Making the first solution good!

Jean-Guillaume Fages, Charles Prud'Homme

Making the first solution good!

Jean-Guillaume Fages[1], Charles Prud'Homme[2]
[1]COSLING S.A.S., France
[2]TASC - IMT Atlantique, France

***Abstract*—Providing efficient black-box search procedures is one of the major concerns for constraint-programming solvers. Most of the contributions in that area follow the *fail-first* principle, which is very useful to close the search tree or to solve SAT/UNSAT problems. However, for real-life applications with an optimization criterion, proving optimality is often unrealistic. Instead, it is very important to compute a good solution fast. This paper introduces a value selector heuristic focusing on objective bounds to make the first solution good. Experiments show that it improves former approaches on a wide range of problems.**

I. Introduction

One of the main strengths of Constraint Programming (CP) is the simplicity of modeling offered by its expressive declarative language. Unfortunately, simple models do not always lead to satisfying performances. For this reason, CP users may have to implement dedicated search heuristics. To make this task easier, high level languages for specifying search strategies have been integrated into CP solvers [1], [2]. However, designing an efficient strategy still requires time and some background in CP. Moreover, such efforts may not be reusable when solving a different problem. Based on this observation, robust black-box search strategies have been proposed in the literature. Such works enabled to improve the trade off between implementation effort and performances of CP solvers on a wide range of problems. Nevertheless, making CP solvers easier to be efficiently used by software engineers remains one of the biggest challenges for the spread of the CP technology. Most black-box search procedures rely on the fail-first principle [3], which is very relevant to constraint satisfaction, and optimization when it comes to closing the search tree. However, it is not efficient when one wants to compute a good quality solution fast, which is the case in most real life applications.

This paper suggests another step towards this Holy Grail, by introducing a value selection heuristic, referred to as *Bound-Impact Value Selector* (BIVS), which improves numerous search strategies on a wide variety of benchmarks, making CP solvers even more competitive when development resources are limited. This paper is organized as follows: Section II provides some background in CP. Then, Section III introduces the BIVS rule, which is experimentally evaluated in Section IV. Finally, Section V shows some conclusions.

II. Constraint Programming Background

Constraint programming is based on relations between variables, which are stated by constraints. A *Constraint Satisfaction Problem* (CSP) is defined as a triplet $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ and consists of a set of n variables $\mathcal{V}$, their associated domains $\mathcal{D}$, and a collection of m constraints $\mathcal{C}$.

A. Main Concepts

Let $\mathcal{X}$ be a universe of values, the domain $d_v \in \mathcal{D}$ associated to a variable $v \in \mathcal{V}$ defines the possible values of v, i.e. $d_v \subseteq \mathcal{X}$. An *assignment*, or *instantiation*, $v^\star$ of a variable v to a value $x \in d_v$ is the reduction of its domain d_v to a singleton, $d_v = \{x\}$. Let $S \subseteq \mathcal{V}$, an assignment $S^\star$ of S is a set such that for all $v_i \in S$, $|d_{v_i}| = 1$. A constraint $c \in \mathcal{C}$ is defined over a set of variables $V_c \subseteq \mathcal{V}$ and defines a set of valid assignments A_c. A *solution* to c is an assignment of its variables, $V_c^\star$, such that $V_c^\star \in A_c$. A constraint c is said to be *satisfiable* if there exists a solution to c. A solution to a CSP is an assignment of all variables of $\mathcal{V}$ such that every constraint of $\mathcal{C}$ is simultaneously satisfied; a CSP is satisfiable if and only if it has a solution. A constraint is equipped with one or more filtering algorithms. A filtering algorithm reduces current domains of V_c by identifying variable-value pairs which cannot satisfy the constraint. The strength of a filtering is qualified through a level of *consistency*, and may vary from one algorithm to another [4]. A *propagation* algorithm applies iteratively filtering algorithms of $\mathcal{C}$ until no more domain reductions can be performed. Solving a CSP consists in performing a Depth First Search (DFS) algorithm with backtrack, during which *branching decisions* are computed, applied and negated. Without loss of generality, 2-way branching will be considered hereafter since it is more powerful and flexible than d-way branching [5]. A branching decision, or simply decision, δ is a triplet $\langle v, o, x \rangle$ composed of a variable v (not yet assigned), an operator o (most of the time $=$, $\neq$, $\leq$ or $\geq$) and a value $x \in d_v$. This triplet represents a constraint over d_v, the domain of v. Each time a decision is applied or negated, its impact is propagated to the CSP by removing some infeasible values from $\mathcal{D}$. If this empties a domain, *i.e.*, the CSP is not satisfiable, a failure occurs and the search algorithm backtracks. Otherwise, another decision is computed, unless a solution has been reached.

