Defining Propagators in MiniZinc

Jip J. Dekker
Department of Data Science and Artificial Intelligence, Monash University, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Australia

Peter J. Stuckey
Department of Data Science and Artificial Intelligence, Monash University, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Australia

Guido Tack
Department of Data Science and Artificial Intelligence, Monash University, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Australia

Huu Quang Tran
Department of Data Science and Artificial Intelligence, Monash University, Australia

Markus Wagner
Department of Data Science and Artificial Intelligence, Monash University, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Australia

Abstract

Solver-independent modelling languages such as MiniZinc are widely used because they enable rapid prototyping of complex optimisation models without committing to a particular solver technology. However, when using a fixed constraint programming solver, one can extend it with custom problem-specific propagators. This can potentially achieve significantly stronger propagation and improved performance. In contrast, within a solver-independent framework, such constraints are typically expressed through decompositions into simpler primitives, which may weaken propagation and increase runtime.

This paper presents an extension to MiniZinc that enables specifying custom propagators directly at the modelling level. These propagators are executed via solver callbacks, preserving solver independence while enabling specialised propagation. Beyond modelling-level propagators, our framework enables the integration of externally implemented propagators, making it possible to incorporate propagation for complex black-box functions, such as simulation models. We demonstrate that this enables the rapid development of efficient problem-specific propagators and facilitates hybrid white-box/black-box optimisation approaches.

2012 ACM Subject Classification Software and its engineering → Constraint and logic languages

Keywords and phrases MiniZinc, black-box solving, indexicals

Funding This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

1 Introduction

Constraint programming (CP) provides a powerful approach to solving discrete optimisation problems, combining specialised reasoning algorithms for global constraints with user-programmable and autonomous search to find good solutions quickly. Building a *solver-dependent* discrete optimisation solution, for a particular CP solver, allows the user to (a) create new specialised propagators for the problem at hand, and (b) create new specialised

search strategies for the problem at hand. But recently the focus of constraint programming has shifted to *solver-independent* uses, where modelling makes use of the powerful global constraint modelling viewpoint of CP, and the model can be run on any solver supported by the modelling system: CP, MIP, SAT, SMT, or otherwise. This avoids lock-in to a particular solver, or even solving technology, which can be highly advantageous. But in doing so, we have lost the abilities (a) and (b) above to specialise the solver to the problem at hand. This specialisation is typically only feasible for optimisation practitioners with highly specialised expertise in developing for a particular solver.

In this paper, we demonstrate how we can recover some of the advantages of (a), creating specialised propagators in a solver-independent way, but still applicable to any CP solver (with some minor extensions), without requiring expertise in the underlying solver implementation.

▶ **Example 1.** Consider Kakuro [12]. In this problem, each clue induces a sum constraint together with an `all_different` constraint. For example, we may have $x_{12}+x_{22}+x_{32}+x_{42} = 12$ together with `all_different`$([x_{12}, x_{22}, x_{32}, x_{42}])$. Both constraints are supported by standard CP solvers, but propagating the two individually is weaker than reasoning over their interaction. With initial domains $D(v) = 1..9$ for $v \in \{x_{12}, x_{22}, x_{32}, x_{42}\}$, neither constraint propagates in isolation. Joint reasoning immediately reduces all domains to $D(v) = 1..6$, since any three distinct digits must sum to at least $1 + 2 + 3 = 6$. Introducing a native `kakuro` propagator in each solver is often impractical. Defining this propagation directly at the modelling level is therefore very attractive.

To enable solver-independent propagator definitions, we adapt ideas from **black-box optimisation** [3] and integrate them into MiniZinc and its backend solvers. Black-box optimisation addresses problems in which the objective function or the constraints can only be evaluated at selected input points, with no explicit analytical form or derivative information available. This setting arises naturally when the objective is computed by a simulation, an external program, or a complex legacy system. A defining feature of black-box optimisation is that evaluations are often expensive and may provide limited information beyond function values or feasibility. As a result, algorithms aim to minimise the number of calls to the external oracle. Typical approaches include local search methods and evolutionary algorithms, as well as methods that build surrogate models using sampled points. Black-box optimisation is widely used in simulation-based engineering design, control, and other domains where optimisation must be coupled with external computational components [2].

The core contribution of this paper is an architecture that enables users to specify **black-box propagators**, functions that are external to the solver, and which solvers can execute through a well-defined, generic interface. We extend the idea of black-box optimisation to enable not only an evaluation of (fixed) assignments, but also a black-box filtering of domain bounds. In addition to enabling hybrid white-box/black-box workflows (combining explicit constraints modelled in MiniZinc with external black-box functions), this also allows us to write bounds propagators directly in MiniZinc, in a solver-independent way. Our experiments show that this architecture yields competitive performance on both established black-box benchmarks and for MiniZinc-defined propagators.

