Solving Satisfaction Problems using Large-Neighbourhood Search

Gustav Björdal✉[1][0000−0002−8032−5774], Pierre Flener[1][0000−0001−8730−4098], Justin Pearson[1][0000−0002−0084−8891], Peter J. Stuckey[2][0000−0003−2186−0459], and Guido Tack[2][0000−0003−3357−6498]

[1] Uppsala University, Department of Information Technology, Uppsala, Sweden
{Gustav.Bjordal,Pierre.Flener,Justin.Pearson}@it.uu.se

[2] Monash University, Faculty of Information Technology, Melbourne, Australia
{Peter.Stuckey,Guido.Tack}@monash.edu

Abstract Large-neighbourhood search (LNS) improves an initial solution, hence it is not directly applicable to satisfaction problems. In order to use LNS in a constraint programming (CP) framework to solve satisfaction problems, we usually soften some hard-to-satisfy constraints by replacing them with penalty-function constraints. LNS is then used to reduce their penalty to zero, thus satisfying the original problem. However, this can give poor performance as the penalties rarely cause propagation and therefore do not drive each CP search, and by extension the LNS search, towards satisfying the replaced constraints until very late. Our key observation is that entirely replacing a constraint is often overkill, as the propagator for the replaced constraint could have performed *some* propagation without causing backtracking. We propose the notion of a *non-failing propagator*, which is subsumed just before causing a backtrack. We show that, by only making a few changes to an existing CP solver, any propagator can be made non-failing without modifying its code. Experimental evaluation shows that non-failing propagators, when used in conjunction with penalties, can greatly improve LNS performance compared to just having penalties. This allows us to leverage the power of the many sophisticated propagators that already exist in CP solvers, in order to use LNS for solving hard satisfaction problems and for finding initial solutions to hard-to-satisfy optimisation problems.

1 Introduction

Large-neighbourhood search (LNS) [19] is a popular method for local search. It often uses constraint programming (CP) for neighbourhood exploration and has been successfully applied to a vast variety of optimisation problems. LNS starts from a feasible assignment and explores a large neighbourhood of similar assignments by forcing most of the variables to take their current values while performing CP search in order to find better values for the other variables. This process is repeated in the hope of finding a good enough feasible assignment. *However, as LNS requires a feasible assignment to start from, LNS cannot be* *directly applied to satisfaction problems*, because the initial feasible assignment would be an acceptable solution. Furthermore, an optimisation problem that is hard to satisfy, that is where finding a feasible assignment is difficult, cannot be solved by LNS until an initial feasible assignment is obtained. We emphasise that finding an initial feasible assignment is in fact a satisfaction problem. Therefore, *if we can efficiently solve satisfaction problems using LNS, then we can also solve hard-to-satisfy optimisation problems using* (*two rounds of*) *LNS.*

One approach to using LNS for satisfaction problems is to (manually) identify and *soften* the constraints that make the problem hard to satisfy. Traditionally, soft constraints for CP have been investigated mostly for *over-constrained problems* [10], that is for problems where not all the constraints can be satisfied. There is little previous work on softening constraints in order to enable LNS (see Section 6). Still, there are generic methods for softening a constraint, such as replacing it by using a penalty function and minimising the penalty via the objective function (see Section 2.1 for examples). However, these methods tend to give poor performance in practice, as they significantly increase the size of the CP search space and provide little propagation to drive the CP search towards a zero-penalty solution (as we show in Section 3).

In this paper, we argue that entirely replacing constraints by using penalty functions in order to enable LNS is overkill, because it means that we lose *all* their propagation, including the propagation that would *not* have caused failure but would have avoided unnecessarily high penalties.

Based on this observation, we propose the notion of a *non-failing propagator*: the inconsistent domain values of a variable are only pruned as long as doing so does *not* cause a failure. As soon as propagation would cause failure, the propagator is disabled. This prevents the propagator from directly causing backtracking and, when used in conjunction with a penalty function, helps the CP search to quickly reach low-penalty solutions.

After giving some definitions on soft constraints and LNS (Section 2) and a motivating example (Section 3), our contributions and impact are as follows:

– the concept and theory of non-failing propagators (Sections 4.1 and 4.3);
– a recipe for modifying a CP solver so that any propagator can automatically become non-failing, without modifying its code (Section 4.2);
– an empirical evaluation of the often drastic effect of non-failing propagators on solving satisfaction problems by LNS, as well as of their use when solving hard-to-satisfy optimisation problems by LNS (Section 5).

We discuss related work (Section 6) and future work and conclude (Section 7).

2 Definitions

A *constraint satisfaction problem* is a triple $\langle \mathcal{X}, D, \mathcal{C} \rangle$ where $\mathcal{X}$ is a set of variables, the function D maps each variable x of $\mathcal{X}$ to a finite set $D(x)$, called the *current domain* of x, and $\mathcal{C}$ is a set of constraints. An *assignment* is a mapping σ where $\sigma(x) \in D(x)$ for each x of $\mathcal{X}$. A *feasible assignment* is an assignment that satisfies all the constraints in $\mathcal{C}$. A *constrained optimisation problem* $\langle \mathcal{X}, D, \mathcal{C}, o\rangle$ has a designated variable o of $\mathcal{X}$ constrained in $\mathcal{C}$ to take the value of an objective function that is to be minimised (without loss of generality). An *optimal assignment* is a feasible assignment where o is minimal.

2.1 Soft Constraints

Given a satisfaction problem $\langle \mathcal{X}, D, \mathcal{C}\rangle$ and an assignment σ, the *penalty under* σ of a constraint $C(\mathcal{V})$ in $\mathcal{C}$, where $\mathcal{V}$ is an ordered (multi)subset of $\mathcal{X}$, is given by a function π, called the *penalty function*, such that $\pi(\sigma)$ is 0 if $C(\mathcal{V})$ is satisfied under σ, and otherwise a positive number proportional to the degree that $C(\mathcal{V})$ is violated. For example, for the constraint $x + y = z$, the penalty could be $|\sigma(x) + \sigma(y) - \sigma(z)|$, which is the distance between $x+y$ and z under σ. See [20] for a variety of penalty functions in the context of constraint-based local search.