Algorithm 1 depicts a recursive implementation of a search algorithm. The main methods that build a deci-

sion for the search algorithm are the variable selection (line 2), the operator selection (line 4) and the value selection (line 5). First, line 2 chooses a variable v among all the non-assigned ones. Whenever all variables are instantiated, the function SELECTVARIABLE returns $\emptyset$, indicating that a solution has been found (line 16). Second, if not all variables are instantiated (line 3), a decision, *e.g.* $\langle v, o, x\rangle$, is computed (lines 2,4,5), *e.g.* $\langle v, =, 1\rangle$. Next, lines 7 to 10 depict the application of δ, *e.g.* $\langle v, =, 1\rangle$, while lines 11 to 14 depict the application of the negation of the decision, *e.g.* $\langle v, \neq, 1\rangle$. Lines 8 and 12 propagate the application or negation of a decision through the CSP. If the function PROPAGATE returns `false`, then the current CSP is not satisfiable. Otherwise, the function returns `true` and the search goes on.

Algorithm 1 Recursive search algorithm.

```
1: procedure SEARCH(⟨V, D, C⟩)
2:     Variable v ← SELECTVARIABLE(⟨V, D, C⟩)
3:     if v ≠ ∅ then
4:         Operator o ← SELECTOPERATOR(v, ⟨V, D, C⟩)
5:         Value x ← SELECTVALUE(⟨v, o⟩, ⟨V, D, C⟩)
6:         ⟨V, D′, C′⟩ ← COPYOF(⟨V, D, C⟩)
7:         APPLY(⟨v, o, x⟩, ⟨V, D, C⟩)
8:         if PROPAGATE(⟨V, D, C⟩) then
9:             SEARCH(⟨V, D, C⟩)
10:        end if
11:        APPLY(⟨v, !o, x⟩, ⟨V, D′, C′⟩)
12:        if PROPAGATE(⟨V, D′, C′⟩) then
13:            SEARCH(⟨V, D′, C′⟩)
14:        end if
15:    else
16:        RECORDSOLUTION(⟨V, D⟩)
17:    end if
18: end procedure
```

A *Constraint Optimization Problem* (COP) is a CSP where one *objective* variable has to be either minimized or maximized. It is solved as a standard CSP in which a new constraint is added upon each solution to state that the next solution should be strictly better. It relies on the enumeration of improving solutions (SAT) until proving that none exists (UNSAT), meaning that the last solution was optimal.

B. Black-Box Search Heuristics

There are many ways of choosing a variable and computing a decision on it. These are referred to as *search strategies*, *labelling strategies* or *heuristics* and have a strong impact on performances. Designing specific search strategies can be a very tough task to do. The concept of black-box search heuristic has naturally emerged from this statement. Most common black-box search strategies observe aspects of the CSP resolution in order to drive the variable selection, and eventually the decision computation (presumably, a value assignment). Three main families of heuristic, stemming from the concepts of *variable impact*, *conflict* and *variable activity*, can be found in most CP solvers.

The impact-based search heuristic (IBS) [6] is based on the importance of a decision for the reduction of the search space, named impact. More precisely, the impact $\bar{I}(\langle v, =, x\rangle)$ of each possible assignment is estimated empirically. An assignment which results in a failure produces a maximal impact. The impact of a variable $\mathcal{I}(v)$ is given by the formula: $\mathcal{I}(v) = \sum_{x \in d_v} \bar{I}(\langle v, =, x\rangle) - |d_v|$. IBS heuristic selects the uninstantiated variable with largest impact and assigns it the least impact value. The computation of impacts is preprocessed at the root node by testing all possible assignments, or a partition of them in case of large domains.

A well known conflict-based search heuristic is the weighted degree ordering heuristic (WDEG) [7]. It maintains a *weight* associated to each constraint, *i.e.*, the number of times it has failed. The weighted degree, $\alpha_{wdeg}(v)$, of a variable v is defined as the sum of weights of its associated constraints which involve at least two uninstantiated variables. WDEG heuristic selects the uninstantiated variable v with the smallest ratio $\frac{|d_v|}{\alpha_{wdeg}(v)}$. It only defines the variable selection, not the value selection.