For the rest of the paper, we assume familiarity with standard constraint programming terminology, including finite-domain variables, propagators, and search. We adopt the usual operational model in which propagation enforces local consistency over variable domains, interleaved with branching during search [11].

2 Modelling Architecture

This section introduces a MiniZinc modelling layer for specifying black-box propagators.

2.1 Propagation functions

Constraints in MiniZinc are defined by *predicates* that take decision variables (and, optionally, fixed parameters) as arguments. For black-box propagators, we define a corresponding predicate without a body (i.e., without a decomposition into simpler constraints). To indicate to the MiniZinc compiler that this predicate is defined externally, we *annotate* it and introduce an additional function that defines the actual propagation.

The most basic kind of propagator we can define through this interface is a *value propagator*, which simply checks whether an assignment satisfies the constraint:

```
% Constraint declaration
predicate baz(list of var int: x) ::minizinc_value_propagator;
% Propagator definition
test baz(list of int: x) = /* Check whether x satisfies baz */;
```

This defines a predicate `baz`, taking a list of integer variables as an argument, and annotates it as a value propagator. The corresponding propagation function is a simple `test` that has the same arguments as the predicate, but instead of variables (`list of var int`), it receives fixed values (`list of int`) and returns a simple true or false result. Since the test `baz` is simply an overloaded version of the predicate `baz`, MiniZinc will also be able to use it during translation of the model whenever all arguments are fixed.

Since constraints often implement functional relations, it is convenient to provide support for *functional value propagators*. Rather than simply checking whether a relational constraint holds, these can compute an output value. This will prevent the solver from blindly generating and testing all possible outputs, and can therefore lead to significantly better performance. The definition is analogous to relational value propagators but using functions:

```
% Constraint declaration
function list of var int: bar(list of var int: x)
    ::minizinc_value_propagator_fn;
% Propagator definition
function list of int: bar(list of int: x) = /* Compute bar(x) */;
```

Finally, the architecture supports defining black-box *bounds propagators*. The propagation function now takes the variable bounds as input and returns the updated (filtered) bounds:

```
% Constraint declaration
predicate foo(list of var int: x) ::minizinc_bounds_propagator;
% Propagator definition
function list of tuple(int,int): foo(list of tuple(int,int): x) =
  /* Compute bounds of foo(x) */;
```

As the example shows, the propagation function receives, for each argument variable, a tuple containing its lower and upper bound, and it returns the updated bounds. Defining propagators in this way is closely related to the concept of *indexicals* [14], which were introduced as a high-level specification language for propagators in constraint logic programming systems. An indexical specifies, for a variable, the set of values that remain consistent given the current domains of other variables. These specifications are compiled into efficient domain filtering procedures. Indexicals were central to early high-performance CLP(FD) systems like SICStus Prolog, where they formed the core of constraint implementations [4]. Building on this idea, indexicals have been extended to offer a higher-level description language for

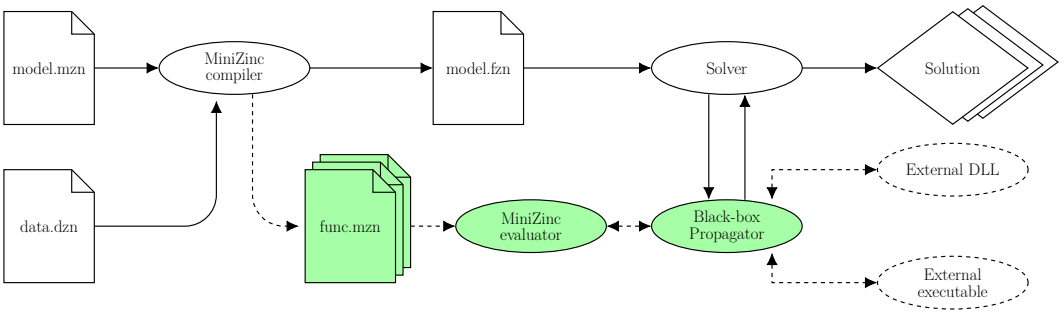

Figure 1 Architecture of MiniZinc with black-box propagation functions.

propagators targeting global constraints [9]. Further work has generalized indexical-style descriptions to richer constraint domains, including finite set constraints [13]. Our propagation functions can be used to define indexical propagators directly. Since we are using MiniZinc for defining bounds, our language is much more expressive than the original indexical language, allowing us to define more complex propagation rules. On the other hand, we lose properties such as automatically determining whether the bounds definition is monotonic.

