ViolationLS: Constraint-Based Local Search in CP-SAT

Toby O. Davies[1], Frédéric Didier[1], and Laurent Perron[1]

1. Google Research

Abstract. We introduce ViolationLS, a Constraint-Based Local Search (CBLS) solver for arbitrary MiniZinc models based on the Feasibility Jump Mixed-Integer Programming Heuristic. ViolationLS uses a novel approach for finding multi-variable moves without requiring any "implicit" or "one-way" constraints traditionally used by CBLS solvers. It significantly outperforms other state-of-the-art CBLS solvers on the MiniZinc challenge benchmarks, both with and without multi-variable moves. We demonstrate that integrating ViolationLS into the parallel portfolio framework of the state-of-the-art CP-SAT Lazy Clause Generation solver improves performance on more than half of the instances in the MiniZinc challenge benchmarks suite. To our knowledge this is the first instance of such an integration of CBLS and Lazy Clause Generation inside the same solver.

1 Background

1.1 Constraint-Based Local Search

Constraint-Based Local Search (CBLS) [6] is an approach to local search that takes inspiration from Constraint Programming. In particular, both exploit the fact that many combinatorial sub-structures occur in many different problems to build models using reusable "constraints" based on these structures.

A CBLS model consists of an objective function, a set of variables, and a set of constraints. Constraints in CBLS each define a "violation" function given an assignment to the variables that takes the value 0 if the constraint is satisfied, and a positive value otherwise. A CBLS solver tries to find an assignment such that all constraints have a violation of 0, while secondarily minimising the objective.

CBLS solvers typically either require explicit neighbourhoods to be provided by the modeller, or provide "implicit constraints"[2] that act as neighbourhoods and define moves and initialisation logic that guarantee that the implicit constraint is initially satisfied and remains satisfied after each move. To create an overall neighbourhood from a set of implicit constraints, each variable may participate in at most one implicit constraint, and any variable in an implicit constraint must only be initialised and modified by moves generated by that constraint.

Additionally CBLS solvers usually use "one way constraints"[2], these must form a Directed Acyclic Graph of functional definitions, used to reduce the number of decision variables. These allow other soft constraints and the objective to depend on the auxiliary variables while reducing the search space.

In order to support arbitrary models, CBLS solvers must define a "soft" version of implicit constraints, as the structure of the problem may contain variables that participate in multiple implicit constraints. Solvers must also provide a soft version of all one-way constraints as arbitrary constraint graphs may contain cycles that must be broken using soft constraints.

1.2 CP-SAT

The CP-SAT solver (also called CP-SAT-LP) [10] is a state-of-the-art Lazy Clause Generation (LCG)-based solver [11]. CP-SAT has won multiple medals in the MiniZinc Challenge [12] every year since it replaced the "CP" solver as the OR-tools entry in 2013.

In its multi-threaded configuration, CP-SAT implements an information-sharing portfolio of diverse subsolvers. There is a distinction between 3 different kinds of subsolvers: "full" subsolvers; "first solution" subsolvers; and "incomplete" subsolvers.

"Full" subsolvers exist for the full duration of the solve. Most "full" workers implement depth-first searches using various branching and restart heuristics, using a Lazy Clause Generation propagation and explanation engine to prune variable domains after each decision and avoid repeated search when back-jumping.

"First solution" workers terminate as soon as the first solution is found, and are replaced with "incomplete" workers. Most "first solution" workers are variants of one of the "full" workers, with additional randomness. Most "incomplete" workers perform Large Neighbourhood Search (LNS), using various neighbourhoods. These neighbourhoods fix part of the solution to be the same as a base solution sampled from a shared solution pool, and are then solved by creating a new CP-SAT instance, presolving the smaller problem, and solving the problem using a single-thread LCG worker.

CP-SAT expects there to be more incomplete solvers than threads to run them, so whenever a thread becomes available, an incomplete subsolver is chosen to generate and run a short-lived "task". The incomplete subsolvers are chosen in a round-robin pattern.

Workers share information between one another by periodically synchronising with various thread-safe shared state classes. Two such classes are relevant to this paper. First, the Shared Solution Manager, which maintains a small pool of the best 3 solutions seen, and the upper and lower bounds on the objective. Second, the Shared Bounds Manager, which allows full workers to share root-level reductions of variable domains with other workers.

