32GB were discarded. The variable selection used is DomWDeg [7] combined with last conflict [20], a popular default selection. The timeout was set to 30 minutes.

3.1 Fundamental Problems

Three fundamental problems are studied. (i) The TSP and the instances from TSPLib [28]. (ii) The JobShop and the instances from [30]. (iii) The Quadratic Assignment Problem and the instances from [1]. For each model, the standard models are used. The JobShop model branches on precedences like in [15]. For each model a custom white-box value heuristic is used called Greedy in the results. In total, the value selectors analyzed are:

Min choosing the smallest value in the domain of the variable.

BIVS the original algorithm as proposed in [12], using the author's implementation in Choco-solver. In this implementation, when the domain size of a variable is larger than 100, only the minimum and maximum values of the domain are considered by BIVS.

BIVS+RF indicates BIVS but with the restricted fixpoint, presented in section 2.1.

RLA depicts the Reverse Look-Ahead.

RLA+RF depicts the Reverse Look-Ahead using the restricted fixpoint.

One criterion used to compare the value selection heuristics is the primal gap introduced in [6]. It gives a value between 0 and 1 measuring the gap between the value of a solution found obj and the best found solution obj_{opt}. A value close to 0 means that the solution found is the best one found, while a value of 1 indicates that no solution was found.

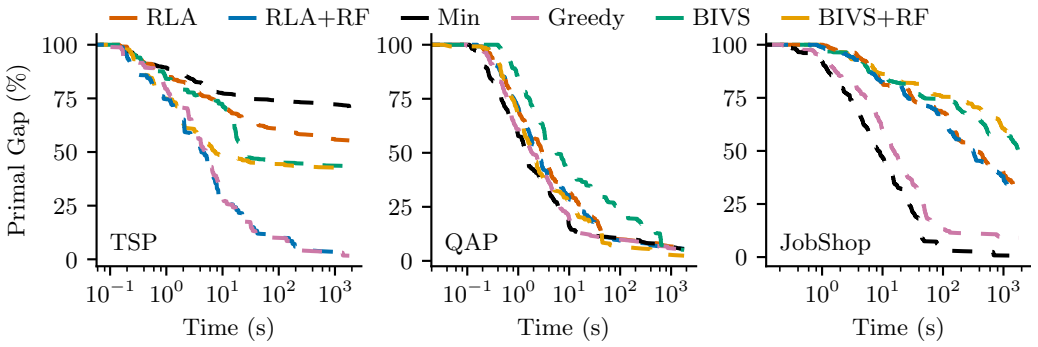

Figure 4 Primal gap in percentage over time.

Heuristic performance varies by problem (Figure 4). For TSP, RLA+RF matches the greedy heuristic, while BIVS+RF lags slightly due to considering bounds for domain sizes over 100; without this restriction, BIVS+RF performs comparably to RLA+RF. In QAP, BIVS and its RF variant outperform others, with RF significantly speeding up BIVS. For JobShop, RLA surpasses BIVS but is less effective than MIN due to the cost of fixpoint calls.

The addition of RF to BIVS results in speedups for TSP and QAP, maintaining average solution quality across instance sizes, as shown in Figure 5 and supported by Figure 4