In addition to propagation functions defined in MiniZinc, the architecture also supports fully externally defined propagation functions. These can be used to implement black-box constraints driven by external evaluators, such as complex numerical calculations or simulation models. As above, we define a predicate or function over decision variables as the modelling interface for the constraint. We then use an annotation to specify an external executable or shared library that implements that function:

```
% Predicate implemented in external executable
predicate foo(list of var int: x)
    ::blackbox_exec("bb_executable.py");
% Function implemented in shared library
function list of var int: bar(list of var int: x)
    ::blackbox_dll("bb.dll");
```

The examples here present minimal signatures, but the architecture supports propagators with additional arguments. Firstly, propagators can also be defined over `list of var float` arguments in addition to, or instead of, `list of var int`. However, no additional decision-variable arguments are permitted beyond these lists of decision variables. In addition, fixed parameter arguments can be given, which are simply passed through to the propagation function. When convenient, users can define alternative interfaces using simple wrappers (e.g. `predicate foo(var int: a, var int: b) = foo([a,b]);`).

Figure 1 gives an overview of the entire architecture. The MiniZinc compiler translates propagation functions into code that can be executed by a MiniZinc evaluator. Calls to black-box predicates are translated into FlatZinc for the backend solver, which then uses a generic external interface to execute the functions (including those implemented by the MiniZinc evaluator). Section 5 contains more implementation details.

2.2 Defining reasons for black-box propagation

Lazy clause generation solvers [10], such as Chuffed and OR-Tools CP-SAT, must provide reasons for each propagated change. A reason is a conjunction of literals implying the propagated change, where a literal denotes a condition on a decision variable domain, such as $x \leq v$, $x \geq v$, $x = v$, or $x \neq v$. For black-box propagators, we need to either infer or

explicitly define valid reasons for their propagations. A conservative default is to assume that all input literals explain all outputs. For value propagators, this means using the conjunction of input-fixing equalities, i.e. $x_i = v_i$. Using this default is generally appropriate: if different outputs depend on disjoint input subsets, the function should be split, so each part can propagate earlier with tighter reasons.

For bounds propagation, the same coarse strategy is often too weak. Using all input bound literals, i.e. $x_i \geq lb_i$ and $x_i \leq ub_i$, for every output bound can significantly reduce explanation quality because lower- and upper-bound updates usually depend on different subsets of inputs. To support tighter explanations, we allow users to define a `_reason` function for a propagator. It is evaluated once during flattening. For each variable index `i`, it returns a tuple (i, R_{lb}, R_{ub}), where R_{lb} and R_{ub} are lists of literals represented as pairs `(j,PropBnd)`, with `PropBnd` being either `PR_LB` or `PR_UB`. Because this template is reused during search, it must be a sound over-approximation for all reachable domains. The following signature can be used for the `foo` propagator introduced above:

```
function list of tuple(int, list of tuple(int,PropBnd), list of
    tuple(int,PropBnd)): foo_reason(list of tuple(int,int): x) =
  /* REASON TEMPLATE */
```

In some cases, even tighter reasons are possible with domain-dependent computation at propagation time. For example, for `element`, explanations for the result can ignore values whose indices are outside the current domain of the index variable. A possible future extension is therefore to allow `_reason` evaluation at propagation time.

3 Defining propagators in MiniZinc

This section shows how to use black-box propagators to implement stronger propagation for the Kakuro problem from the introduction, and a more complex propagator for the spread and gini constraints.

3.1 Case Study: Kakuro

Consider the Kakuro constraint `kakuro`$([x_1, \ldots, x_n], s)$, which requires the variables x_i to sum to a constant s and be all different. We can implement this using the following decomposition, which combines standard constraints with a solver-independent MiniZinc propagator that captures the interaction between them.

```
predicate kakuro(list of var int: x, int: s) =
  all_different(x) /\ sum(x) = s /\ kakuro_prop(x, s);

predicate kakuro_prop(list of var int: x, int: s)
  ::minizinc_bounds_propagator;
```

The propagator only needs to enforce additional pruning beyond what is achieved by the decomposition itself. Below, we show the upper-bound computation; the lower-bound computation is analogous.

```
function list of tuple(int, int): kakuro_prop(list of tuple(int,int): x,
    int: s) =
  let {list of int: flb = kakuro_prop_lb(x.2, s);
       list of int: fub = kakuro_prop_ub(x.1, s);
  } in [(flb[i], fub[i]) | i in index_set(x)];

function list of int: kakuro_prop_ub(list of int: lbx, int: s) =
  let {int: l = length(lbx);
```

```
        array[1..l] of int: slbx = sort(lbx);
        tuple(int,int): adjlb = kub_rec(slbx, 1, l, slbx[1]-1, sum(lbx));
        int: slack = s - adjlb.1;
        array[1..l] of int: fub = [ if slack >= 0 then slack + adjlb.2
                                    else lbx[i]-1 endif| i in 1..l ]; }
  in fub;

function tuple(int,int): kub_rec(list of int:slb, int:i, int:l,
    int:prev, int: lb) =
  if i > l then (lb, prev)
  elseif prev < slb[i] then kub_rec(slb, i+1, l, slb[i], lb)
  else kub_rec(slb, i+1, l, prev+1, lb+prev+1-slb[i]) endif;
```