Another important component of CP-SAT is its presolver which runs before any subsolvers start, modifying the model in various ways. A full description of these transformations is beyond the scope of this paper, but they are summarised here. The presolver replaces some complex constraints with simpler ones (often

a set of linear constraints); adds symmetry breaking constraints; and substitutes equivalent variables. The CBLS solvers we have implemented and described in this paper all use CP-SAT's presolver.

The presolver may act similarly to limited "one-way" constraints in this final regard as many CP-SAT constraints accept affine transformations of the form $ax+b$ in place of a variable, where a and b are constants and x is a variable. This allows some variables to be removed, but is still far more restrictive than the general functional dependencies enabled by one-way constraints in other CBLS solvers, which could for example remove the variable y in $y = x^2$ from the set of decision variables.

2 Violation Functions

Each constraint in a CBLS solver defines a violation that is 0 when the constraint is satisfied and strictly positive otherwise. Our solver defines a decomposition or violation function for every type of constraint supported natively by CP-SAT.

Constraints in our solver have a simple API, they need to be able to: compute their initial violation given a new assignment; compute the violation delta when a single variable changes value; and update their internal state when a single variable changes value. We use the notation $\delta_G(v, j)$ to denote the function returning a sparse vector containing the change in violation of each constraint in the constraint graph G when variable v changes its value to j.

These three operations on linear constraints are identical to those in the paper introducing Feasibility Jump [8] (modulo half-reification, explained below [3]).

In the implementation tested in this paper all non-linear constraints compute their violation delta by computing their violation, temporarily changing the value of the variable in question, recomputing their violation from scratch, then resetting the variable and returning the difference between the two violations. We expect significant improvements in performance on larger scheduling and routing instances by improving this aspect of the implementation as the computation of violation for these types of constraints in particular is expensive.

To maximise the benefit of making linear constraint violation efficient and incremental to compute, many constraints in our implementation are partially or fully decomposed into a set of linear constraints instead of or in addition to defining a general constraint. Our implementation generally fully linearises a constraint whenever a complete encoding of the constraint requires no new variables and at-most a linear number of additional constraints. Additionally it will partially linearise the violation function whenever doing so requires no new variables, at most a linear number of new constraints, and the linear constraints form a sparser constraint graph than the global would.

CP-SAT has native support for half-reified versions of many constraints. Half-reified versions of the constraints defined below are the same, except have violation defined to be 0 if the constraint is disabled in the current assignment.

Linear constraints define violation as in the original Feasibility Jump paper: the absolute value of the difference between the left and right hand side of the constraint if violated, or 0 if the constraint is satisfied.

Boolean constraints are transformed into equivalent linear expressions, except XOR which is defined to have violation 0 if satisfied, and 1 otherwise.

Min (and max) constraints where $t = min(xs)$ ($t = max(xs)$) define violation by decomposing into a $t \leq x$ ($t \geq x$) constraints for each variable $x \in xs$. Thus the linear submodel enforces that t is a lower (upper) bound on the other arguments, and we also add a non-linear constraint $t \geq min(xs)$ ($t \leq max(cs)$). This definition creates a sparser graph in the linear submodel, so fewer variables will be considered when repairing its violation if few variables are close to t.

Circuit constraints in CP-SAT are slightly unusual compared to other CP solvers in that they are defined in terms of a binary variable for each arc, rather than a "next" array of integers. The violation of this constraint is decomposed into the flow-conservation linear constraints (in-arcs and out-arcs for each node must sum to 1), plus a subcircuit-elimination constraint. The subcircuit-elimination constraint considers the graph containing only arcs whose controlling literals are 1. Its violation is defined to be 0 if all Strongly-Connected Components (SCCs) are singletons (i.e. contain only a single node). Otherwise, the violation is the number of non-singleton strongly connected components (SCCs), minus 1 (as a valid circuit contains at most 1 non-singleton SCC), plus the number of non-singleton SCCs that do not have an active arc from any node in that SCC to any other node in a different non-singleton SCC, minus 1. An important property of this definition is that removing arcs from a valid tour leaves the subcircuit-elimination constraint satisfied (though flow constraints will become violated), only introducing a second cycle causes it to become violated.

Route constraints are nearly identical to circuit constraints, but the subcircuit-elimination constraint also adds 1 to its violation if the size of the SCC containing the depot is one, unless there are no non-unit-size SCCs at all. Also the in and out arcs at the depot must merely equal one another, rather than equalling exactly 1.

Cumulative and Disjunctive constraints violations are defined to be the total area of above the capacity of the cumulative (above 1 for Disjunctive constraints).

2D Packing constraints are implemented using a pairwise decomposition, and the violation of each pairwise constraint is the area of overlap of the two rectangles.