The activity-based search heuristic (ABS) [8] records the activity of variables during propagation. A variable activity, $A(v)$, is the number of domain modifications induced by decisions involving variable v. An assignment activity, $A(\langle v, =, x\rangle)$, is based on the number of variables modified by the application of $\langle v, =, x\rangle$. A sampling process initializes activities prior to search. ABS heuristic selects the uninstantiated variable v with largest ratio $\frac{A(v)}{|d_v|}$ and assigns it the least activity value.

Last Conflict [9] (LC) and Conflict Ordering Search [10] (COS) are two variable selectors that tend to branch on the variables that trigger conflicts the most. They can be seen as a plugin that can be combined with another variable selector heuristic. LC only interferes with the former search upon backtrack whereas COS will completely replace the former heuristic during search.

Overall, these heuristics are blind to the optimization function and tend to follow the fail-first principle, which is by essence in contradiction with finding a good solution.

III. THE *Bound-Impact Value Selector*

This section presents the contribution of this paper. It defines the BIVS heuristic and shows how such a rule can be easily integrated into a CP solver.

Definition 1: Given a COP in minimization (maximization), BIVS is a value selection heuristic that selects the value with the lowest (highest) objective lower (upper) bound after propagation of such an assignment.

It is worth noticing that BIVS is not a standalone heuristic, but only a value selector. Therefore, it can be combined with any variable selector and operator. This provides a lot of flexibility.

A. Algorithm

Algorithm 2 describes how BIVS selects the best value *w.r.t.* a variable and an operator. An iteration over all values in $\mathcal{D}(v)$ is done in the SELECTVALUE function, lines 4 to 10, wherein each value x is evaluated calling

BOUND function, line 5. As loop goes on, the best bound found so far is stored together with the corresponding value, lines 6 to 9. Finally, the value that gives the smallest bound is returned, line 11, or the variable lower bound otherwise, lines 2 and 3. The BOUND function applies a fake decision $\langle v, o, x\rangle$, line 15, on a copy of the COP, line 14, and evaluates the reduction effect on the objective variable f through propagation (lines 16 to 24). If the propagation fails, an arbitrary big value is returned (line 23) to defer the selection of x. Otherwise, the lower bound of f (in minimization) or its upper bound multiplied by -1 (in maximization)[1] is returned.

Note that BIVS introduces an overhead over a classical value selection. Assuming no fail occurs, the number of nodes explored to reach a solution moves from $O(|\mathcal{V}|)$ to $O(|\mathcal{D}|)$. This can be problematic when domain are very large. If so, Algorithm 2 should be adapted to evaluate variable bounds only, to keep runtime under control.

Algorithm 2 BIVS value selector.

```
global Variable f // objective variable
global Boolean isMinimization // whether to minimize or maximize f
 1: function SELECTVALUE(⟨v, o⟩, ⟨V, D, C⟩)
 2:     int bestValue ← D(v)
 3:     int bestBound ← +∞
 4:     for x ∈ D(v) do
 5:         int bound ←BOUND(⟨v, o, x⟩, ⟨V, D, C⟩)
 6:         if bound < bestBound then
 7:             bestValue ← x
 8:             bestBound ← bound
 9:         end if
10:     end for
11:     return bestValue
12: end function
13: function BOUND(⟨v, o, x⟩, ⟨V, D, C⟩)
14:     ⟨V, D', C'⟩ ← COPYOF(⟨V, D, C⟩)
15:     APPLY(⟨v, o, x⟩, ⟨V, D', C'⟩)
16:     if PROPAGATE(⟨V, D', C'⟩) then
17:         if isMinimization then
18:             return D(f)
19:         else
20:             return −D(f)
21:         end if
22:     else
23:         return +∞
24:     end if
25: end function
```

B. Related work

BIVS is closely related to IBS as it evaluates the effect of an assignment through propagation. However, the evaluation of BIVS is limited to a single variable and its *impact* is only measured on one objective variable bound.