After sorting variable lower bounds (`slbx`), it uses `kub_rec` to compute the adjusted lower bound and the highest (adjusted) lower bound used in the calculation. It scans the list and, whenever `slb[i]` fails to increase by at least one over `prev`, adds the excess `prev+1-slb[i]` to the adjusted sum. The slack (difference between adjusted bound and sum) is calculated. If the slack is not negative, then we set the upper bound of each variable to the slack added to the largest lower bound used; otherwise, we can create an empty domain.

Consider again Example 1. Initially, the sorted list of lower bounds is [1,1,1,1], which sums to 4. The call to `kub_rec` computes an adjusted lower bound of $10 = 4 + 0 + 1 + 2 + 3$, with the highest adjusted lower bound of 4. We then set each upper bound to at most $6 = (12 - 10) + 4$.

For `kakuro_prop`, the upper bound of variable `i` depends on the lower bounds of all other variables, and the lower bound depends on the upper bounds of all other variables. The following `kakuro_prop_reason` definition captures these propagation reasons.

```
function list of tuple(int, list of tuple(int,PropBnd), list of
    tuple(int,PropBnd)): kakuro_prop_reason(list of tuple(int,int): x,
    int: s) =
[(i, [(j, PR_UB) | j in index_set(x) where j!=i],
     [(j, PR_LB) | j in index_set(x) where j!=i] ) | i in index_set(x)];
```

Over the Kakuro instances, this propagator reduced the search nodes and failures by more than 50%, but does not improve solve times.

3.2 Case study: Gini and Spread Propagators

Spread and Gini are dispersion measures that are useful for modelling fairness, where one aims to measure the balance between different stakeholders. Ek *et al.* [7] introduced dedicated propagators for these functions that showed substantial performance improvements over decompositions. We replicate their fair job-shop benchmark, comparing their decompositions and Chuffed propagators against a value and a bounds MiniZinc propagator, BB (value) and BB (bounds), within the same solver. The benchmark uses two optimisation modes, MinDis (minimise dispersion under an efficiency bound) and MaxEff (maximise efficiency under a dispersion bound), with dispersion defined by either Spread or Gini. The MiniZinc functions used in this case study are given in Appendices A and B. Figure 2 summarises the results. Unsurprisingly, the native Chuffed propagators remain strongest overall, but BB (bounds) consistently narrows the gap and substantially improves over decomposition. BB (value) is generally weaker than BB (bounds), but remains competitive on several settings, particularly where dispersion constraints activate later during search. Overall, this case study indicates that MiniZinc-level propagator definitions can recover a substantial fraction of native-propagator performance while preserving solver-independent modelling.