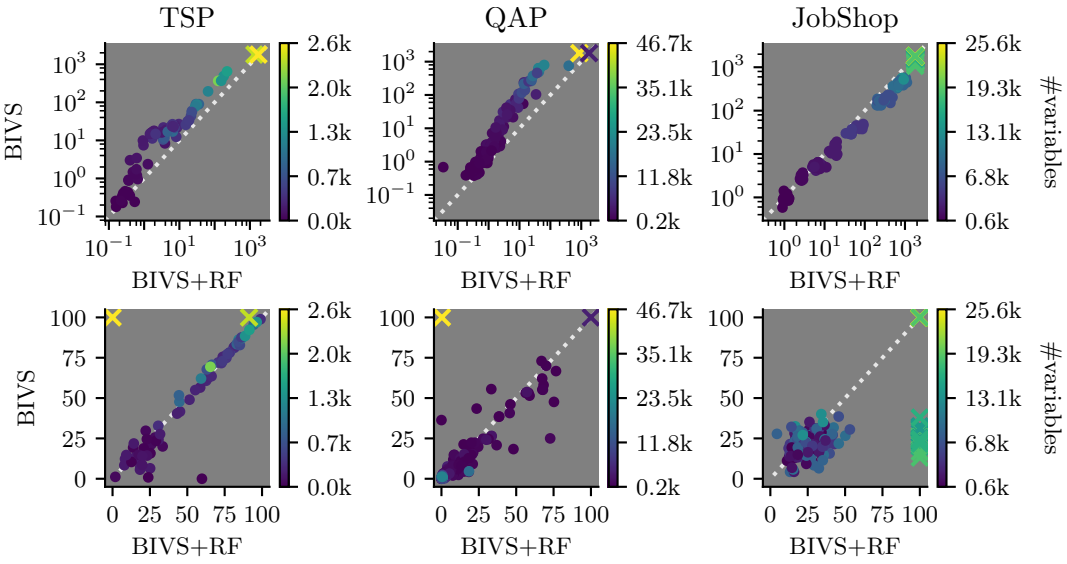

Figure 5 Comparison of BIVS and BIVS+RF regarding the time to find the first feasible solution (top, in seconds) and its corresponding gap (bottom, in percentage). Each dot represents an instance across problems: TSP (left), QAP (center), and JobShop (right). Dots on the diagonal indicate equal performance between methods. Dots above the diagonal show that BIVS was slower or found poorer solutions. Crosses denote timeouts by at least one method, resulting in a 100% gap.

where BIVS falls behind BIVS+RF. Conversely, RF slows performance on JobShop. On this problem, the cost of scanning all constraints and potentially deactivating some, to save time on the two iterations performed by BIVS (as the precedence variables have a domain of size 2) does not offer benefits compared to considering all constraints.

3.2 XCSP^3

We consider instances from the XCSP^3 COP 2023 competition [8, 3]. The instances requiring more than 32GB were discarded, leaving 18 problems and 232 instances.

Table 1 displays gap and number of solutions found per problem, with the average gap over time shown in Figure 6. Performance varies greatly across problems, with no single heuristic outperforming others universally. However, adding RF to BIVS reduces the average gap and aids in finding solutions missed otherwise. Similar benefits, though smaller, are observed with RLA. On average, RLA outperforms BIVS, with RF enhancing both methods. Min excels in finding feasible solutions, mostly attributed to its constant time complexity, allowing for better learning from variable selection.

Moreover, Figure 6 presents two Virtual Best Solver entries: "VBS (old)" computed from previous selectors available in Choco (BIVS, Min, middle, Max and random domain selection) and "VBS (new)" adding RLA and RF-based methods. The 3.87% increase in the final gap demonstrates that RLA and RF explore the search space differently than traditional heuristics, enhancing portfolio efficiency. The figure also compares BIVS(+RF) with BIVS*(+RF), where the latter considers all domain values — not just the bounds when the domain size exceeds 100 — showing improved performance with RF, suggesting the removal of the domain size consideration. Both BIVS+RF and BIVS*+RF outperform their non-RF versions, indicating their superior efficiency. Notably, the average gap by BIVS+RF at 100.0s is matched by BIVS only at 938.32s, demonstrating its significant speed advantage.