BIVS is also closely related to HBFS [11], which is a another hybridization between the classical DFS used in CP and a *Best-First-Search* (BFS). The HBFS framework stores *open nodes* representing unexplored subproblems, i.e. left and right child nodes of branching decisions. Then, the best node *w.r.t.* the lower (upper) bound of the objective variable in minimization (maximization) is popped and explored using a standard DFS limited to a number of backtracks which, in turn, will push new open nodes. The main differences between BIVS and HBFS are :

- The Breadth aspect of BIVS is local to the domain of one selected variable, in one node of the search tree, whereas it is related to left and right child nodes of the entire search tree in HBFS
- BIVS is a value selector heuristic whereas HBFS is an exploration framework, which still requires a search heuristic to compute decisions.

Therefore, BIVS and HBFS can be combined.

Note also that this heuristic may identify infeasible assignments (see line 23 of Algorithm 2). If we filter out these values from variable domains, then BIVS may be seen as a form of Singleton Arc Consistency [12], restricted to the current decision variable. However, implementing it would be slightly more intrusive and adding another propagation step would delay even more branching on a good value, whereas our will is to compute a good solution fast. Thus, we do not discuss this point further.

IV. EVALUATIONS

This section demonstrates the benefit of plugging BIVS into search procedures, over a wide spectrum of problems. BIVS was implemented in the Choco Solver 4.0.5 [13], a Java library for constraint programming. All the experiments were done on a Mac Pro with a 8-core Intel Xeon E5 at 3Ghz running on MacOS 10.12.5, and Java 1.8.0_25. All experimental details are available on demand.

A. Evaluation on a combinatorial problem

In this experiment, we consider the Traveling Salesman Problem, which is a good illustration of combinatorial optimization. In this problem, constraint satisfaction is trivial: every permutation is a feasible solution, leading to an exponential number of solutions. The difficulty lies in the cost function. Therefore, if the search heuristic is blind to costs, results will be presumably poor. This gets even worse if the search heuristic is oriented to fail-first (note that the fail-first principle is relevant to close the search tree once we have a near-optimal upper bound [14] but not to reach a good solution, which is the focus of this paper).

To avoid wasting time exploring low-quality solutions, CP developers generally implement a search heuristic in their model. We consider the classical CP formulation of the TSP, which relies on successor variables, a CIRCUIT constraint, and ELEMENT constraints to capture the cost of each unit travel. The most natural approach when designing a search heuristic for this model is to branch on the successor variables and select the value associated to the smallest distance. We therefore compare BIVS to this approach in order to evaluate the gap between our generic approach and what an expert would do in few minutes. More precisely, we compare three heuristics:

[1]The upper bound is multiplied by -1 because `selectValue` function stored the smallest evaluation.

- DEFAULT: WDEG combined with LC, selecting the lower bound for each variable. This corresponds to what most users would get[2].
- MINCOST: selects variables according to their input (static) ordering, selecting the value associated to the cheapest distance. This corresponds to what an advanced user would do.
- BIVS: selects variables according to their input (static) ordering, selecting the value leading to the minimal objective lower bound (BIVS). This is the contribution of this paper.

For this purpose, we randomly generated 90 TSP instances, having 10, 50 or 100 nodes. Results[3] on the first solution and after 30 seconds of solving are reported on Table I. Note that the first solution value has a strong impact in practice. Indeed, it is common to solve optimization problems in a satisfaction way, i.e. stopping at the first solution found, provided that the search heuristic ensures that it will be *good enough* [15].

TABLE I
TSP EVALUATION

TSP size	Search	1^{st} sol. time (s)	1^{st} sol. value	last sol. value
10	DEFAULT	0.01	748	241
10	MINCOST	0.01	380	241
10	BIVS	0.01	**310**	241
50	DEFAULT	**0.02**	3775	2043
50	MINCOST	0.03	629	352
50	BIVS	0.60	**455**	**327**
100	DEFAULT	**0.16**	7627	6008
100	MINCOST	0.17	748	497
100	BIVS	9.80	**570**	**428**

Results confirms the first claim that when the heuristic is blind to costs, which is the case of the DEFAULT search, results are very poor. The first solution is really bad and 30 seconds are not sufficient to achieve a decent solution quality. In opposite, both MINCOST and BIVS provide much better good solutions. Quite surprisingly, there is a large gap between the first solution value of BIVS and MINCOST. More astonishing is the same gap after the time limit. As BIVS requires more runtime, it could be expected that MINCOST catches back over BIVS. Indeed, we observe in practice the theoretical runtime overhead induced by BIVS, which evaluates each assignment, leading to $O(n^2)$ nodes instead of exactly n nodes to compute the first solution (note that the first solution is found without failing). This shows that the strength of BIVS clearly compensates its overhead.