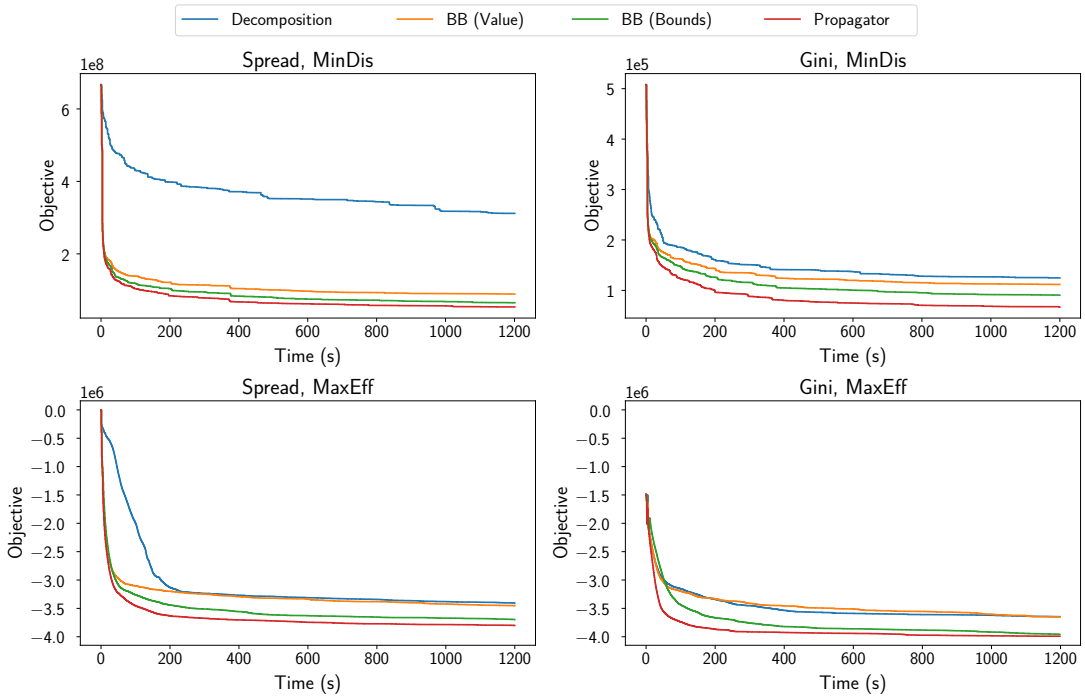

Figure 2 Primal integral for different Chuffed implementations of Gini and Spread on (fair) jobshop scheduling benchmarks [7].

4 Optimisation with black-box functions

By using externally defined propagation functions, our method opens up exciting opportunities for combining MiniZinc modelling with black-box optimisation. Many solvers that support MiniZinc already include techniques common in black-box optimisation. Constraint-based local search (CBLS) solvers [8] are particularly well-suited, and we have extended Atlantis [1] with support for black-box propagators. CP solvers can likewise exploit local-search techniques such as large neighbourhood search and simulated annealing [5, 6], which have proven effective. This lowers the barrier to prototyping black-box and hybrid black-box/white-box optimisation applications in a solver-independent modelling workflow. A CP solver can enforce combinatorial constraints and propagate domains before invoking an expensive black-box evaluator, so calls are concentrated on candidates that remain feasible. This is difficult to realise robustly in pure local-search pipelines. Such pipelines often require complex search strategies to avoid infeasible states, or they must resort to reactive constraint checking.

4.1 Case study: wind farm layout

Wind farm layout optimisation aims to maximise expected energy output by choosing turbine locations while accounting for wake interactions and geometric placement constraints [15]. This problem is practically important for renewable-energy planning, but the objective is complex because wake effects couple many turbine pairs, making it a good fit for black-box optimisation. Wagner *et al.* [15] describe a bespoke local-search solver for wind-farm layout, considering fixed farm sizes and comparing energy output after a fixed runtime budget.

We recreated the application of Wagner *et al.* in MiniZinc, where placement constraints

are enforced by the model and energy output is computed by a black-box function. The search is performed by Gecode, using simulated annealing (initial temperature 20,000 and cooling rate 0.955) and large neighbourhood search, where each turbine is independently released with probability 10% in each neighbourhood. The MiniZinc model is given in Appendix C. As shown in Table 1, the MiniZinc approach remains within approximately 2% of the specialised solver across all tested instance sizes, which is a strong outcome for solver-independent prototyping, avoiding the creation of a bespoke solution.

Table 1 Comparison of the MiniZinc implementation with the specialised solver of Wagner *et al.* [15]. Achieved objective (energy output in kW) after 15 min. for different wind farm sizes.

n	MiniZinc (kW)	Wagner *et al.* (kW)	diff. (%)
10	72,798,726	72,973,799	0.24%
20	141,881,017	144,800,330	2.02%
30	207,745,382	211,624,219	1.83%
40	270,678,378	276,527,895	2.12%
50	331,536,366	337,909,824	1.89%

5 Implementation

For each call to a MiniZinc predicate annotated with `::blackbox_exec` or `::blackbox_dll`, the MiniZinc compiler produces corresponding FlatZinc calls to a generic `fzn_blackbox` constraint with suitable arguments, including the target executable or dynamic library. For propagation functions defined in MiniZinc, the compiler emits the corresponding `fzn_blackbox` constraint in FlatZinc and annotates it with the *MiniZinc evaluator* (see Figure 1) as the target dynamic library, and arguments that define the actual propagation function to be evaluated.

Solvers therefore only need to implement two generic propagators to support this architecture: a value-consistent variant and a bounds-consistent variant. The value-consistent propagator is scheduled once all input decision variables are fixed, and then invokes the external function to compute the output values. The bounds-consistent propagator is scheduled whenever an input bound changes, and invokes the external function to derive tightened bounds for the outputs. Because external function evaluations may be expensive, both propagators are scheduled at low priority to reduce unnecessary calls.