Problem (#instances)	Min	BIVS	BIVS+RF	RLA	RLA+RF
AircraftAssemblyLine (20)	95.00 (1)	**90.00 (2)**	100.00 (0)	95.00 (1)	95.00 (1)
CarpetCutting (20)	61.29 (9)	57.17 (9)	65.48 (8)	62.67 (8)	**54.48 (11)**
GBACP (20)	78.86 **(11)**	**61.43** (8)	79.69 (10)	69.54 (9)	85.51 (9)
GeneralizedMKP (15)	49.74 **(15)**	**6.78** (14)	6.81 (14)	26.57 (12)	20.26 (13)
HCPizza (10)	**33.56** (10)	34.91 (10)	34.43 (10)	34.67 (10)	34.62 (10)
Hsp (18)	**0.00 (18)**	5.56 (17)	5.56 (17)	5.56 (17)	**0.00 (18)**
KMedian (15)	50.84 (8)	57.32 (7)	**43.96 (9)**	56.01 (7)	50.84 (8)
KidneyExchange (14)	44.63 **(14)**	85.71 (3)	51.11 (10)	**43.68** (13)	52.79 (10)
LargeScaleScheduling (9)	66.76 **(6)**	66.83 **(6)**	66.83 **(6)**	45.56 (5)	**34.44 (6)**
PSP1 (8)	100.00 (0)	100.00 (0)	**87.50 (1)**	100.00 (0)	100.00 (0)
PSP2 (8)	**87.50** (1)	**87.50** (1)	87.62 (1)	**87.50** (1)	87.66 (1)
ProgressiveParty (7)	57.14 (3)	57.14 (3)	57.14 (3)	57.14 (3)	57.14 (3)
RIP (12)	5.06 (12)	6.31 (12)	4.59 (12)	**3.24** (12)	4.78 (12)
Rulemining (9)	100.00 (0)	100.00 (0)	100.00 (0)	100.00 (0)	100.00 (0)
SREFLP (15)	7.27 (15)	**3.08** (15)	7.44 (15)	8.78 (15)	7.72 (15)
Sonet (16)	**1.40** (16)	2.28 (16)	3.42 (16)	2.42 (16)	3.15 (16)
TSPTW1 (8)	87.81 **(1)**	100.00 (0)	87.84 **(1)**	87.81 **(1)**	**87.50 (1)**
TSPTW2 (8)	**75.19 (2)**	87.50 (1)	**75.19 (2)**	75.42 **(2)**	75.24 **(2)**
All (232)	52.02 **(142)**	51.04 (124)	50.23 (135)	49.85 (132)	**49.52** (136)

Table 1 Performances between the methods for each problem. Each column shows the average primal gap over all instances, in percentage, and the instances where at least one feasible solution was found, in parentheses. Best results are highlighted in bold if at least one heuristic was outperformed.

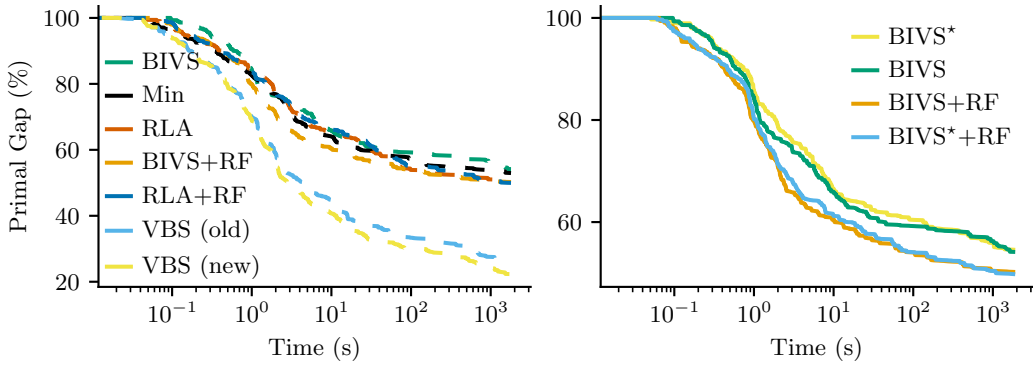

Figure 6 Average primal gap over time on the XCSP^3 instances, in percentage. The right part shows only 4 selectors compared to the left part, and their y-scale differs.

4 Conclusion

Deriving effective, generic value heuristics that balance speed and informativeness remains challenging. Bound-Impact Value Search (BIVS) stands out among these approaches, yet its cost limited its applicability in certain scenarios. By incorporating a restricted fixpoint in the look-ahead process and employing a reverse look-ahead strategy, we significantly reduced costs, making previous restrictions on BIVS usage less relevant. The methods we proposed do not require any training, are well suited for black-box settings, and substantially improve performance. When utilized alone or within a portfolio approach, these strategies continue to enhance the efficiency of solving constrained optimization problems.

