

Automatic Relaxation and Multi-Armed Bandit Learning for Large Neighbourhood Search

Frej Knutar Lewander, Pierre Flener,
Justin Pearson, Peter J. Stuckey

Department of Information Technology
Uppsala University
Sweden

CP 2026
July 22, 2026

Large Neighbourhood Search (LNS)

LNS is a **local**-search approach to solving **optimisation** problems.

Large Neighbourhood Search (LNS)

LNS is a **local**-search approach to solving **optimisation** problems.

1. First, an initial **feasible** incumbent is found.

Large Neighbourhood Search (LNS)

LNS is a **local**-search approach to solving **optimisation** problems.

1. First, an initial **feasible** incumbent is found.
2. Then, the incumbent is improved in LNS iterations.

Large Neighbourhood Search (LNS)

LNS is a **local**-search approach to solving **optimisation** problems.

1. First, an initial **feasible** incumbent is found.
2. Then, the incumbent is improved in LNS iterations.

There are drawbacks to LNS.

Large Neighbourhood Search (LNS)

LNS is a **local**-search approach to solving **optimisation** problems.

1. First, an initial **feasible** incumbent is found.
2. Then, the incumbent is improved in LNS iterations.

There are drawbacks to LNS.

- An initial **feasible** incumbent is required to kickstart the LNS iterations, but that would already be a solution to a satisfaction problem.

Large Neighbourhood Search (LNS)

LNS is a **local**-search approach to solving **optimisation** problems.

1. First, an initial **feasible** incumbent is found.
2. Then, the incumbent is improved in LNS iterations.

There are drawbacks to LNS.

- An initial **feasible** incumbent is required to kickstart the LNS iterations, but that would already be a solution to a satisfaction problem.
- Feasibility is difficult to establish for some (difficult-to-satisfy) optimisation problems.

Large Neighbourhood Search (LNS)

LNS is a **local**-search approach to solving **optimisation** problems.

1. First, an initial **feasible** incumbent is found.
2. Then, the incumbent is improved in LNS iterations.

There are drawbacks to LNS.

- An initial **feasible** incumbent is required to kickstart the LNS iterations, but that would already be a solution to a satisfaction problem.
- Feasibility is difficult to establish for some (difficult-to-satisfy) optimisation problems.

This talk: automatic relaxation of **any** satisfaction or (difficult-to-satisfy) optimisation problem.

Running example: Car Sequencing

Consider an assembly line for cars:

- There are 3 types of cars: , , .
- Produce 2 cars per type (that is 6 cars).
- There are stations for 2 options: painting in red, and mounting a windshield.
- For each option: at most 2 of any 3 consecutive cars have that option.

A **solution**:



Running example: Car Sequencing Model

For all i : let $t_i \in \{\text{🚚}, \text{🏎️}, \text{🚗}\}$ denote the type of car i . Let $r_i = \checkmark$ denote that car i is red, else $r_i = \text{👎}$. Let $w_i = \checkmark$ denote that car i has a windshield, else $w_i = \text{👎}$.

Car Type			
Red Paint			
Windshield			

Running example: Car Sequencing Model

For all i : let $t_i \in \{\text{red pickup}, \text{red sports car}, \text{blue SUV}\}$ denote the type of car i . Let $r_i = \checkmark$ denote that car i is red, else $r_i = \text{thumbs down}$. Let $w_i = \checkmark$ denote that car i has a windshield, else $w_i = \text{thumbs down}$.

- Per option: at most 2 of any 3 consecutive cars have that option ($\checkmark \rightsquigarrow 1$, $\text{thumbs down} \rightsquigarrow 0$).

SLIDINGSUM(0, 2, 3, $[r_1, \dots, r_6]$)

SLIDINGSUM(0, 2, 3, $[w_1, \dots, w_6]$)

Car Type			
Red Paint			
Windshield			

Running example: Car Sequencing Model

For all i : let $t_i \in \{\text{red truck}, \text{red sports car}, \text{blue car}\}$ denote the type of car i . Let $r_i = \checkmark$ denote that car i is red, else $r_i = \text{thumbs down}$. Let $w_i = \checkmark$ denote that car i has a windshield, else $w_i = \text{thumbs down}$.