External functions are called either from a dynamically loaded shared library or by communicating with a subprocess over pipes. In the shared-library variant, the solver dynamically loads the library as a plug-in at start-up (e.g. using `dlopen`) and resolves a symbol `fzn_blackbox` whose C signature is as follows.

```
void fzn_blackbox(
  const int64_t* int_in,    size_t int_in_size,
  const double*  float_in,  size_t float_in_size,
  int64_t*       int_out,   size_t int_out_size,
  double*        float_out, size_t float_out_size
);
```

The propagator then calls this function at each propagation step. For a value propagator, the `int_in` and `float_in` arrays simply contain the values the variables are assigned to. For the bounds propagation variant, both the input and output arrays encode variable bounds and therefore contain twice as many elements as the number of decision variables.

In the subprocess variant, the solver emits the input values as a delimiter-separated string stream onto the standard input stream of the helper process and expects output on standard output in the same format, enabling language-agnostic implementations. Integer values and floating-point values are comma-separated, the integer and floating-point segments are separated by a semicolon, and each message is terminated by a newline. For example, sending integers 5 and -7, and floating-point values 2.5 and 1.125, is achieved by writing `"5,-7;2.5,1.125\n"`.

The implementations of the MiniZinc extensions as well as the modified solvers will be made available under open-source licences.

6 Conclusion

We presented an extension to MiniZinc that enables user-defined value and bounds propagators through a solver callback interface. This recovers much of the benefit of specialised propagation in solver-independent modelling, without requiring users to implement solver internals. Our case studies indicate that MiniZinc-defined propagators can be competitive with specialised implementations on Gini/Spread fairness benchmarks. The same mechanism also enables practical hybrid white-box/black-box workflows, where CP propagation filters infeasible candidates before expensive external evaluation. When used directly as a black-box interface, this callback infrastructure also achieves performance close to that of wind farm black-box optimisation. Overall, the approach provides a useful path for prototyping problem-specific and black-box optimisation methods while retaining the flexibility of solver-independent models.

References

1 Atlantis CBLS solver, 2026. URL: `https://github.com/astra-uu-se/atlantis`.

2 Stéphane Alarie, Charles Audet, Aïmen E. Gheribi, Michael Kokkolaras, and Sébastien Le Digabel. Two decades of blackbox optimization applications. *EURO Journal on Computational Optimization*, 9:100011, 2021. URL: `https://www.sciencedirect.com/science/article/pii/S2192440621001386`, `doi:10.1016/j.ejco.2021.100011`.

3 Charles Audet and Warren Hare. *Derivative-free and blackbox optimization*, volume 1. Springer, 2017.

4 Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, pages 191–206, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

5 Jip J. Dekker, Maria Garcia de la Banda, Andreas Schutt, Peter J. Stuckey, and Guido Tack. Solver-independent large neighbourhood search. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 81–98. Springer, 2018. `doi:10.1007/978-3-319-98334-9\_6`.

6 Jip Joris Dekker. *A Modern Architecture for Constraint Modelling Languages*. PhD thesis, Monash University, 11 2021. URL: `https://bridges.monash.edu/articles/thesis/A_Modern_Architecture_for_Constraint_Modelling_Languages/16968229`, `doi:10.26180/16968229.v1`.

7 Alexander Ek, Andreas Schutt, Peter J. Stuckey, and Guido Tack. Explaining propagation for gini and spread with variable mean. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, Haifa, Israel,*

July 31 - August 8, 2022, volume 235 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.CP.2022.21, doi:10.4230/LIPICS.CP.2022.21.

8 Laurent Michel and Pascal Van Hentenryck. *Constraint-Based Local Search*, pages 223–260. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-07124-4_7.

9 Jean-Noël Monette, Pierre Flener, and Justin Pearson. Towards solver-independent propagators. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, pages 544–560, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

10 O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

11 Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. 31(1), December 2008. doi:10.1145/1452044.1452046.

12 Helmut Simonis. Kakuro as a constraint problem. In *Proceedings of the Seventh International Workshop of Constraint Modelling and Reformulation (MODREF2008)*, 1008. https://www.researchgate.net/publication/228524341_Kakuro_as_a_Constraint_Problem.

13 Guido Tack, Christian Schulte, and Gert Smolka. Generating propagators for finite set constraints. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, pages 575–589, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

14 Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *The Journal of Logic Programming*, 37(1):139–164, 1998. URL: https://www.sciencedirect.com/science/article/pii/S0743106698100067, doi:10.1016/S0743-1066(98)10006-7.

15 Markus Wagner, Jareth Day, and Frank Neumann. A fast and effective local search algorithm for optimizing the placement of wind turbines. *Renewable Energy*, 51:64–70, 2013. URL: https://www.sciencedirect.com/science/article/pii/S0960148112005745, doi:10.1016/j.renene.2012.09.008.