References

1 QAPLIB-Problem Instances and Solutions – COR@L, April 2024. [Online; accessed 5. Apr. 2024]. URL: https://coral.ise.lehigh.edu/data-sets/qaplib/qaplib-problem-instances-and-solutions.

2 David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Finding cuts in the tsp (a preliminary report), 1995.

3 Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2023 xcsp3 competition. *arXiv preprint arXiv:2312.05877*, 2023.

4 Gilles Audemard, Christophe Lecoutre, and Charles Prud'Homme. Guiding backtrack search by tracking variables during constraint propagation. In *International Conference on Principles and Practice of Constraint Programming*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

5 Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, volume 4, pages 555–564. MatFyzPress Prague, 1999.

6 Timo Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.

7 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

8 Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: an integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*, 2016.

9 Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3677–3687, 2021.

10 Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial intelligence*, 34(1):1–38, 1987.

11 Emir Demirović, Geoffrey Chu, and Peter J Stuckey. Solution-based phase saving for cp: A value-selection heuristic to simulate local search behavior in complete solvers. In *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*, pages 99–108. Springer, 2018.

12 Jean-Guillaume Fages and Charles Prud'Homme. Making the first solution good! In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1073–1077. IEEE, 2017.

13 Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 140–148. Springer, 2015.

14 P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, pages 31–35, 1992.

15 Diarmuid Grimes, Emmanuel Hebrard, and Arnaud Malapert. Closing the open shop: Contradicting conventional wisdom. In *International conference on principles and practice of constraint programming*, pages 400–408. Springer, 2009.

16 Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

17 Robert Klein and Armin Scholl. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. *European Journal of Operational Research*, 112(2):322–346, 1999.

18 Philippe Laborie. Objective landscapes for constraint programming. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018, Proceedings 15*, pages 387–402. Springer, 2018.

19 Christophe Lecoutre. Ace, a generic constraint solver. *arXiv preprint arXiv:2302.05405*, 2023.

20 Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict (s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

21 Hongbo Li, Minghao Yin, and Zhanshan Li. Failure based variable ordering heuristics for solving csps (short paper). In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2021.

22 Tom Marty, Tristan François, Pierre Tessier, Louis Gautier, Louis-Martin Rousseau, and Quentin Cappart. Learning a Generic Value-Selection Heuristic Inside a Constraint Programming Solver. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2023.25`, `doi:10.4230/LIPIcs.CP.2023.25`.

23 Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021.

24 Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28–June1, 2012. Proceedings 9*, pages 228–243. Springer, 2012.

25 Gilles Pesant. From support propagation to belief propagation in constraint programming. *Journal of Artificial Intelligence Research*, 66:123–150, 2019.

26 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. `doi:10.21105/joss.04708`.

27 Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming–CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27-October 1, 2004. Proceedings 10*, pages 557–571. Springer, 2004.

28 Gerhard Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

29 Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, 1, 2010.

30 Eva Vallada, Rubén Ruiz, and Jose M Framinan. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3):666–677, 2015.

A Additional overview of the XCSP^3 results

Figure 7 shows the cumulated number of instances solved to optimality over time. Table 2 shows the number of optimality proven and best bound founds, for each problem. We can also see that the best heuristic changes depending on the problems, highly impacting the overall readings (for instance BIVS being the best methods regarding the number of best bounds is mostly due to its good performances on the SREFLP problem).