3 Generalised Feasibility Jump

Our two closely-related CBLS solvers are both based on extending the Feasibility Jump heuristic [8] designed for Mixed-Integer Programming to work with general

constraints. Feasibility Jump is not described as nor compared to CBLS, we believe only because CBLS is not well-known in the mathematical programming community. For all intents and purposes Feasibility Jump is a type of CBLS solver restricted to Linear Constraints.

Feasibility Jump is novel compared to existing CBLS solvers in that it does not require constraints to define separate implicit, one-way, and soft versions. Nor for the solver to have to choose which of these implementations each particular instance of a constraint should use. In Feasibility Jump, all constraints are soft. Using only soft constraints has been shown to have a significant negative effect on the performance of another CBLS solver [2], but in section 6 we show that our Generalised Feasibility Jump outperforms other CBLS solvers by a significant margin.

Generalised Feasibility Jump and ViolationLS both maintain very similar state, to simplify presentation we define a state tuple that we use extensively below and in our algorithm listings.

$$S = \langle G, X, W, V, Q, J \rangle$$

Where G is a constraint graph containing variables and constraint, G_c denotes the set of variables in constraint c, and G_v is the set of constraints containing variable v. Each variable v in G has a finite domain of possible values it may take, denoted $D(v)$. X is a valuation of the variables in G, where $X[v] \in D(v)$. W is a vector with a strictly positive weight for each constraint c. V is the subset of constraints in G that are violated in the assignment X. Q the set of variables that may be valid moves, and J is a "jump table" containing either a "jump value" and "score" (explained below) or nil for each variable in G.

Feasibility Jump repeatedly selects a variable v and sets its value to the value $j \in D(v)$ (excluding the current value of v) that minimises the weighted sum of violations over all constraints. This value j is called the variable's jump value, and is cached in J to maximise incrementality. The reduction in weighted violation from assigning a variable to its jump value is the variable's score, $-W \cdot \delta_G(v, j)$. Weights (W) are initialised to 1, and the weights of violated constraints are increased by 1 whenever no variable has positive score. We also implement a variant where weights are decayed by $\rho \in (0, 1)$ before weights are updated.

Note that our implementation of Generalised Feasibility Jump differs from that described in the original paper [8] in 2 key ways when applied to a model containing only linear constraints. First, we re-compute the jump values and scores of a variable v whenever any variable that shares a constraint with v changes, in the original, only v's jump value would be recomputed, and others would be recomputed at fixed point. Second, we apply the best jump of 5 with positive score, whereas the original algorithm selects 25 random variables that participate in a violated constraint, and if none have negative score then weights of all violated constraints are increased.

To compensate for some of the extra work due to the first difference above, we use the fact the jump value cannot change for binary variables to only consider the constraints that have been updated when computing the score, using the

same logic used for all variables in the original algorithm. For other variables, we compute jumps and scores lazily, whenever a variable is updated in Algorithm 1 we invalidate the jump for adjacent variables, and recompute as needed in Algorithm 2.

Feasibility Jump exploits the structure of linear constraints to compute a subset of values for a variable that must include one with minimal score. This structure does not apply to general constraints, so in our generalisation we instead assume a slightly weaker property: that each violation function is convex when we change a single variable within its domain. This allows us to use a convex minimisation algorithm like ternary search to compute a variable's jump value and score, since a positive-weighted sum of convex functions is convex.

Note that it does not actually impact the correctness of any of our algorithms if violation functions are not in fact convex. The only difference is that the search is more restricted, as we may incorrectly conclude that a variable has no good jump values.

Our implementation maintains a set of variables to scan Q, initialised with all variables participating in any violated constraint. The main loop repeatedly samples from that set, setting variables with positive score to their jump value, then adding any variable in any constraint touched by the changed variable into the set to scan. When no more improving jumps can be found, it increases the weights of all violated constraints by 1, re-initialises Q and repeats. This is a type of Guided Local Search (GLS) [13], shown in Algorithm 3.

Our generalisation first finds a solution to the linear subset of constraints, by performing the GLS using only the linear submodel, then applies the same algorithm to the full model, starting from the solution found in the first phase. This allows us to minimize the impact of relatively inefficient global constraint evaluation. This simple algorithm alone finds a surprising number of solutions to problems in the MiniZinc Challenge instances, as shown in Section 6.