For a constraint $C(\mathcal{V})$ and a penalty function π, the *soft constraint* $C_\pi(\mathcal{V}, p)$ constrains a new variable p, called the *penalty variable*, to take the value π takes.

Example 1. In our experiments of Section 5, the soft constraint for a linear equality constraint $\sum_i \mathcal{A}_i\mathcal{V}_i = c$ is $p = |\sum_i \mathcal{A}_i\mathcal{V}_i - c|$, that is $p = |x + y - z|$ for the unweighted equality constraint $x + y = z$ we considered above. For a linear inequality constraint $\sum_i \mathcal{A}_i\mathcal{V}_i \le c$, we use $p = \max(0,\ \sum_i \mathcal{A}_i\mathcal{V}_i - c)$. For a global cardinality constraint $\mathrm{GCC}(\mathcal{V}, \mathcal{A}, \mathcal{L}, \mathcal{U})$, constraining every value $\mathcal{A}_i$ to be taken between $\mathcal{L}_i$ and $\mathcal{U}_i$ times by the variables $\mathcal{V}_j$, which cannot take any other values, we use $p = \sum_i \max(0,\ \mathcal{L}_i - \sum_j [\mathcal{V}_j = \mathcal{A}_i],\ \sum_j [\mathcal{V}_j = \mathcal{A}_i] - \mathcal{U}_i) + \sum_{d \notin \mathcal{A}} \sum_j [\mathcal{V}_j = d]$, where $[\alpha]$ denotes value 1 if constraint α holds and value 0 otherwise. □

We say that we *soften* a constraint $C(\mathcal{V})$ when we replace it in $\mathcal{C}$ by $C_\pi(\mathcal{V}, p)$ for some π, with variable p added to $\mathcal{X}$ and used in an objective function. We call $C(\mathcal{V})$ the *replaced constraint*, not to be mixed up with $C_\pi(\mathcal{V}, p)$.

We define $\textsc{Soft}(\langle \mathcal{X}, D, \mathcal{C}\rangle, \mathcal{S}, \pi, \lambda)$ as the softening of a subset $\mathcal{S} \subseteq \mathcal{C}$ of n constraints in the satisfaction problem $\langle \mathcal{X}, D, \mathcal{C}\rangle$ into the optimisation problem $\langle \mathcal{X} \cup \{p_i \mid i \in 1..n\} \cup \{o\}, D', \mathcal{C} \setminus \mathcal{S} \cup \{C^i_{\pi_i}(\mathcal{V}, p_i) \mid C^i(\mathcal{V}) \in \mathcal{S}\} \cup \{o = \sum_{i=1}^n \lambda_i p_i\}, o\rangle$ by using the penalty functions π_i and weights $\lambda_i \ge 0$, where D' is D extended to give the initial domain $0 \mathinner{..} \infty$ to the introduced objective variable o and each introduced penalty variable p_i.

Definition 1. For any $\textsc{Soft}(\mathcal{P}, \mathcal{S}, \pi, \lambda)$ of a satisfaction problem $\mathcal{P}$, we define:

- a *pseudo-solution* is a feasible assignment where at least one introduced penalty variable takes a positive value, and therefore the non-penalty variables do not form a feasible assignment for $\mathcal{P}$; and
- a *solution* is a feasible assignment where all penalty variables take the value 0, and therefore the non-penalty variables do form a feasible assignment for $\mathcal{P}$.

Note that a solution to $\textsc{Soft}(\mathcal{P}, \mathcal{S}, \pi, \lambda)$ is in fact an optimal solution to it, as the introduced objective variable takes its lower bound 0, no matter what the weights λ_i are, and thereby that solution establishes the satisfiability of $\mathcal{P}$.

Consider a soft constraint $C_\pi(\mathcal{V}, p)$ and a variable v in $\mathcal{V}$: we say that a value d in $D(v)$ *imposes a penalty* when $\min(D(p))$ would increase if $D(v)$ became $\{d\}$.

2.2 Large-Neighbourhood Search

Large-neighbourhood search (LNS) [19] is a local-search method for solving an optimisation problem $\langle \mathcal{X}, D, \mathcal{C}, o \rangle$. It starts from a feasible assignment σ, which evolves as what we call the *current assignment*. At each iteration, an LNS heuristic selects a non-empty strict subset $\mathcal{M}$ of the variables $\mathcal{X}$, called the *fragment*, where $o \in \mathcal{M}$. The optimisation problem $\langle \mathcal{X}, D, \mathcal{C} \cup \{x = \sigma(x) \mid x \in \mathcal{X} \setminus \mathcal{M}\}, o \rangle$, where all but the variables of the fragment take their current values, is solved, not necessarily to optimality. If an improving feasible assignment is found, then it replaces the current assignment, otherwise the current assignment is unchanged. The search continues from the current assignment by selecting a new fragment.

LNS can in principle be used to solve a satisfaction problem $\mathcal{P}$ that has been softened by some $\textsc{Soft}(\mathcal{P}, \mathcal{S}, \pi, \lambda)$ into an optimisation problem, but we show in Sections 3 and 5 that the performance can be poor in practice.

The optimisation problem at each LNS iteration, as well as the satisfaction problem of finding an initial feasible assignment, can in principle be solved by using any technology, but we here only consider CP.

There exists extensive literature on the challenge of *heuristically* selecting a fragment at each LNS iteration: either by exploiting the structure of the underlying problem [19] or by using more generic methods [9,13,14]. Both approaches can have a significant impact on the performance of LNS.

In this paper, we focus on an orthogonal challenge of LNS, namely efficiently solving (*hard*) *satisfaction problems* (*and even optimisation problems that are hard to satisfy, that is where finding a feasible assignment is difficult*)*, so that there is a need to improve the* propagation *in each LNS iteration in a new way.*

3 Motivation

To make some motivating observations, we consider as a running example the satisfaction problem of subset sum, as solved by CP. We then soften the problem in order to show how it can in principle be solved by LNS.