For ranking the solvers in the XCSP^3 competition, one criterion used is a score, telling the number of times a method gave the best known results, compared to its competitors. For each instance and each heuristic:

- 1 point is awarded for proving the unsatisfiability of an instance;
- 0 point is won for proving a solution with a worse bound than its competitors
- 1 point for proving the optimality of a solution

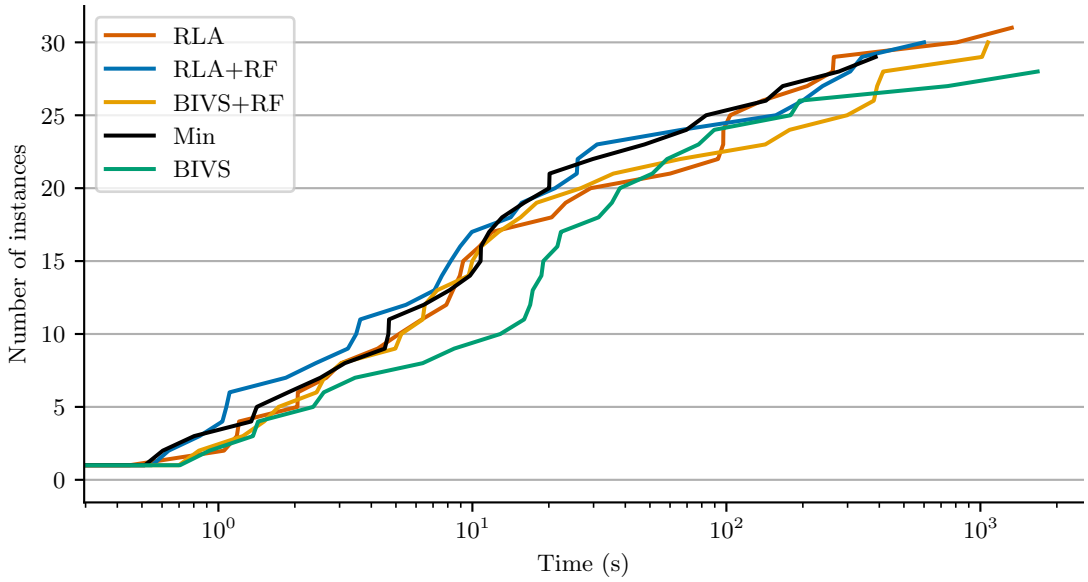

Figure 7 Cactus plot showing the cumulated number of XCSP^3 instances solved to optimality for each method.

Problem (#instances)	Min	BIVS	BIVS+RF	RLA	RLA+RF
AircraftAssemblyLine (20)	0 (1)	0 **(2)**	0 (0)	0 (1)	0 (1)
CarpetCutting (20)	2 (4)	**4 (9)**	2 (4)	3 (6)	2 (4)
GBACP (20)	0 (1)	0 **(7)**	0 (1)	0 (2)	0 (0)
GeneralizedMKP (15)	0 (1)	0 (10)	0 **(12)**	0 (0)	0 (1)
HCPizza (10)	0 **(9)**	0 (3)	0 (4)	0 (2)	0 (3)
Hsp (18)	**17 (18)**	16 (17)	**17** (17)	**17** (17)	**17 (18)**
KMedian (15)	0 (2)	0 (2)	0 **(5)**	0 (1)	0 (3)
KidneyExchange (14)	**2** (3)	1 (2)	**2 (8)**	**2** (4)	**2** (4)
LargeScaleScheduling (9)	0 (0)	0 (0)	0 (0)	0 **(3)**	0 **(3)**
PSP1 (8)	0 (0)	0 (0)	0 **(1)**	0 (0)	0 (0)
PSP2 (8)	0 **(1)**	0 **(1)**	0 (0)	0 **(1)**	0 (0)
ProgressiveParty (7)	3 (3)	3 (3)	3 (3)	3 (3)	3 (3)
RIP (12)	2 (4)	2 (3)	**3 (6)**	**3 (6)**	**3** (5)
Rulemining (9)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
SREFLP (15)	1 (3)	1 **(12)**	1 (1)	1 (2)	1 (1)
Sonet (16)	1 **(13)**	1 (11)	1 (8)	1 (11)	1 (8)
TSPTW1 (8)	0 (0)	0 (0)	0 (0)	0 (0)	0 **(1)**
TSPTW2 (8)	**1** (1)	0 (1)	**1** (1)	**1** (1)	**1** (1)
All (232)	29 (64)	28 **(83)**	30 (71)	**31** (60)	30 (56)

Table 2 Performances between the methods for each problem. Each column shows first the number of instances where optimality was proven, and then number of best bounds found, in parentheses. Best results are highlighted in bold if at least one heuristic was outperformed.