4 Novelty Jump

Local-search solvers' moves usually affect more than one variable at a time in order to maintain desirable invariants (notably, maintaining the satisfaction of all implicit constraints in CBLS). Many algorithms to generate such moves search for a sequences of smaller component changes, where later components repair invariants broken earlier in the sequence. The Lin-Kerighan TSP heuristic [5] can be viewed as using a neighbourhood that chains dependent sequences of changes. Similarly, the "Ejection Chain" family of heuristics explicitly attempt to find sequences of changes [4].

To take advantage of the incremental updates used by Feasibility Jump, we search for such sequences using a depth-first search, using modified constraint weights to find values likely to allow chains to continue. Our approach to modifying the weights used in the search is inspired by the concept of Novelty used in AI planning [7], in particular Novelty Jump is very similar to a sequence of calls to a depth-first variant of $IW(1)$.

Algorithm 1: UpdateVar(S, v)

Input:
S: A state tuple $\langle G, X, W, V, Q, J \rangle$
v: The variable to update
Output: S is modified

```
1  j, s ← J[v]
2  x_0 ← X[v]
3  X[v] ← j
4  foreach c ∈ G_v do
5      Update the internal state of c with the new value of v
6      if v(X, c) > 0 then
7          V ← V ∪ {c}
8      else
9          V ← V \ {c}
10 foreach c ∈ G_v do
11     foreach v' ∈ G_c \ {v} do
12         if v' is binary and c is linear then
13             Update the score of v' in J as in [8]
14         else
15             J[v'] ← nil
16         if V ∩ G_v' ≠ ∅ then
17             Q ← Q ∪ {v'}
```

$IW(1)$ performs a structured exploration of the state-space, requiring each transition to set at least one state variable to a "novel" value not seen since the current best state. Analogously in Novelty Jump, we use modified weights to make it significantly cheaper to increase the violation of any constraint that has not been violated since the best state seen so far.

These weights prioritise jump values that fix constraints that are initially broken, in the hope that constraints that have not yet been broken can be fixed later, by further changes. If backtracking is necessary, subsequent children will prefer to avoid breaking the same constraints as were broken in prior exploration, making the search more structured than a purely random exploration.

Novelty Jump initialises the weights of currently violated constraints to the same value as used in the Guided Local Search metaheuristic, and sets non-violated constraint weights to ϵ times the GLS weight ($\epsilon = 2^{-10}$ in our implementation). Whenever we take a move, the weight of any newly violated constraints are set to the GLS weight. Jump values used in Novelty Jump are computed using these "novelty weights", to avoid confusion with scores using the original weights we will use "score" to refer to the score with respect to the original weights unless otherwise stated, and "novelty score" of a jump value to be the score computed with the two weights.

Algorithm 2: ApplyJump(S)

Input: S: A state tuple $\langle G, X, W, V, Q, J \rangle$
Output: S is modified, returns False if no move could be found.

$v, j, s \leftarrow \text{nil}, \text{nil}, 0$
$n \leftarrow 0$
`// Sample 5 vars from Q and keep the one with the best score`
while *Q is not empty* **do**
  $v' \leftarrow$ a random variable from Q
  if $J[v'] = nil$ **then**
    $j' \leftarrow \underset{j' \in D(v')}{\text{ConvexArgMin}}(W \cdot \delta_G(v', j'))$
    $J[v'] \leftarrow j', -W \cdot \delta_G(v', j')$
  $j', s' \leftarrow J[v']$
  if $s' <= 0$ **then**
    $Q \leftarrow Q \setminus \{v\}$
    continue
  if $s' > s$ **then**
    $v, j, s \leftarrow v', J[v'], s'$
  $n \leftarrow n + 1$
  if $n \geq 5$ **then**
    break
if $v = nil$ **then return** False
UpdateVar(S,v)
return True

During depth-first search if we can't find any way to extend the compound move given the filtering mechanisms described below, we backtrack. When backtracking we revert the last assignment, but not the changes to any novelty weights, which we update only when we fail to find any move at the root.

Note that whenever a constraint is violated, its novelty weight is equal to the original weight. Thus reductions in violation of currently violated constraints are responsible for all positive terms in the computation of a variable's score. Consequently the novelty score of a jump cannot be lower than its score, as only the negative terms can have smaller weights. So long as the depth-first search always explores at least one positive score move at the root (if one exists), any state that is a local minimum for Novelty Jump must also be a local minimum for Feasibility Jump.

Our preliminary investigation showed that exploring all moves with negative novelty score explores too much, taking too long to detect a local minimum and increase the GLS weights. To combat this, Novelty Jump limits exploration in three ways. First, we do not change the same variable twice in the same compound move.