Example 2. Given an integer set $\mathcal{S}$ and an integer t, the subset-sum problem is to find a subset $\mathcal{S}' \subseteq \mathcal{S}$ such that $\sum_{s \in \mathcal{S}'} s = t$. We can express this as a satisfaction problem using a 0/1 variable x_i for each element of $\mathcal{S}$, and a single constraint, say for $\mathcal{S} = \{11, -3, 2, 5, 9, -6\}$ and $t = 1$:

$$11x_1 - 3x_2 + 2x_3 + 5x_4 + 9x_5 - 6x_6 = 1 \tag{1}$$

Using the classical idempotent bounds-consistency propagator in Algorithm 1 for the linear equality (1) — because achieving domain consistency is NP-hard [2] — and the CP search strategy that branches on the variables in order from x_1 to x_6, always with $x_i = 1$ as the left-branch decision, we obtain the following CP search tree. At the root node, the value 1 is pruned from $D(x_1)$. Upon the decision $x_2 = 1$, no value is pruned. Upon the decision $x_3 = 1$, propagation first prunes the value 1 from both $D(x_5)$ and $D(x_6)$, but then fails as all values of $D(x_4)$ must

Algorithm 1 Bounds-consistency propagator for $\sum_i \mathcal{A}_i \mathcal{V}_i = c$, where $\mathcal{A}$ is an array of integers, $\mathcal{V}$ an equally long array of integer variables, and c an integer; it updates the domain function D. (Idempotent due to lines 1–3 and 12–13.)

```
1:  $done \leftarrow false$
2:  while not $done$ do
3:      $done \leftarrow true$   // this might be the last iteration
4:      $\ell \leftarrow \sum_{i=1}^{|\mathcal{V}|} \mathcal{A}_i \cdot \min(D(\mathcal{V}_i))$;   $u \leftarrow \sum_{i=1}^{|\mathcal{V}|} \mathcal{A}_i \cdot \max(D(\mathcal{V}_i))$
5:      if $\ell = c = u$ then
6:          return SUBSUMED
7:      else if $c < \ell$ or $u < c$ then
8:          return FAILED
9:      for $i = 1$ to $|\mathcal{V}|$ do
10:         $D(\mathcal{V}_i) \leftarrow D(\mathcal{V}_i) \cap \left\lceil \frac{c-u+\max(D(\mathcal{V}_i))}{\mathcal{A}_i} \right\rceil \,..\, \max(D(\mathcal{V}_i))$   // tighten lower bound
11:         $D(\mathcal{V}_i) \leftarrow D(\mathcal{V}_i) \cap \min(D(\mathcal{V}_i)) \,..\, \left\lfloor \frac{c-\ell+\min(D(\mathcal{V}_i))}{\mathcal{A}_i} \right\rfloor$   // tighten upper bound
12:     if some $D(\mathcal{V}_i)$ has changed then
13:         $done \leftarrow false$   // continue iterating
14: return ATFIXPOINT
```

be pruned. The search backtracks, failing at two more nodes, until it finds the only feasible assignment (namely $x_1 = x_2 = x_5 = 0$ and $x_3 = x_4 = x_6 = 1$, corresponding to $\mathcal{S}' = \{2, 5, -6\}$) upon the decisions $x_2 \neq 1$ and $x_3 = 1$. □

To solve a satisfaction problem with LNS, one must soften some constraints in order to turn it into an optimisation problem where a zero-penalty solution corresponds to a feasible assignment to the satisfaction problem. If no such softening is performed, then the initial feasible assignment will be an acceptable solution, which means that LNS adds no benefit.

Although constraints can be softened in a generic way by using penalty functions (as explained in Section 2.1), *softening will in practice significantly increase the size of the CP search space for each LNS iteration, as soft constraints usually only cause propagation towards the bottom of the search tree, where most variables are fixed, and provide little to no propagation that drives the CP search towards an (optimal) solution*, as shown in the following example.

Example 3. In order to solve the subset-sum satisfaction problem of Example 2 by LNS, its constraint (1) must be softened, say as:

$$p = |11x_1 - 3x_2 + 2x_3 + 5x_4 + 9x_5 - 6x_6 - 1| \tag{2}$$

where $p \in 0\,..\,26$, and the objective is to minimise p. By Definition 1, if $p = 0$ then the solution corresponds to a feasible assignment to the satisfaction problem. Consider the CP search tree while finding an initial feasible assignment for LNS, using the search strategy of Example 2. Since p is essentially unconstrained, there is no propagation (no matter what consistency is targeted) and search descends the left branch, arriving at a pseudo-solution where all $x_i = 1$ and

$p = 17$. Improving it into a solution requires $x_1 = 0$: this was achieved by root-node propagation in Example 2, but is here only achievable by x_1 being in a fragment. However, even that may not be enough: if the first fragment is $\{x_1, x_3\}$, then the next pseudo-solution will have $x_3 = 0$ and $p = 15$. Clearly, even in a small instance, early bad decisions severely degrade performance. $\square$

4 Avoiding Bad CP Search Decisions

We are here concerned with constraints that must be softened to enable the use of LNS for satisfaction problems. *We want to improve the CP search for an initial (pseudo-)solution for the first LNS iteration, as well as the CP search within each LNS iteration for better pseudo-solutions and eventually a solution.*

We saw in Example 3 that bad CP search decisions can be made early if the propagation from the *soft* constraints does not prune values that impose high penalties. It could therefore be beneficial to prune *some* of those values, using *some* propagation from the *replaced* constraints, so that the CP search avoids those bad decisions. However, it would be counterproductive to prune all values that impose a penalty, as that would again make the constraints (and by extension the problem) hard to satisfy, namely by causing significant backtracking.

Still, given the crucial role that propagation plays in the effectiveness of CP search, we argue that only replacing some constraints by using penalty functions (and thereby effectively removing all the propagation for those constraints) does not fully utilise the decades of research on efficient and powerful propagators.