A MiniZinc Function for Spread LB

```
% v-centred assignment 0v (Ek et al. Definition 2)
function float: theta(tuple(int, int): x, float: v) =
  if x.2 <= v then
    x.2
  elseif x.1 >= v then
    x.1
  else
    v
  endif;

% LBV'(X, v) = (1/n) * sum_i (theta(x_i, v) - v)^2
function float: lbv_prime(array[int] of tuple(int, int): X, float: v) =
  sum(i in index_set(X))(
    pow(theta(X[i], v) - v, 2)
  ) / length(X);


function int: spread_lb(array[int] of tuple(int, int): X, tuple(int,
    int): M, int: s) =
  let {
    int: m_l = M.1 + ((M.2 - M.1) div 2);
    int: m_r = m_l + 1;
    float: V_l = lbv_prime(X, m_l / length(X));
    float: V_r = lbv_prime(X, m_r / length(X));
  } in
  if V_l = 0 \/ V_r = 0 then
    0
  elseif V_l = V_r then
```

```
    floor(V_l * s)
  elseif V_l < V_r then
    if M.1 >= m_l then
      floor(V_l * s)
    else
      spread_lb(X, (M.1, m_l), s)
    endif
  elseif m_r >= M.2 then
    floor(V_r * s)
  else
    spread_lb(X, (m_r, M.2), s)
  endif;
```

B MiniZinc Function for Gini LB

```
% v-centred assignment Ov (Ek et al. Definition 2)
function float: theta(tuple(int, int): x, float: v) =
 if x.2 <= v then
   x.2
 elseif x.1 >= v then
   x.1
 else
   v
 endif;

% Gini(X, v) for the v-centred assignment, using Ek et al. Eq. (2) on
    the sorted Ov-values
function float: gini_inner(array[int] of tuple(int, int): X, float: v) =
 let {
   any: t = [theta(x, v) | x in X];
   any: denom = length(X) * sum(t);
 } in
 if denom = 0 then
   0
 else
   sum(i in index_set(X), j in i..max(index_set(X)))(
     abs(t[i] - t[j])
   ) / denom
 endif;

% Move l left while Gini(X, Bl) = Gr, stopping either at L or when it
    differs.
% Returns (l', Gl').
function tuple(int, float): gini_move_left(
   array[int] of tuple(int,int): X,
   array[int] of float: B,
   int: L,
   int: l,
   float: Gr
) =
 let { float: Gl = gini_inner(X, B[l]) } in
 if (Gl = Gr) /\ (l > L) then
   gini_move_left(X, B, L, l - 1, Gr)
 else
   (l, Gl)
 endif;

% Binary chop from Algorithm 2, but returning the index m (= R at
    termination)
function int: gini_argmin_index(
   array[int] of tuple(int,int): X,
   array[int] of float: B,
   int: L,
```

```
    int: R
) =
 if L >= R then
   R
 else
   let {
     int: l0 = L + floor((R - L) / 2);
     int: r0 = l0 + 1;
     float: Gl0 = gini_inner(X, B[l0]);
     float: Gr0 = gini_inner(X, B[r0])
   } in if Gl0 = Gr0 then
     if l0 > L then
       let {
         tuple(int, float): t = gini_move_left(X, B, L, l0 - 1, Gr0);
         int: l = t.1;
         float: Gl = t.2
       } in
         if Gl = Gr0 /\ l = L then
           % matches: else { L <- r; break } then continue with (L=r, R)
           gini_argmin_index(X, B, r0, R)
         elseif Gl < Gr0 then
           gini_argmin_index(X, B, L, l)
         else
           gini_argmin_index(X, B, r0, R)
         endif
     else
       % l == L: set L <- r and continue
       gini_argmin_index(X, B, r0, R)
     endif
   elseif Gl0 < Gr0 then
     gini_argmin_index(X, B, L, l0)
   else
     gini_argmin_index(X, B, r0, R)
   endif
 endif;

function int: gini_lb(array[int] of tuple(int,int): X, int: s) =
 let {
   int: n = length(X);
   array[1..2*n] of float: B =
     sort(
       [ X[i].1 | i in 1..n ] ++
       [ X[i].2 | i in 1..n ]
     );
   int: m = gini_argmin_index(X, B, 1, 2*n);
   float: G = gini_inner(X, B[m])
 } in floor(G * s);
```