Second we limit the number of backtracks along any path, once this is exceeded we backtrack until we reach a node with remaining backtrack budget

Algorithm 3: GLS(G, X, W, M, ρ)

Input:
G: A bipartite graph of variables and constraints,
X: An assignment from variables to values
W: A weight per constraint
M: A "move" function that modifies X or returns False
ρ: A discount factor for weights

Output:
X, and W are modified

$V \leftarrow \{c \mid c(X) > 0, \quad \forall c \in \text{Constraints}(G)\}$
$Q \leftarrow \bigcup_{c \in V} G_c$
$J[v] \leftarrow \text{nil} \qquad \forall v \in Variables(G)$
while *effort limit is not reached* **do**
  if $M(\langle G, X, W, V, Q, J\rangle) = False$ **then**
    if $V = \emptyset$ **then**
      return `FEASIBLE`
    $W \leftarrow \rho W$
    foreach $c \in V$ **do**
      $W[c] \leftarrow W[c] + 1$
      foreach $v \in G_c$ **do**
        $S[v] \leftarrow \text{nil}$
return `UNSOLVED`

$b > 0$. If we backtrack to the root, the limit is increased, to a maximum of 2. If the search fails with its maximum limit, we increment weights like in feasibility jump, re-initialise the novelty weights and continue.

Third, we track the cumulative score of individual moves along the path (s_m), and for each node on the stack we track the best score of any immediate child move that has previously been explored (s_c, initially 0). To be considered in the search, a move must either have sufficient novelty that the novelty score plus s_m is positive, or the score is greater than the s_c value of the current search node. This is defined by the filter predicate F, used in Algorithm 5:

$$F(G, W, W', s_m, s_c, v, j) \leftrightarrow (s_m - W' \cdot \delta_G(v, j) > 0) \vee (-W \cdot \delta_G(v, j) > s_c)$$

This definition allows 2 categories of move to be explored. The first criterion allows chains of values to be swapped between variables. The second allows multiple small changes to combine to repair a large change that was allowed by the first criterion.

For example if the first criterion allowed search to explore $x = 1$ with a score of -1, the constraint $x \to y + z = 0$ would require at least 2 moves to repair if $y = z = 1$, but these two moves would both have positive score so either jump would be allowed at the first node, and the other would be allowed as a child

of that node (assuming its score was still positive). However if the search later backtracked over these moves, the sequence would *not* be explored again, as the score would not be higher than the best score explored previously (s_c).

The second criterion alone would obviously not be sufficient to perform exploration as this would only explore positive score moves which would be immediately committed, making this algorithm nearly equivalent to Feasibility Jump. However it does guarantee that Novelty Jump will always explore at least one positive score move (if one exists) at the root node of the tree, ensuring its local minima are at least as good as Feasibility Jump.

Another property of this filtering mechanism to note is that no two moves causing the same violation delta can be explored as children of the same node. After a move $X[v] = j$ has been explored W' is updated so that $W \cdot \delta_G(v, j) = W' \cdot \delta_G(v, j)$, so the first criterion cannot apply to the new move if $\delta_G(v', j') = \delta_G(v, j)$, and the new move must also have the same score as the previously explored move, so cannot have a better score than all previously explored moves, thus the second criterion cannot apply.

If, at any point in the search, the sum of scores of single-variable moves on the stack is positive, we commit the compound move, and reset the per-node backtrack limit to 0. This resetting means that the algorithm limits exploration further when moves are easy to find.

We provide pseudo-code for this algorithm in Algorithm 4, and the recursive helper function shown in Algorithm 5. Our actual implementation is iterative, with an explicit stack of moves instead of using the call stack, which is somewhat more efficient, but more cumbersome to present.

Algorithm 4: ApplyNoveltyJump(S)

Input: S: A state tuple $\langle G, X, W, V, Q, J \rangle$
Output: S and W' are modified, returns False if no move could be found.

```
b ← 0
while b ≤ 2 do
    W' ← ϵW
    foreach c ∈ V do
        W'[c] ← W[c]
    while NoveltyJumpSearch(S, W', 0, b, ∅) do
        if V = ∅ then return True
        b ← 0
    b ← b + 1
return False
```

Algorithm 5: NoveltyJumpSearch(S, W', s_m, b, T)

Input:
S: A state tuple $\langle G, X, W, V, Q, J\rangle$
W': A "novelty" weight per constraint
s_m: The score of the partial move so far
b: The remaining backtrack limit for the search
T: The set of variables on the stack, not eligible for moves.