- Per option: at most 2 of any 3 consecutive cars have that option ($\checkmark \rightsquigarrow 1$, $\text{thumbs down} \rightsquigarrow 0$).

$\text{SLIDINGSUM}(0, 2, 3, [r_1, \dots, r_6])$

$\text{SLIDINGSUM}(0, 2, 3, [w_1, \dots, w_6])$

- $\forall i : [\checkmark, \checkmark, \text{thumbs down}][t_i] = r_i$
 $\forall i : [\checkmark, \text{thumbs down}, \checkmark][t_i] = w_i$

Car Type			
Red Paint			
Windshield			

Running example: Car Sequencing Model

For all i : let $t_i \in \{\text{red truck}, \text{red sports car}, \text{blue car}\}$ denote the type of car i . Let $r_i = \checkmark$ denote that car i is red, else $r_i = \text{thumbs down}$. Let $w_i = \checkmark$ denote that car i has a windshield, else $w_i = \text{thumbs down}$.

- Per option: at most 2 of any 3 consecutive cars have that option ($\checkmark \rightsquigarrow 1$, $\text{thumbs down} \rightsquigarrow 0$).

$\text{SLIDINGSUM}(0, 2, 3, [r_1, \dots, r_6])$

$\text{SLIDINGSUM}(0, 2, 3, [w_1, \dots, w_6])$

- $\forall i : [\checkmark, \checkmark, \text{thumbs down}][t_i] = r_i$
 $\forall i : [\checkmark, \text{thumbs down}, \checkmark][t_i] = w_i$
- Produce 2 cars per type.

$\text{GCC}([t_1, \dots, t_6], [\text{red truck}, \text{red sports car}, \text{blue car}]) = [2, 2, 2]$

Car Type			
Red Paint			
Windshield			

Generic Model Relaxation for LNS

1. Copy the constraints of the given model that are easy to satisfy simultaneously

Generic Model Relaxation for LNS

1. Copy the constraints of the given model that are easy to satisfy simultaneously
2. Relax the remaining constraints of the given model (as they are difficult to satisfy)

Generic Model Relaxation for LNS

1. Copy the constraints of the given model that are easy to satisfy simultaneously 🙌
2. Relax the remaining constraints of the given model (as they are difficult to satisfy)

A **dependency constraint** $c(\mathcal{I}, \mathcal{O})$ fixes the variables in $\mathcal{O}$, called its *output variables*, when the variables in $\mathcal{I}$, called its *input variables*, become fixed.

A **dependency constraint** $c(\mathcal{I}, \mathcal{O})$ fixes the variables in $\mathcal{O}$, called its *output variables*, when the variables in $\mathcal{I}$, called its *input variables*, become fixed.

We say that $c(\mathcal{I}, \mathcal{O})$ **functionally defines** $\mathcal{O}$ from $\mathcal{I}$.

1. Dependency Constraints

[FKL et al. @ CP'25]

A **dependency constraint** $c(\mathcal{I}, \mathcal{O})$ fixes the variables in $\mathcal{O}$, called its *output variables*, when the variables in $\mathcal{I}$, called its *input variables*, become fixed.

We say that $c(\mathcal{I}, \mathcal{O})$ **functionally defines** $\mathcal{O}$ from $\mathcal{I}$.

Examples

- ELEMENT constraints, such as $[\checkmark, \text{👎}, \checkmark][t_3] = w_3$.

A **dependency constraint** $c(\mathcal{I}, \mathcal{O})$ fixes the variables in $\mathcal{O}$, called its *output variables*, when the variables in $\mathcal{I}$, called its *input variables*, become fixed.

We say that $c(\mathcal{I}, \mathcal{O})$ **functionally defines** $\mathcal{O}$ from $\mathcal{I}$.

Examples

- ELEMENT constraints, such as $[\checkmark, \text{👎}, \checkmark][t_3] = w_3$.
- Note that $\text{GCC}([t_1, \dots, t_6], [\text{🚚}, \text{🚧}, \text{🚗}]) = [2, 2, 2]$ is **not** a dependency constraint.

A **dependency constraint** $c(\mathcal{I}, \mathcal{O})$ fixes the variables in $\mathcal{O}$, called its *output variables*, when the variables in $\mathcal{I}$, called its *input variables*, become fixed.