Based on this observation, we propose non-failing propagators (Section 4.1), show how to modify a CP solver such that any propagator can be made non-failing without modifying its code (Section 4.2), and discuss how the scheduling of non-failing propagators can impact backtracking (Section 4.3).

4.1 Non-Failing Propagators

We want to extend an optimisation problem $\textsc{Soft}(\langle \mathcal{X}, D, \mathcal{C} \rangle, \mathcal{S}, \pi, \lambda)$ by prescribing *additional* propagators for $\mathcal{S}$ to prune values that would impose a penalty, but without causing backtracking. For this, we propose non-failing propagators:

Definition 2. For a constraint $C(\mathcal{V})$, a *non-failing propagator* prunes domain values that are inconsistent with $C(\mathcal{V})$, but only until the pruning would empty the domain of some variable v in $\mathcal{V}$; at that point, the propagator is subsumed instead of pruning the last domain value (or values) of v and being failed, which would cause backtracking.

For example, the propagator of Algorithm 1 can be turned into a non-failing propagator by rewriting line 8 to return the status Subsumed instead of Failed. In Section 4.2, we achieve this at the *solver* level instead of the propagator level.

We use the notation $C(\mathcal{V}) :: \textsc{NonFailing}$ to indicate that the propagator used for $C(\mathcal{V})$ should be non-failing.

A non-failing propagator never causes backtracking itself, but it can do so *indirectly* by pruning values that make the normal propagators for the constraints $\mathcal{C} \backslash \mathcal{S}$ be failed. Just like any propagator that is disabled during CP search, a non-failing propagator is restored upon backtracking and restarts. Non-failing propagators are safe to use on problems that are satisfiable:

Theorem 1. *If a problem is satisfiable, then non-failing propagators for any of its constraints will never remove any feasible assignments from the search space.*

Proof. Consider a feasible assignment σ for a problem. A non-failing propagator prunes no more than a normal propagator, by Definition 2. Therefore, no propagator for any constraint of the problem can prune a value occurring in σ if all values in σ are in the domains of the corresponding variables. $\square$

We define $\textsc{Soft}_{\text{nonfail}}(\mathcal{P}, \mathcal{S}, \pi, \lambda)$ as the softening of the constraints $\mathcal{S}$ in the satisfaction problem $\mathcal{P}$ via $\textsc{Soft}(\mathcal{P}, \mathcal{S}, \pi, \lambda)$ but with the addition of the constraints $\{C(\mathcal{V})::\textsc{NonFailing} \mid C(\mathcal{V}) \in \mathcal{S}\}$: that is, the constraints $\mathcal{S}$ are implemented *both* by non-failing propagators for themselves *and* by normal propagators (or decompositions) for their soft versions.

Example 4. The application of $\textsc{Soft}_{\text{nonfail}}$ to the subset-sum problem of Example 2 uses *both* a non-failing propagator for its constraint (1) *and* normal propagation for its soft constraint from Example 3:

$$\begin{aligned} &\text{minimise } p \\ &\text{such that } (11x_1 - 3x_2 + 2x_3 + 5x_4 + 9x_5 - 6x_6 = 1)::\textsc{NonFailing} \\ &\qquad\qquad p = |11x_1 - 3x_2 + 2x_3 + 5x_4 + 9x_5 - 6x_6 - 1| \\ &\qquad\qquad x_i \in \{0,1\}, \qquad \forall i \in 1..6 \end{aligned}$$

Root-node propagation of the non-failing propagator prunes value 1 from $D(x_1)$, as in Example 2 and unlike in Example 3. Given the same CP search strategy as in Example 2, the first decision, $x_2 = 1$, does not trigger any propagation. The second decision, $x_3 = 1$, causes the non-failing propagator to first prune value 1 from $D(x_5)$, then prune value 1 from $D(x_6)$, and then infer failure. However, rather than failing, the propagator is subsumed. The next decision is then $x_4 = 1$, at which point the CP search is at a pseudo-solution (namely $x_1 = x_5 = x_6 = 0$ and $x_2 = x_3 = x_4 = 1$). The propagator of the soft constraint from Example 3 so far had no impact on the decisions, but gives $p = 3$ at this pseudo-solution. Given this initial LNS assignment, if a fragment consisting of x_2 and x_6 is selected in the first LNS iteration, then root-node propagation of the non-failing propagator will immediate solve the problem by inferring $x_2 = 0$ and $x_6 = 1$, giving $p = 0$. Unlike in Example 3, where only the soft constraint is used, we here see how a non-failing propagator can help the CP searches avoid bad decisions and also allow the LNS search to quickly arrive at a zero-penalty assignment. While this example was specifically constructed to showcase this behaviour, our experimental evaluation in Section 5 shows that this seems also to be beneficial in practice, across a variety of benchmarks. $\square$

Algorithm 2 FixPoint(P, Q, D), where P is the set of all non-disabled propagators (initially those for the constraints $\mathcal{C}$ of the problem), Q is the priority queue of propagators not known to be at fixpoint (initially those for $\mathcal{C}$, later those of a CP search decision), and D is the function giving the current domains.

```
 1: while $Q$ is not empty do
 2:   $p \leftarrow Q$.dequeue()
 3:   $status \leftarrow p$.propagate($D$)   // note that this can enqueue propagators of $P$
 4:   if $status$ = SUBSUMED then
 5:     $p$.disable()   // this achieves $P \leftarrow P \setminus \{p\}$
 6:   else if $status$ = FAILED then
 7:     return FAILURE   // fail and cause backtracking
 8:   else
 9:     ...   // other status messages are not relevant here
10: return AtCommonFixpoint
```

4.2 Implementation

In principle, any propagator can be made non-failing by modifying its code (such as in the example after Definition 2). However, this can be both tedious and error-prone. Fortunately, we can instead modify the propagation engine of a CP solver to treat as non-failing any propagator tagged as ::NonFailing.