Output: S, and W' are modified, returns False if no move can be applied.

if $b < 0$ **then return** False
$s_c \leftarrow 0$
while $Q \supset T$ **do**
  while $b \geq 0$ **do**
    $v \leftarrow$ Best of 3 sampled vars in $Q \setminus T$ satisfying $F(G, W, W', s_m, s_c, v, j)$
    if $v = nil$ **then return** False
    $Q \leftarrow Q \setminus \{v\}$
    $j \leftarrow \underset{j' \in D(v')}{\text{ConvexArgMin}}(W' \cdot \delta_G(v', j'))$
    $s \leftarrow -W \cdot \delta_G(v, j)$
    $s_c \leftarrow \min(s, s_c)$
    $x_0 \leftarrow X[v]$
    UpdateVar(S,v)
    foreach $c \in V \cap G_v$ **do**
      if $W'[c] \neq W[c]$ **then**
        $W'[c] \leftarrow W[c]$
        $Q \leftarrow Q \cup G_c$
    if $s_m + s > 0$ **then return** True
    if *NoveltyJumpSearch*$(S, W', s_m + s, b, T \cup \{v\})$ **then**
      return True
    $J[v] \leftarrow x_0, -s$
    $UpdateVar(S, v)$
    $b \leftarrow b - 1$
return False

5 ViolationLS

Our overall algorithm, ViolationLS (Algorithm 6) picks either Feasibility Jump or Novelty Jump (in our experiments, with 50% probability each), and runs it for a "batch". The stopping criteria for a batch is complicated and is not particularly scientifically interesting, as choosing a good value mostly depends on the other solvers in the "incomplete" and "first solution" worker pools rather than any property of the algorithm itself.

At the start of each batch, if there is a new best solution in the solution pool or we have seen no solutions for 100 batches, we choose the algorithm again. Otherwise we continue with the same algorithm as last batch, reusing the same weights from the end of the last batch.

In our experiments below, where we report the performance of "Feasibility Jump", we use a 100% probability of choosing Feasibility Jump in this algorithm.

Algorithm 6: ViolationLS(G)

```
   Input: G: A bipartite graph of variables and constraints
 1 W[c] ← 1      ∀c ∈ Constraints(G)
 2 Initialize X to the value in each variable's domain closest to 0
 3 A ← RandomChoice({FJ, NJ})
 4 while The problem is not solved do
 5     if If a new best solution S is available in the shared pool then
 6         X ← S
 7         Tighten the RHS of the objective constraint.
 8         W[c] ← 1      ∀c ∈ Constraints(G)
 9         ρ ← RandomChoice({0.95, 1.0})
10         A ← RandomChoice({FJ, NJ})
11     if No new solutions found or imported for 100 iterations then
12         Perturb X, randomising each variable's value with probability 0.1
13         W[c] ← 1      ∀c ∈ Constraints(G)
14         ρ ← RandomChoice({0.95, 1.0})
15         A ← RandomChoice({FJ, NJ})
16     if A = FJ then
17         G' ← the linear submodel of G
18         if GLS(G', X, W, ApplyJump, ρ) = FEASIBLE then
19             if GLS(G, X, W, ApplyJump, ρ) = FEASIBLE then
20                 Add X to the Shared Solution Pool
21     else
22         if GLS(G, X, W, ApplyNoveltyJump, ρ) = FEASIBLE then
23             Add X to the Shared Solution Pool
24     Yield to allow other incomplete solvers to run
```

6 Experimental Results

We compare our Feasibility Jump and ViolationLS purely local search solvers against 2 local search solvers that have previously won medals in the local search category of the MiniZinc challenge: Yuck [9], and fzn-oscar-cbls [2]. In addition we show the impact of adding our CBLS solvers to CP-SAT's "first solution" and "incomplete" subsolver pool by comparing several configurations of CP-SAT with and without these algorithms. Each solver was run 5 times with 5 different random seeds and a 5 minute time limit, using 8 worker threads, as CP-SAT is designed for multi-threaded solving, and 8 cores is the minimum number of threads that CP-SAT recommends. Note that fzn-oscar-cbls is not

multi-threaded, so can only take advantage of one core, but is included for completeness as it is better-described in the literature than Yuck. The experiments were performed on 64-cores machines (n2d-standard-64 Google Compute Engine instances), each running 8 solves concurrently.