We say that $c(\mathcal{I}, \mathcal{O})$ **functionally defines** $\mathcal{O}$ from $\mathcal{I}$.

Examples

- ELEMENT constraints, such as $[\checkmark, \text{👎}, \checkmark][t_3] = w_3$.
- Note that $\text{GCC}([t_1, \dots, t_6], [\text{🚗}, \text{🚗}, \text{🚗}]) = [2, 2, 2]$ is **not** a dependency constraint.
- Also, $\text{SLIDINGSUM}(0, 2, 3, [w_1, \dots, w_6])$ is **not** a dependency constraint.

The Dependency Graph

The dependency constraints induce a directed graph, called the **dependency graph**, where:

- variables and dependency constraints are the nodes;

The Dependency Graph

The dependency constraints induce a directed graph, called the **dependency graph**, where:

- variables and dependency constraints are the nodes;
- for each dependency constraint, there is an incoming edge from each input variable and an outgoing edge to each output variable; and

The Dependency Graph

The dependency constraints induce a directed graph, called the **dependency graph**, where:

- variables and dependency constraints are the nodes;
- for each dependency constraint, there is an incoming edge from each input variable and an outgoing edge to each output variable; and
- the nodes without incoming edges are the **source variables**, and they transitively functionally define all other variables.

The Dependency Graph

The dependency constraints induce a directed graph, called the **dependency graph**, where:

- variables and dependency constraints are the nodes;
- for each dependency constraint, there is an incoming edge from each input variable and an outgoing edge to each output variable; and
- the nodes without incoming edges are the **source variables**, and they transitively functionally define all other variables.

For car sequencing and each **ELEMENT** constraint, there is an incoming edge from a t_i and an outgoing edge to either r_i or w_i .

Generic Model Relaxation for LNS

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

Generic Model Relaxation for LNS

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

1. Model L has all the constraints of model M that are easy to satisfy simultaneously.

Generic Model Relaxation for LNS

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

1. Model L has all the constraints of model M that are easy to satisfy simultaneously.
2. Model L has relaxations (see a later slide) of all the other constraints of model M .

Generic Model Relaxation for LNS

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

1. Model L has all the constraints of model M that are easy to satisfy simultaneously.
 - 1a. All the dependency constraints of M are easy to satisfy simultaneously.

2. Model L has relaxations (see a later slide) of all the other constraints of model M .

Generic Model Relaxation for LNS

Given a model M of a satisfaction or optimisation problem, produce a relaxation L of M that is suitable for LNS initialisation and iterations:

1. Model L has all the constraints of model M that are easy to satisfy simultaneously.
 - 1a. All the dependency constraints of M are easy to satisfy simultaneously.

Example: Our car-sequencing model has two `ELEMENT` dependency constraints per car.

2. Model L has relaxations (see a later slide) of all the other constraints of model M .

Generic Model Relaxation for LNS

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

1. Model L has all the constraints of model M that are easy to satisfy simultaneously.
 - 1a. All the dependency constraints of M are easy to satisfy simultaneously, except if the dependency graph is cyclic: need for cycle elimination (see extra slide).
Example: Our car-sequencing model has two `ELEMENT` dependency constraints per car.
2. Model L has relaxations (see a later slide) of all the other constraints of model M .

Generic Model Relaxation for LNS

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

1. Model L has all the constraints of model M that are easy to satisfy simultaneously.
 - 1a. All the dependency constraints of M are easy to satisfy simultaneously, except if the dependency graph is cyclic: need for cycle elimination (see extra slide).
Example: Our car-sequencing model has two `ELEMENT` dependency constraints per car.
 - 1b. A Berge-acyclic subset of the constraints that are **only on the source variables** of the dependency graph of M can be satisfied without backtracking.
2. Model L has relaxations (see a later slide) of all the other constraints of model M .