B. Benchmarking

In order to obtain a larger view over the potential benefits of using BIVS to solve other problems, it has been evaluated on a wide range of optimization problems from the MiniZinc [16], [17] distribution, which are used in CP solver competitions. The set of benchmarks used consists of the COP instances from the MiniZinc Challenges [18] (2012 to 2016) and is composed of 403 instances from 58 problems. The models rely on a large variety of constraints, including global constraints. Each model specifies a dedicated search strategy, which possibly includes domain splitting.

We compare the performances of using the search defined in the MiniZinc file (FIX) with various black-box search heuristics (ABS, IBS and WDEG[4], all associated with LC). Finally, we evaluate the combination of WDEG(+LC) with BIVS as a value selector. Figure 1 displays the number of instances for which each search configuration was able to find the best solution. Each run had a 15-minute timeout. As can be seen, BIVS allows to obtain the best solution on a larger number of instances. Overall, results show that BIVS is a good value selector for black-box search strategies.

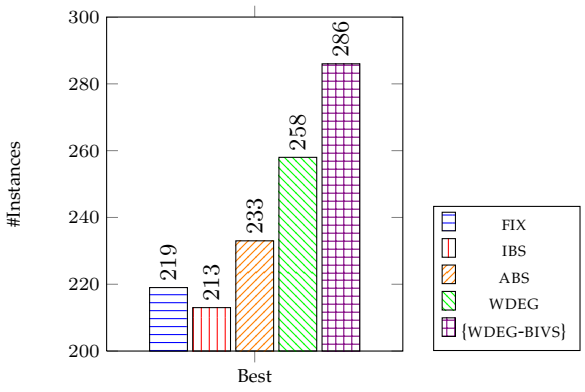

Fig. 1. Evaluation of BIVS on MiniZinc Challenge instances

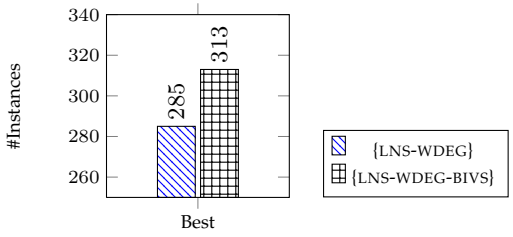

Fig. 2. Evaluation of BIVS on MiniZinc Challenge instances using Large Neighborhood Search together with DomOverWDeg variable selection.

Then, we evaluate the impact of BIVS within a Large Neighborhood Search (LNS) [19] approach, which is probably the most common approach to tackle real-life optimization problems using Constraint-Programming [20], [21], [22]. More precisely we consider the generic approach of [23] that is based on constraint propagation and randomness, and use WDEG as a variable selector. Recall that, mimicking local-search, LNS

[2]Such heuristic is the default one in Choco Solver.

[3]Results are an average over the 30 instances of each size

[4]Using InDomainMin as value selector.

partially unassigns some variables from a given solution and attempts to instantiate them to a close yet different assignment that betters the objective variable. Having a good solution as input, which BIVS presumably does, should improves the entire process. Figure 2 displays the number of instances for which each search configuration was able to find the best solution within a 15-minute timeout. Results show that BIVS allows to obtain the best solution on 28 more instances. Therefore, BIVS combines very well with LNS.

V. Conclusion

This paper introduces the *Bound-Impact Value Selector* (BIVS), which selects the value of a variable that would lead to the best objective bound after propagation of the branching decision. This generic heuristic enables to ensure that the first solution is relatively good, without having to implement a dedicated search strategy. This is very useful for industrial applications that require computing a *good enough* solution as fast as possible.

This heuristic is compatible with any variable selector and operator, leading to a large usage potential. A wide variety of benchmarks have shown BIVS to be highly efficient on numerous problems. Since it is both simple and effective, BIVS is definitely worthwhile to implement into constraint programming solvers. BIVS is used in the default search of Choco Solver since version 4.0.5.

Nevertheless, this selector is not recommended when no solution exists, which happens during the optimality proof. This is where the fail-first should be considered. Unfortunately, knowing when to switch from one approach to the other remains an open question. Without such a rule, our approach is to use both in a parallel portfolio sharing the objective bound. In particular, this has been used by Choco Solver for the parallel tracks of MiniZinc[5] and XCSP3[6] competitions of 2017, where the solver is used as a black-box.