Algorithm 2 shows a typical fixpoint algorithm, based on a queue of propagators that need to be executed. The only change required to support non-failing propagators is to replace line 4 by

if *status* = Subsumed **or** (non-failing(p) **and** *status* = Failed) **then**

so that when a propagator tagged as non-failing returns the status Failed, then the status is instead treated as Subsumed. However, in order for this modification to Algorithm 2 to be correct, the CP solver and its propagators must guarantee that the domains of the variables are always in a *consistent state*: when a propagator returns Failed, the domain of each variable must be a *non-empty* subset of the domain before running the propagator.

For our experiments in Section 5, we modified Gecode [6] in this way. Domain updates in Gecode are not guaranteed to leave the domains in a consistent state after failure: we therefore modified the domain update functions to check whether an update would result in a domain wipe-out *before* they modify that domain.

4.3 Scheduling of Non-Failing Propagators

Non-failing propagators are non-monotonic: the amount of propagation they achieve (and whether they propagate or are subsumed) depends on the order in which all propagators are executed [18]. Many CP solvers order propagators using a *priority queue*, for example based on their algorithmic complexity [17].

The priority assigned to non-failing propagators can therefore determine if a node fails or succeeds. However:

Theorem 2. *There is no static priority order (independent of the internal state of the fixpoint algorithm) of propagators that guarantees that non-failing propagators only cause failure when failure occurs for all possible priority orders.*

Proof. Consider the following problem:

$$D(x) = D(y) = D(z) = 0\,..\,5,\ D(b_0) = D(b_1) = D(b_2) = 0\,..\,1$$
$$x + y + z = 5,\ b_1 \rightarrow x \geq 2,\ b_1 \rightarrow y \geq 2,\ b_2 \rightarrow x \leq 1,\ b_2 \rightarrow y \leq 1$$
$$b_0 \rightarrow b_1 :: \textsc{Nonfailing},\ b_0 \rightarrow b_2 :: \textsc{Nonfailing}$$

where the non-failing propagators are p_1 for $b_0 \rightarrow b_1 :: \textsc{Nonfailing}$ and p_2 for $b_0 \rightarrow b_2 :: \textsc{NonFailing}$. We assume that both non-failing propagators are always propagated last, as that decreases the probability of failure.

Upon the initial CP search decision $z \geq 2$, we reach the node where $D(z) = 2\,..\,5$ and $D(x) = D(y) = 0\,..\,3$. If we make the decision $b_0 = 1$, then both p_1 and p_2 are enqueued. If p_1 is dequeued first, then we propagate $b_1 = 1$, $x \geq 2$, $y \geq 2$, $b_2 = 0$, and then the propagator for $x + y + z = 5$ is dequeued and fails. If p_2 is dequeued first, then we propagate $b_2 = 1$, $x \leq 1$, $y \leq 1$, $b_1 = 0$, $z \geq 1$, and then p_1 is dequeued and subsumed (because it fails). That is, the node only succeeds when p_2 runs before p_1.

Upon the opposite CP search decision $z < 2$, we reach the node where $D(z) = 0\,..\,1$. If we make the decision $b_0 = 1$, then both p_1 and p_2 are enqueued. If p_1 is dequeued first, then we propagate $b_1 = 1$, $x \geq 2$, $y \geq 2$, $b_2 = 0$, and then p_2 is dequeued and subsumed (because it fails). If p_2 is dequeued first, then we propagate $b_2 = 1$, $x \leq 1$, $y \leq 1$, $b_1 = 0$, and then the propagator for $x+y+z = 5$ is dequeued and fails. That is, the node only succeeds when p_1 runs before p_2.

So, no static priority order can always avoid indirect failures caused by non-failing propagators, even when the indirect failure could have been avoided. □

Empirically, we found it beneficial to always run non-failing propagators at the lowest priority. Intuitively, this makes sense: consider two propagators p_1 and p_2, where running p_1 causes p_2 to fail, and vice versa: if only p_1 is non-failing, then backtracking is only avoided when running p_1 after p_2. We therefore modified Gecode to schedule all non-failing propagators to run as late as possible, with a first-in-first-out tie-breaking between non-failing propagators.

5 Experimental Evaluation

This section presents an empirical evaluation of the benefit of non-failing propagators for both satisfaction and optimisation problems.

5.1 Setup

We compare three approaches to finding a solution to a satisfaction problem $\mathcal{P}$:

hard treat all constraints as hard, that is: solve $\mathcal{P}$ by CP;

soft soften some constraints $\mathcal{S}$ of $\mathcal{P}$, that is: solve $\textsc{Soft}(\mathcal{P}, \mathcal{S}, \pi, \lambda)$ by LNS;
non-failing soften the same constraints $\mathcal{S}$ of $\mathcal{P}$, but also use non-failing propagators for those constraints, that is: solve $\textsc{Soft}_{\text{nonfail}}(\mathcal{P}, \mathcal{S}, \pi, \lambda)$ by LNS.

For all problems, we used published MiniZinc (version 2.4.3) [12] models.[3]

We modified Gecode (version 6.2.0) [6] as described in Sections 4.2 and 4.3. We modified its MiniZinc interface so that some constraints, when tagged with a new `soften` annotation for MiniZinc, are automatically softened under the **soft** and **non-failing** approaches with $\lambda = \bar{1}$ and using the penalty functions π in Example 1. For the **non-failing** approach, the `soften` annotation also tags the constraint with the $::\textsc{NonFailing}$ annotation, that is, we have *both* a soft version *and* a non-failing version of the tagged constraint when using the **non-failing** approach. To propagate a soft constraint, we use a decomposition of its penalty function rather than a specialised propagator. We used Gecode's built-in LNS via its MiniZinc interface:[4] it selects a variable *not* to be in the fragment under a given probability; we prescribed a probability of 70% or 80%, depending on the size of the problem instances,[3] and used the `constant` restart strategy.

For each problem instance, we report the average time for finding a first solution by LNS, over 10 independent runs, each allocated 10 minutes. The average was only computed over the runs where a solution was actually found.

We used Linux Ubuntu 18.04 (64 bit) on an Intel Xeon E5520 of 2.27 GHz, with 4 processors of 4 cores each, with 24 GB RAM and an 8 MB L2 cache. Note that we only run Gecode on a single core for our experiments.