In Table 1, we show the mean number of points scored by each solver, averaged across 5 different random seeds for each solver and instance. A solver scores a point for every other solver and instance where it solved that instance better than the other solver. For the "Complete" column, the better solver either closed the problem faster; found a better solution; or found the same quality solution faster (if neither solver closed the problem). For the "Incomplete" column, the better solver either found a better solution, or found the same quality solution faster. The percentage columns show what percentage of the maximum possible score each solver achieved.

Solver	Complete		Incomplete	
	Points	%	Points	%
CP-SAT+ViolationLS	**5841.04**	**80.06%**	**5630.04**	**77.17%**
CP-SAT+Feasibility Jump	5812.60	79.67%	5583.20	76.52%
CP-SAT (No LS)	5643.76	77.35%	5406.68	74.10%
ViolationLS	2632.60	36.08%	3003.92	41.17%
Feasibility Jump	1788.08	24.51%	2050.24	28.10%
Yuck	1180.92	16.19%	1200.08	16.45%
fzn-oscar-cbls	487.32	6.68%	511.76	7.01%

Table 1. MiniZinc Challenge Scores, averaged over 5 random seeds

Figure 1 shows that while ViolationLS helps substantially more often than it hinders time to prove optimality, the magnitude of improvement is often small, and it can occasionally cause substantial slowdown. We suspect two possible causes. First, some global constraints are slow to evaluate, this may cause CP-SAT to allocate disproportionate CPU-time to ViolationLS that would be better spent on LNS. Improving the incrementality of global constraints may mitigate this, and is likely to lead to better performance generally, there is significant room for improvement in our implementation.

Second, ViolationLS finds many (quite similar) solutions in quick succession. We suspect the solution pool becomes full of similar solutions before LNS workers have the opportunity to explore neighbourhoods around earlier solutions. However quickly finding nearby solutions is the goal of ViolationLS, so mitigating this would require changes to encourage diversity in the solution pool.

Table 2 shows the average number of times that each of the pure local-search solvers tested found a better solution faster than each other local-search solver. Similarly, Table 3 shows how often adding the two local search algorithms introduced in this paper outperformed the other CP-SAT configurations.

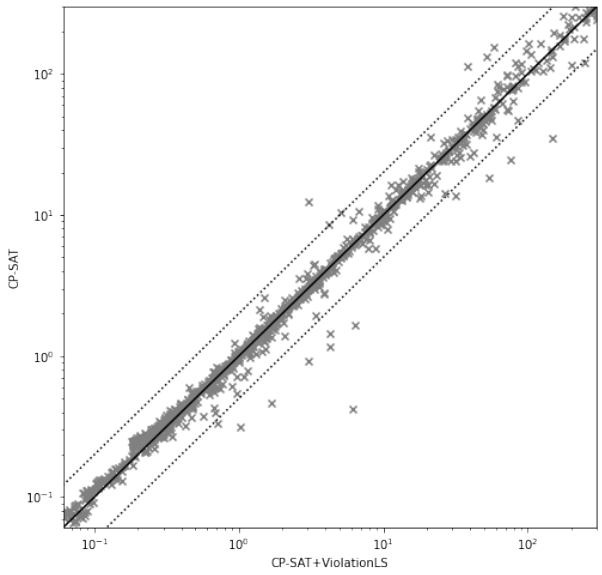

Fig. 1. Shifted geometric mean time to optimality for CP-SAT with and without ViolationLS on each the MiniZinc challenge benchmark instances, averaged over 5 seeds, shifted by 10s. The 609 instances (50.1%) above the diagonal are faster with ViolationLS, and 397 (32.6%) below are slower, 209 (17.2%) were consistently unsolved by either solver within 5 minutes. Dotted lines show 2x increase or decrease in solve time.

7 Conclusions and further work

Both our generalisation of Feasibility Jump and ViolationLS improve substantially on the current state-of-the-art local search solver for general models expressed in MiniZinc. ViolationLS finds better solutions faster compared to "Yuck" more than 3x as frequently as Yuck outperformed ViolationLS.

Including these algorithms in the incomplete subsolver framework inside the state-of-the-art CP-SAT improves performance in 20% more instances than it degrades it. However, it does slow solve time significantly on some instances. Mitigating this will make using ViolationLS more effective as a primal heuristic, but further investigation into the causes and possible solutions is required. We suspect improving the incrementality of global constraint violation computation to improve this significantly.