C MiniZinc Model for Wind Farm Layout

```
include "experimental/on_restart.mzn";
include "experimental/blackbox.mzn";
include "globals.mzn";

% PARAMETERS
int: M = 8;                          % a proximity factor
float: PI = 3.141;                   % the constant PI
0.0..infinity: R;                    % Each windfarm's rotor radius

0..infinity: w;                      % width of the windfarm
0..infinity: h;                      % height of the windfarm

set of int: WIDTH = 0..w;            % domains of windfarm's width
```

```
set of int: HEIGHT = 0..h;          % domains of windfarm's height

int: nzones;                        % number of infeasible zones
set of int: ZONES = 1..nzones;      % domains of infeasiable zones

array[ZONES] of WIDTH: zx;          % x coordinates of infeasiable zones
array[ZONES] of HEIGHT: zy;         % y coordinates of infeasiable zones
array[ZONES] of WIDTH: zwidth;      % width of each infeasible zones
array[ZONES] of HEIGHT: zheight;    % height of each infeasible zones

int: nturb;                         % number of wind turbines
set of int: TURB = 1..nturb;        % domain of wind turbines

float: min_dist::output = M * R;    % minimal distance between turbines

% SEARCH PARAMETERS
int: percentage = 90;               % relax_and_reconstruct percentage
float: initTemp = 20000;            % for simulated_annealing
float: coolingRate = 0.955;         % for simulated_annealing

% DECISION VARIABLES
array[TURB] of WIDTH: init_x;

array[TURB] of HEIGHT: init_y;

int: half_mdist = ceil(min_dist / 2) div 5;

array[TURB] of var -half_mdist..half_mdist: xvariant_div5;
array[TURB] of var -half_mdist..half_mdist: yvariant_div5;

array[TURB] of var int: xvariant = [ x * 5 | x in xvariant_div5];
array[TURB] of var int: yvariant = [ y * 5 | y in yvariant_div5];

% MAIN CONSTRAINTS

% making sure the placement of turbine is not out of the windfarm layout
constraint forall(t in TURB)(
  let {
    var int: x = init_x[t] + xvariant[t];
    var int: y = init_y[t] + yvariant[t];
  } in 0 <= x /\ x <= w /\ 0 <= y /\ y <= h);

% each turbines must be in a feasible zone
constraint forall(t in TURB, z in ZONES)(
  let {
    var WIDTH: x    = init_x[t] + xvariant[t],
    var HEIGHT: y   = init_y[t] + yvariant[t],
    WIDTH: x1       = zx[z],
    HEIGHT: y1      = zy[z],
    WIDTH: x2       = zx[z] + zwidth[z] - 1,
    HEIGHT: y2      = zy[z] + zheight[z] - 1
  } in x1 > x \/ x >= x2 \/ y1 > y \/ y >= y2);

% maintaining the minimal distance between turbines
constraint forall(i, j in TURB where i < j)(
  let {
    var WIDTH: x_i    = init_x[i] + xvariant[i],
    var HEIGHT: y_i   = init_y[i] + yvariant[i],
    var WIDTH: x_j    = init_x[j] + xvariant[j],
    var HEIGHT: y_j   = init_y[j] + yvariant[j]
  } in pow(x_i - x_j, 2) + pow(y_i - y_j, 2) >= ceil(pow(M * R, 2)));

% OBJECTIVE

% This blackbox function is calculating the Windfarm output.
```

```
% The required input coordinates for the blackbox is [x_1, y_1, x_2, y_2,...]
function list of var int: windfarm_output_bb(list of var int: x)
    ::blackbox_exec("java -cp ./windWithTDAandCMAESandOPENWIND.jar directSpread.WindTurbineBlackBoxMain");
function var int: windfarm_output(list of var int: x) = let {
    array[int] of var int: coords::output = [
      if j == 1 then init_x[t] + xvariant[t]
      else init_y[t] + yvariant[t] endif
    | t in TURB, j in 1..2  ];
  } in windfarm_output_bb(x)[1];

var int: obj ::output = windfarm_output(coords);

predicate simulated_annealing(float: initTemp, float: coolingRate) =
  let {
    var 0.0..initTemp: temp::output;
    var int: new_objective::output;
    int: y = ln(uniform(0.0, 1.0));
  } in
  if status()=START then temp = initTemp
  else
    temp = last_val(temp)*(1-coolingRate) /\ % cool down
    new_objective = sol(obj) + ceil(y * temp) /\
    obj > new_objective
  endif;
constraint simulated_annealing(initTemp, coolingRate);

solve
  ::seq_search([
     int_search([ if j == 1 then xvariant_div5[i] else yvariant_div5[i] endif | i in TURB, j in 1..2 ], input_order, indomain_random),
     relax_and_reconstruct(xvariant_div5 ++ yvariant_div5, percentage)
   ])
% `maximize` is applied using simulated annealing.
satisfy;
```