Generic Model Relaxation for LNS

Given a model M of a satisfaction or optimisation problem, produce a relaxation L of M that is suitable for LNS initialisation and iterations:

1. Model L has all the constraints of model M that are easy to satisfy simultaneously.
 - 1a. All the dependency constraints of M are easy to satisfy simultaneously, except if the dependency graph is cyclic: need for cycle elimination (see extra slide).
Example: Our car-sequencing model has two `ELEMENT` dependency constraints per car.
 - 1b. A Berge-acyclic subset of the constraints that are **only on the source variables** of the dependency graph of M can be satisfied without backtracking. 
2. Model L has relaxations (see a later slide) of all the other constraints of model M .

1b. Neighbourhood Constraints

A **set of neighbourhood constraints** has as scope a subset of the **source** variables of the dependency graph of the given model and induces a Berge-acyclic hypergraph (as that makes them satisfiable simultaneously without backtracking).

1b. Neighbourhood Constraints

A set of neighbourhood constraints has as scope a subset of the source variables of the dependency graph of the given model and induces a Berge-acyclic hypergraph (as that makes them satisfiable simultaneously without backtracking).

Running example: Car Sequencing

$\{GCC([t_1, \dots, t_6], [\text{car}, \text{motorcycle}, \text{car}]) = [2, 2, 2]\}$ is a set of neighbourhood constraints.

1b. Neighbourhood Constraints

A **set of neighbourhood constraints** has as scope a subset of the **source** variables of the dependency graph of the given model and induces a Berge-acyclic hypergraph (as that makes them satisfiable simultaneously without backtracking).

Running example: Car Sequencing

$\{GCC([t_1, \dots, t_6], [\text{🚗}, \text{🚗}, \text{🚗}]) = [2, 2, 2]\}$ is a set of neighbourhood constraints.

We greedily find a set of non-overlapping constraints c_i with the lexicographically maximum pair $\langle \sum_i \text{rank}(c_i), \sum_i |\text{scope}(c_i)| \rangle$, where 'rank' induces a total order.

We use: $\text{rank}(\text{CUMULATIVE}) > \text{rank}(\text{CIRCUIT}) > \text{rank}(\text{ALLDIFFERENT})$.

Generic Model Relaxation for LNS — Recapitulation

1. Copy the constraints of the given model that are easy to satisfy simultaneously
2. Relax the remaining constraints of the given model (as they are difficult to satisfy)

Generic Model Relaxation for LNS — Recapitulation

1. Copy the constraints of the given model that are easy to satisfy simultaneously
2. Relax the remaining constraints of the given model (as they are difficult to satisfy)
Example: `SLIDINGSUM(0, 2, 3, [ $r_1, \dots, r_6$ ])` triggers backtracking & needs relaxation.

Generic Model Relaxation for LNS — Recapitulation

1. Copy the constraints of the given model that are easy to satisfy simultaneously
2. Relax the remaining constraints of the given model (as they are difficult to satisfy) 
Example: `SLIDINGSUM(0, 2, 3, [ $r_1, \dots, r_6$ ])` triggers backtracking & needs relaxation.

2. Constraint Relaxation — Roadmap

We relax all the non-dependency non-neighbourhood constraints of the given model.

2. Constraint Relaxation — Roadmap

We relax all the non-dependency non-neighbourhood constraints of the given model.

Our relaxation scheme for constraints has two ingredients:

2. Constraint Relaxation — Roadmap

We relax all the non-dependency non-neighbourhood constraints of the given model.

Our relaxation scheme for constraints has two ingredients:

2a. Constraint softening (see the next slide) makes a constraint easier to satisfy.

2. Constraint Relaxation — Roadmap

We relax all the non-dependency non-neighbourhood constraints of the given model.

Our relaxation scheme for constraints has two ingredients:

- 2a. **Constraint softening** (see the next slide) makes a constraint easier to satisfy.
- 2b. **Non-failing propagation** (see a later slide) keeps some of the original propagation.

2a. Constraint Softening

A constraint $c(\mathcal{X})$ can be reformulated as a **soft constraint** $\hat{c}(\mathcal{X}) = v$, which is a dependency constraint such that the new variable v , called a **violation variable**, is 0 when $c(\mathcal{X})$ is satisfied, else > 0 , say proportionally to the amount of violation of $c(\mathcal{X})$.

2a. Constraint Softening

A constraint $c(\mathcal{X})$ can be reformulated as a **soft constraint** $\hat{c}(\mathcal{X}) = v$, which is a dependency constraint such that the new variable v , called a **violation variable**, is 0 when $c(\mathcal{X})$ is satisfied, else > 0 , say proportionally to the amount of violation of $c(\mathcal{X})$.