Global constraints can also make the constraint graph more densely connected, causing ViolationLS to needlessly recompute jumps for variables that cannot have positive score because they do not actually participate in the violation detected by the global constraint. For example if 2 intervals overlap in an otherwise satisfied Disjunctive constraint, there is no need to consider changing the value of any variables in any other intervals (unless these variables partic-

Solver	ViolationLS	Feasibility Jump	Yuck	fzn-oscar-cbls
ViolationLS	-	**650.6**	**652.28**	**734.64**
Feasibility Jump	182.00	-	523.84	617.24
Yuck	**209.52**	268.96	-	435.72
fzn-oscar-cbls	101.96	135.36	147.6	-

Table 2. Pairwise incomplete MiniZinc Challenge scores. Each cell shows the average number of instances where the solver in that row was better than the other solver in that column, using the "incomplete" definition of better. Averaged over 5 random seeds. The highest score in each column is highlighted.

CP-SAT+	ViolationLS	Feasibility Jump	Baseline
ViolationLS	-	**609.20**	**659.04**
Feasibility Jump	**588.36**	-	651.80
Baseline	538.32	545.48	-

Table 3. Pairwise complete MiniZinc Challenge scores for CP-SAT variants. Each cell shows the average number of instances where CP-SAT plus the heuristic in that row was better than CP-SAT plus the heuristic in that column, using the "complete" definition of better. Averaged over 5 random seeds. The highest score in each column is highlighted.

ipate in other violations). It may be possible to extend the constraint API to allow ViolationLS to cheaply skip computing jumps for such variables, or to skip enqueing them entirely.

Another possible avenue for improvement would be to revisit the definition of the violations of global constraints. The theoretical basis of jump values relies on the convexity of violation functions, however several of our constraint definitions are non-convex (notably scheduling constraints). While this has no impact on correctness, there may be opportunities for improved or more consistent performance if the violation definitions of more constraints were convex.

The parameters in ViolationLS were chosen based only on very limited preliminary experiments. There will almost certainly be a benefit to automatically tuning this during search based on which configurations find solutions faster on a specific instance.

Finally, the Novelty Jump heuristic was inspired by treating the search for an improving neighbour as a domain-independent planning problem. There may be other promising avenues of research based on adapting other planning algorithms to create other effective, problem-independent CBLS neighbourhoods.

8 Related Work

Previous work [1] has explored adding local-search to CP-based MiniZinc solvers. This approach requires annotations to be written by the modeller to define the neighbourhood, so cannot be used with an arbitrary MiniZinc model.

References

1. Björdal, G., Flener, P., Pearson, J., Stuckey, P.J., Tack, G.: Declarative local-search neighbourhoods in minizinc. In: 2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI). pp. 98–105. IEEE (2018)
2. Björdal, G., Monette, J.N., Flener, P., Pearson, J.: A constraint-based local search backend for MiniZinc. Constraints **20**, 325–345 (2015)
3. Feydy, T., Somogyi, Z., Stuckey, P.J.: Half reification and flattening. In: Lee, J. (ed.) Principles and Practice of Constraint Programming – CP 2011. pp. 286–301. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
4. Glover, F., Rego, C.: Ejection chain and filter-and-fan methods in combinatorial optimization. 4OR **4**, 263–296 (2006)
5. Helsgaun, K.: An effective implementation of the Lin–Kernighan traveling salesman heuristic. European journal of operational research **126**(1), 106–130 (2000)
6. Hentenryck, P.V., Michel, L.: Constraint-based local search. The MIT press (2009)
7. Lipovetzky, N., Geffner, H.: Width and serialization of classical planning problems. In: European Conference on Artificial Intelligence (ECAI), pp. 540–545. IOS Press (2012)
8. Luteberget, B., Sartor, G.: Feasibility Jump: an LP-free Lagrangian MIP heuristic. Mathematical Programming Computation **15**(2), 365–388 (2023)
9. Marte, M.: Yuck. https://github.com/informarte/yuck, accessed: 2023-11-01
10. Perron, L., Didier, F., Gay, S.: The CP-SAT-LP solver. In: Yap, R.H.C. (ed.) 29th International Conference on Principles and Practice of Constraint Programming (CP). Dagstuhl, Germany (2023)
11. Stuckey, P.J.: Lazy clause generation: Combining the power of sat and cp (and mip?) solving. In: Lodi, A., Milano, M., Toth, P. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 5–9. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
12. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The minizinc challenge 2008–2013. AI Magazine **35**(2), 55–60 (Jun 2014)
13. Voudouris, C., Tsang, E.P., Alsheddy, A.: Guided local search. In: Handbook of metaheuristics, pp. 321–361. Springer (2010)