5.2 Satisfaction Problems

We want to see whether our new generic **non-failing** approach allows LNS to outperform the classical generic **soft** approach to LNS for satisfaction problems, and whether both beat the **hard** approach via only CP. We look at instances of three satisfaction problems that are difficult to solve with Gecode via MiniZinc.

Nurse Rostering. We use the model for a simple nurse rostering problem from the MiniZinc Handbook[5] but modify it by using global-cardinality constraints on the daily numbers of nurses on each shift. We handcrafted $10 + 10 = 20$ satisfiable instances to be either *easy* (by having many nurses available) or *difficult* (by being at the border of unsatisfiability in terms of available nurses), both for Gecode under the **hard** approach. We prescribe softening for all the global-cardinality constraints. In Figure 1a, we see that **soft** solves fewer instances than **hard** and often needs over an order of magnitude more time (both only solve the easy instances), while **non-failing** solves all but one (difficult) instance and does so with seemingly no overhead compared to **hard** (on the easy instances).

[3] We modified the models in order to deploy more global constraints and better CP search strategies. Our versions of the MiniZinc models, the instances, and the Gecode library are available at `https://github.com/astra-uu-se/CP2020`.
[4] See `https://www.minizinc.org/doc-latest/en/lib-gecode.html`
[5] Section 2.3.1.4 of `https://www.minizinc.org/doc-latest/en/predicates.html`

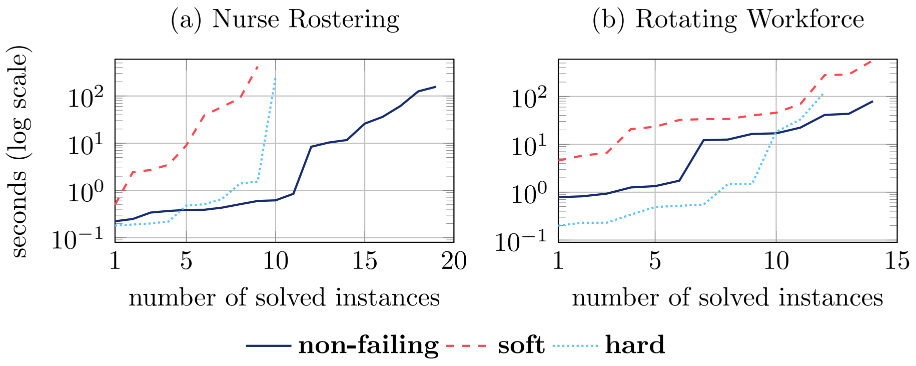

Figure 1. Number of instances that are each solved within a given time.

Rotating Workforce. In the rotating workforce problem [11], a roster is to be built for employees, satisfying complex rules on sequences of shifts. We use the model and 50 instances in [11]: these are difficult for Gecode under the **hard** approach, although not too difficult for mixed-integer-programming and lazy-clause-generation solvers [11]. We prescribe softening for the global-cardinality constraints on the daily numbers of assigned shifts. In Figure 1b, we see that each approach only solves at most 14 of the 50 instances: **soft** is slowest but solves as many instances (though not the same) as **non-failing**, while **hard** is arguably fastest but solves fewer instances than **non-failing**.

Car Sequencing. In this problem [4], a sequence of cars of various classes, each class having a set of options, is to be produced, satisfying capacity constraints on the options over a sliding window on the sequence and occurrence constraints on the classes. We use the MiniZinc Benchmark model[6] and the classic 78 instances for sequences of 100 to 200 cars.[7] We prescribe softening for the capacity constraints, which are expressed by linear inequalities. A problem-specific softening, which does *not* rely on penalties in the spirit of those in Example 1, was successfully used with LNS in [13]: we model it in MiniZinc and solve it by LNS with Gecode calling this the **reformulation** approach. In Figure 2, we see that **hard** only solves 4 instances, while **soft** solves 65 instances but takes an order of magnitude more time than **non-failing**, which solves 69 of the 78 instances; **reformulation** solves 67 instances and takes time between **soft** and **non-failing**.

5.3 Hard-to-Satisfy Optimisation Problems

For a hard-to-satisfy optimisation problem, we solve the *satisfaction* problem of finding a *first* solution, which enables another round of LNS to find better ones.

[6] Available at `https://github.com/MiniZinc/minizinc-benchmarks`
[7] Available also at `http://www.csplib.org/Problems/prob001/data/data.txt.html`

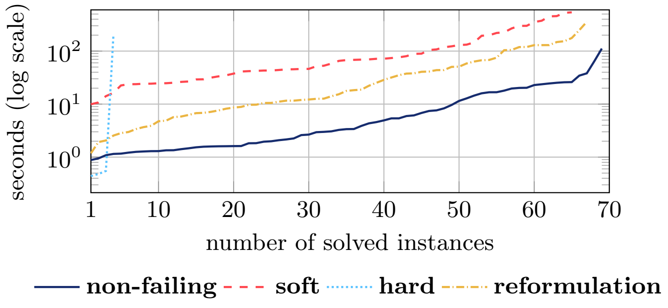

Figure 2. Number of car-sequencing instances that are each solved within a given time.

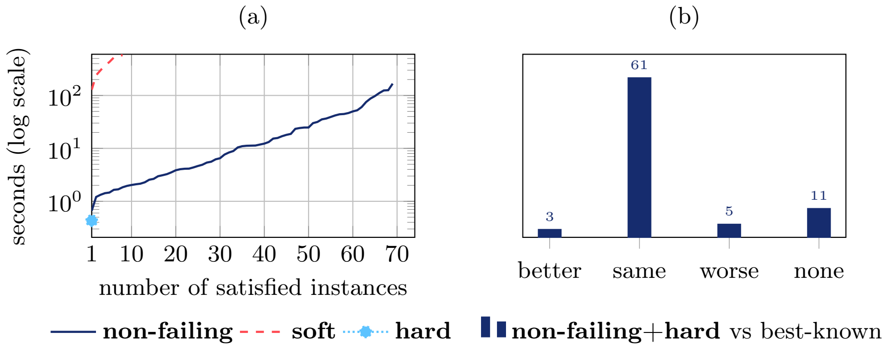

Figure 3. (a) Number of TSPTW instances that are each satisfied within a given time. (b) Numbers of TSPTW instances where our best-found minimum is better, the same, or worse than the best-known one; 'none' means we found no feasible assignment.