- 1 point is won for providing (a solution with) the best bound on an instance without proving optimality. In case where another solver proved the optimality of this bound, only 0.5 points are granted.

The evolution of the score over time is shown in Figure 8. The score of an heuristic at time t may decrease, for instance if the method provided the best bound to an instance before time t, but a better solution was provided at t by another selector.

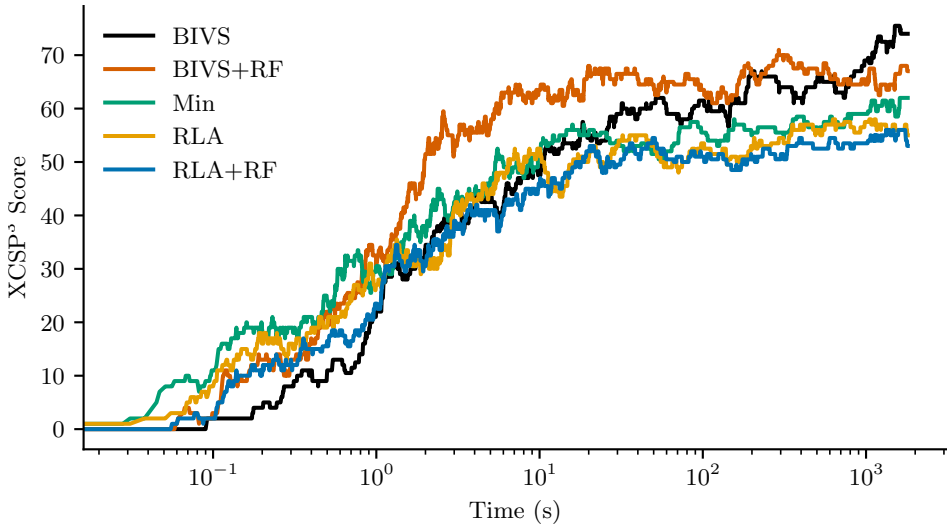

Figure 8 XCSP^3 score over time on the XCSP^3 instances, for each method.

B TSP, QAP and JobShop description

This section describes the models used for the TSP, the QAP and the Jobshop, as well as the decision variables considered and the greedy heuristics that were implemented.

TSP the model is the same as in Example 1

JobShop : a scheduling problem where jobs must be planned, each job being decomposed into tasks having precedence between them, and each tasks being assigned to a given machine. No task can overlap on a machine and the goal consists in minimizing the completion time of the latest task.

QAP : an assignment problem where facilities must be opened at given locations, in order to minimize the sum of distances multiplied by the flow between facilities.

Note that, for all those problems, a first solution satisfying the constraints can always be easily derived. In the case of the TSP, any permutation of nodes corresponds to a valid tour for the salesman. The same reasoning holds for any permutation of facility location for the QAP. Regarding the JobShop, it suffices to provide a timing horizon long enough so that a schedule can always be constructed. Those first solutions being easy to construct, we have implemented a custom greedy value selection heuristic for those problems in order to compare white-box approaches to our own black-box approaches. More precisely, the variables considered for selection and the greedy value selection are as follows:

TSP : the decision variables are the n successor variables in a TSP tour. The greedy value selection consists in visiting the closest city.

JobShop : the decision variables to consider are the precedences between two tasks executed on the same machine. Each variable is a boolean telling, for a given machine, whether a tasks comes before of after another one. Once all precedences between tasks are fixed, the makespan can be assigned to its lower bound in order to produce a valid solution. Given

,

a precedence variable, the greedy heuristic consists in choosing the value producing the most slack, the slack describing how much time is still available between the two tasks.

QAP : with n facilities, the n decision variables are the locations where the facilities will be opened. Given a facility not assigned to a location, the greedy heuristic choose its location as the one minimizing the weighted flow with the other facilities already placed.