References

[1] . IBM ILOG, *CPLEX CP Optimizer Manual*, http://www-01.ibm.com/software/inte- gration/optimization/cplex-cp-optimizer/, 2009.

[2] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P. J. Stuckey, "Search combinators," in *Principles and Practice of Constraint Programming*, 2011, pp. 774–788.

[3] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," in *Proceedings of the 6th international joint conference on Artificial intelligence - Volume 1*, ser. IJCAI'79. Morgan Kaufmann Publishers Inc., 1979, pp. 356–364.

[4] C. Bessière, "Constraint propagation," in *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, F. Rossi, P. v. Beek, and T. Walsh, Eds. Elsevier Science Inc., 2006, ch. 3.

[5] D. G. Mitchell, "Resolution and constraint satisfaction," in *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, 2003, pp. 555–569. [Online]. Available: https://doi.org/10.1007/978-3-540-45193-8_38

[6] P. Refalo, "Impact-based search strategies for constraint programming," in *Principles and Practice of Constraint Programming*, 2004, pp. 557–571.

[7] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *European Conference on Artificial Intelligence*, 2004, pp. 146–150.

[8] L. Michel and P. Van Hentenryck, "Activity-based search for black-box constraint programming solvers," in *CPAIOR*, 2012, pp. 228–243.

[9] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal, "Last conflict based reasoning," in *ECAI 2006, Including (PAIS 2006), Proceedings*, ser. Frontiers in Artificial Intelligence and Applications, vol. 141. IOS Press, 2006, pp. 133–137.

[10] S. Gay, R. Hartert, C. Lecoutre, and P. 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*, ser. Lecture Notes in Computer Science, G. Pesant, Ed., vol. 9255. Springer, 2015, pp. 140–148.

[11] D. Allouche, S. De Givry, G. KATSIRELOS, T. Schiex, and M. Zytnicki, "Anytime hybrid best-first search with tree decomposition for weighted CSP," in *CP 2015*, Cork, Ireland, Aug. 2015, p. 17 p.

[12] C. Bessiere and R. Debruyne, "Theoretical analysis of singleton arc consistency and its extensions," *Artif. Intell.*, vol. 172, no. 1, pp. 29–41, 2008.

[13] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Solver Documentation*, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. [Online]. Available: http://www.choco-solver.org

[14] J. Fages, X. Lorca, and L. Rousseau, "The salesman and the tree: the importance of search in CP," *Constraints*, vol. 21, no. 2, pp. 145–162, 2016.

[15] S. Kadioglu, M. Colena, S. Huberman, and C. Bagley, "Optimizing the cloud service experience using constraint programming," in *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, ser. Lecture Notes in Computer Science, G. Pesant, Ed., vol. 9255. Springer, 2015, pp. 627–637. [Online]. Available: https://doi.org/10.1007/978-3-319-23219-5_43

[16] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: towards a standard cp modelling language," in *Principles and Practice of Constraint Programming*, ser. CP'07. Springer-Verlag, 2007, pp. 529–543.

[17] NICTA, *MiniZinc and FlatZinc*, http://www.g12.cs.mu.oz.au/minizinc/, 2011.

[18] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer, "The minizinc challenge 2008-2013," *AI Magazine*, vol. 35, no. 2, pp. 55–60, 2014. [Online]. Available: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2539

[19] P. Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, 1998, pp. 417–431. [Online]. Available: https://doi.org/10.1007/3-540-49481-2_30

[20] L. Perron and P. Shaw, "Combining forces to solve the car sequencing problem," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, 2004, pp. 225–239. [Online]. Available: https://doi.org/10.1007/978-3-540-24664-0_16

[21] P. Schaus, "Variable objective large neighborhood search: A practical approach to solve over-constrained problems," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, 2013, pp. 971–978. [Online]. Available: https://doi.org/10.1109/ICTAI.2013.147

[22] M. Lombardi and P. Schaus, "Cost impact guided LNS," in *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, 2014, pp. 293–300. [Online]. Available: https://doi.org/10.1007/978-3-319-07046-9_21

[23] L. Perron, P. Shaw, and V. Furnon, "Propagation guided large neighborhood search," in *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, 2004, pp. 468–481. [Online]. Available: https://doi.org/10.1007/978-3-540-30201-8_35

[5] http://www.minizinc.org/challenge2017/results2017.html

[6] http://xcsp.org/competition