TSPTW. In the travelling salesperson problem (TSP) with time windows, a shortest tour visiting all nodes in a graph during their time windows is to be found. We use the 80 satisfiable `n40`, `n60`, and `n80` instances of the Gendreau-DumasExtended benchmark,[8] as they are very difficult to satisfy under the **hard** approach. We prescribe softening for the linear inequalities that require the arrival time at a node to be at least the arrival time at its predecessor plus the travel time in-between. In Figure 3a, we see that **hard** only satisfies 1 instance and **soft** only 8 instances, while **non-failing** satisfies 69 of the 80 instances. When **non-failing** finds a solution within the allocated 10 minutes, we switch to **hard**, but with LNS, in order to try and improve it for the remaining time. We call this the **non-failing+hard** approach. In Figure 3b, we compare the best solutions found by **non-failing+hard** to the best known solutions from the literature.[8] Our new approach can improve these bounds for three instances.

[8] Available at `http://lopez-ibanez.eu/tsptw-instances`

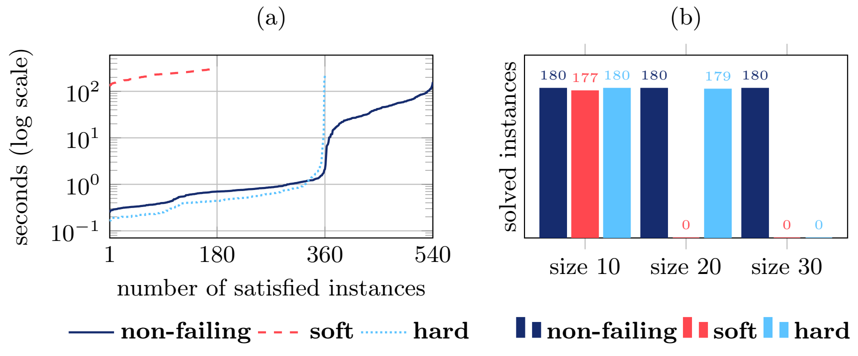

Figure 4. (a) Number of TDTSP instances that are each satisfied within a given time. (b) Numbers of satisfied TDTSP instances for the three batches of instances.

TDTSP. In the time-dependent TSP [1], a shortest tour visiting all nodes in a graph during their time windows is to be found, similarly to TSPTW, but the travel time between two nodes depends on the arrival time at the first node. We use the model of the MiniZinc Challenge 2017,[9] but modify it to constrain the tour using successor variables and a `circuit` global constraint [8]. We prescribe softening as for TSPTW. By private communication with the author of the original MiniZinc model, we received 540 generated instances of 10, 20, and 30 nodes, with 180 instances for each batch; some of the size-10 and size-20 instances were used in the Challenge, but none of the size-30 ones as, for most of them, no MiniZinc backend ever found a feasible solution so that they were deemed too hard. In Figure 4a, we see that **soft** only satisfies 177 instances and is over two orders of magnitude slower than the other approaches. Note that **hard** satisfies 359 instances about as fast as **non-failing**, but as the instances become more difficult to satisfy its runtime quickly increases. In Figure 4b, we see that **soft** only satisfies instances of size 10, whereas **hard** only satisfies instances of size at most 20, and **non-failing** satisfies all the instances.

HRC. For the hospitals/residents matching problem with couples (HRC), we use the model and 5 instances of the MiniZinc Challenge 2019.[9] We prescribe softening for the linear inequalities on the hospital capacities. In Table 1, we see that **soft** satisfies all the instances, whereas **non-failing** completely backfires and satisfies no instance (and actually does not even find a pseudo-solution for any instance). The non-failing propagators prevent the CP search from reaching a pseudo-solution due to too many *indirect* failures. Furthermore, since **hard** can solve 3 instances, it seems that the CP search space for **non-failing** is larger than for **hard**. We describe how to address indirect failures in Section 7.

[9] Available at `https://www.minizinc.org/challenge.html`

Table 1. Runtimes (in seconds, or '–' if more than 600) to satisfy each HRC instance: the **non-failing** approach backfires.

instance	**non-failing**	**soft**	**hard**
exp1-1-5110	–	83.47	26.96
exp1-1-5425	–	21.98	240.74
exp1-1-5460	–	341.01	40.89
exp2-1-5145	–	65.10	–
exp2-1-5600	–	208.22	–

6 Related Work

We now discuss three areas of related work: soft constraints, variable-objective LNS, and streamliners.

Traditionally, soft constraints for CP have almost exclusively been researched in the context of over-constrained problems: see [10] and [16, Chapter 3] for extensive overviews. In this paper, we *assume* the problem is *not* over-constrained but hard to satisfy and therefore requires softening to enable the use of LNS. To the best of our knowledge, there exists very little research on softening problems that are *not* over-constrained, and replacing constraints by using penalties seems to originate from the over-constrained setting. The only other work we found is [5], which generalises Lagrangian relaxations to CP models.

Variable-objective LNS (VO-LNS) [15] is based on the observation that the penalty variables introduced by softening are usually connected to the objective variable by a linear equality and any new bounds on the objective variable result in little to no propagation on the penalty variables. Therefore, VO-LNS eagerly bounds penalty variables during branch-and-bound. This achieves more propagation from the soft constraints. This is conceptually related to our approach: we improve the poor propagation from the soft constraints and reduce their negative impact on LNS, by pruning more. VO-LNS satisfies Theorem 1: it never removes a solution from the CP search space if the problem is satisfiable. But, like our approach, VO-LNS can remove pseudo-solutions from the CP search space, and might therefore remove all pseudo-solutions with the lowest positive penalty. VO-LNS and non-failing propagators can be complementary: first experiments, where both approaches are used together, indicate that there can be a synergy.

Streamliners [7] are constraints added in order to remove a large portion of the CP search space while ideally not removing all solutions. Streamliners are identified by empirically observing structures in solutions to easy instances and hoping that those structures, and thereby constraints, extend to difficult instances. However, while streamliners are ideally safe, by not removing non-dominated solutions, they are not always guaranteed to be safe; their addition can even make a satisfiable instance unsatisfiable. Non-failing propagators, when added to a problem $\textsc{Soft}(\mathcal{P}, \mathcal{S}, \pi, \lambda)$, can be thought of as streamliners since they remove a large portion of the CP search space. But, unlike streamliners, non-failing propagators are always safe to use as they never make a satisfiable instance unsatisfiable, due to Theorem 1, and they are not based on empirical observation, but rather on the actual constraints of the problem.

7 Conclusion and Future Work

LNS is a powerful approach to solving difficult problems, but is typically only applied to (easy-to-satisfy) optimisation problems. We show that by using our non-failing propagators we can apply LNS to effectively tackle hard satisfaction problems (including those arising from hard-to-satisfy optimisation problems). Implementing non-failing propagators is not difficult in a CP solver, and can be done at the engine level with some care. Experimental results show that non-failing propagators can drastically improve the solving of hard-to-satisfy problems, although they are not universally beneficial.

Future work includes the design of constraint-specific non-failing propagators. For example, consider $\textsc{AllDifferent}([x_1, x_2, x_3, \ldots, x_n])$: rather than disabling its propagator upon detecting $x_1 = x_2$, one can replace it by a propagator for $\textsc{AllDifferent}([x_3, \ldots, x_n])$, thereby still avoiding the failure but now without losing the propagation on the remaining variables.

A weakness of non-failing propagators is that they can cause too many *indirect* failures, via normal propagators, as seen in Table 1. Indirect failures can sometimes be avoided by giving non-failing propagators the right priority (see Section 4.3). As future work, we can address this weakness with the following two orthogonal ideas. First, when using a learning CP solver such as Chuffed [3], which explains failures, we can detect that a failure is indirect and we can identify a node where a responsible non-failing propagator ran: the CP search can then backtrack and disable that propagator at that node, thus avoiding the failure. Second, non-failing propagators can be made more cautious about the values they prune: for example, rather than eagerly pruning values that impose a penalty, a non-failing propagator can prune only the values that impose a penalty above some threshold, which can be adjusted during CP search. Initial experiments show that these ideas work in principle, but do not yet outperform our generic implementation (of Section 4.2), as they bring their own sets of challenges, which require further investigation.

Acknowledgements. We thank the reviewers for their constructive feedback, and Graeme Gange for discussions on how to implement these ideas in LCG solvers. This work is supported by the Swedish Research Council (VR) through Project Grant 2015-04910, by the Vice-Chancellor's travel grant from the Wallenberg Foundation for a visit by the first author to Monash University, and by the Australian Research Council grant DP180100151.

References

1. Aguiar Melgarejo, P., Laborie, P., Solnon, C.: A time-dependent no-overlap constraint: Application to urban delivery problems. In: Michel, L. (ed.) CP-AI-OR 2015. LNCS, vol. 9075, pp. 1–17. Springer (2015)
2. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: Sattar, A., Kang, B.H. (eds.) AI 2006. LNCS, vol. 4304, pp. 49–58. Springer (2006)
3. Chu, G.: Improving Combinatorial Optimization. Ph.D. thesis, Department of Computing and Information Systems, University of Melbourne, Australia (2011), available at `http://hdl.handle.net/11343/36679`; the Chuffed solver and MiniZinc backend are available at `https://github.com/chuffed/chuffed`
4. Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving the car-sequencing problem in constraint logic programming. In: Kodratoff, Y. (ed.) ECAI 1988. pp. 290–295. Pitman (1988)
5. Fontaine, D., Michel, L., Van Hentenryck, P.: Constraint-based Lagrangian relaxation. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 324–339. Springer (2014)
6. Gecode Team: Gecode: A generic constraint development environment (2020), the Gecode solver and its MiniZinc backend are available at `https://www.gecode.org`
7. Gomes, C., Sellmann, M.: Streamlined constraint reasoning. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 274–287. Springer (2004)
8. Laurière, J.L.: A language and a program for stating and solving combinatorial problems. Artificial Intelligence **10**(1), 29–127 (February 1978)
9. Lombardi, M., Schaus, P.: Cost impact guided LNS. In: Simonis, H. (ed.) CP-AI-OR 2014. LNCS, vol. 8451, pp. 293–300. Springer (2014)
10. Meseguer, P., Rossi, F., Schiex, T.: Soft constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 9, pp. 281–328. Elsevier (2006)
11. Musliu, N., Schutt, A., Stuckey, P.J.: Solver independent rotating workforce scheduling. In: van Hoeve, W.J. (ed.) CP-AI-OR 2018. LNCS, vol. 10848, pp. 429–445. Springer (2018)
12. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer (2007), the MiniZinc toolchain is available at `https://www.minizinc.org`
13. Perron, L., Shaw, P., Furnon, V.: Propagation guided large neighborhood search. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 468–481. Springer (2004)
14. Prud'homme, C., Lorca, X., Jussien, N.: Explanation-based large neighborhood search. Constraints **19**(4), 339–379 (October 2014)
15. Schaus, P.: Variable objective large neighborhood search: A practical approach to solve over-constrained problems. In: Brodsky, A. (ed.) ICTAI 2013. pp. 971–978. IEEE Computer Society (2013)
16. Schiendorfer, A.: Soft Constraints in MiniBrass: Foundations and Applications. Ph.D. thesis, Universität Augsburg, Germany (2018), available at `https://doi.org/10.13140/RG.2.2.10745.72802`
17. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. ACM Transactions on Programming Languages and Systems **31**(1), 1–43 (December 2008)
18. Schulte, C., Tack, G.: Weakly monotonic propagators. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 723–730. Springer (2009)

19. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer (1998)
20. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)