We say that $\hat{c}(\mathcal{X}) = v$ is a **softening** of $c(\mathcal{X})$.

2a. Constraint Softening

A constraint $c(\mathcal{X})$ can be reformulated as a **soft constraint** $\hat{c}(\mathcal{X}) = v$, which is a dependency constraint such that the new variable v , called a **violation variable**, is 0 when $c(\mathcal{X})$ is satisfied, else > 0 , say proportionally to the amount of violation of $c(\mathcal{X})$.

We say that $\hat{c}(\mathcal{X}) = v$ is a **softening** of $c(\mathcal{X})$.

Examples

- A softening of $x = y$ is $|x - y| = v$.

2a. Constraint Softening

A constraint $c(\mathcal{X})$ can be reformulated as a **soft constraint** $\hat{c}(\mathcal{X}) = v$, which is a dependency constraint such that the new variable v , called a **violation variable**, is 0 when $c(\mathcal{X})$ is satisfied, else > 0 , say proportionally to the amount of violation of $c(\mathcal{X})$.

We say that $\hat{c}(\mathcal{X}) = v$ is a **softening** of $c(\mathcal{X})$.

Examples

- A softening of $x = y$ is $|x - y| = v$.
- A softening of any constraint γ is $1 - [\gamma] = v$, where $[\gamma] \in \{0, 1\}$ reifies γ .

2b. Non-Failing Propagation

[Björdal et al. @ CP'20]

A **non-failing propagator** prunes until it **would** empty some domain, leaving at least one value in the domain. Instead of triggering failure, it is withdrawn from subsequent inference phases, but restored on backtracking.

2. Constraint Relaxation — Putting Things Together

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

2. Constraint Relaxation — Putting Things Together

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

2. L has relaxations of all the non-dependency non-neighbourhood constraints of M .

2. Constraint Relaxation — Putting Things Together

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

2. L has relaxations of all the non-dependency non-neighbourhood constraints of M .

The **relaxation** of a constraint $c(\mathcal{X})$ has two parts:

[Björddal et al. @ CP'20]

2. Constraint Relaxation — Putting Things Together

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

2. L has relaxations of all the non-dependency non-neighbourhood constraints of M .

The **relaxation** of a constraint $c(\mathcal{X})$ has two parts: [Björddal et al. @ CP'20]

- Add a softening $\hat{c}(\mathcal{X}) = v$ of $c(\mathcal{X})$ to L , with a normal propagator.

2. Constraint Relaxation — Putting Things Together

Given a model M of a satisfaction or optimisation problem,
produce a relaxation L of M that is suitable for LNS initialisation and iterations:

2. L has relaxations of all the non-dependency non-neighbourhood constraints of M .

The **relaxation** of a constraint $c(\mathcal{X})$ has two parts: [Björddal et al. @ CP'20]

- Add a softening $\hat{c}(\mathcal{X}) = v$ of $c(\mathcal{X})$ to L , with a normal propagator.
- Add also $c(\mathcal{X})$ to L , with a non-failing propagator until a first no-violation solution is found, and with a normal propagator thereafter.

Model Softening

Upon all softening – with violation variables v_i – the objective of the model is changed:

Model Softening

Upon all softening – with violation variables v_i – the objective of the model is changed:

- A satisfaction model becomes the optimisation model where $\sum_i v_i$ is minimised.

Model Softening

Upon all softening – with violation variables v_i – the objective of the model is changed:

- A satisfaction model becomes the optimisation model where $\sum_i v_i$ is minimised.
- An optimisation model minimising (wlog) the objective variable o becomes the optimisation model where the pair $\langle \sum_i v_i, o \rangle$ is lexicographically minimised.

Multi-Armed Bandit

One **arm** for each configuration of our two LNS hyperparameters:

- LNS freeze heuristic: {random, propagation-guided (PG), reverse propagation-guided (RPG), variable-objective (VO), cost-impact guided (CIG), variable-relationship-guided (VRG)}.

Multi-Armed Bandit

One **arm** for each configuration of our two LNS hyperparameters:

- LNS freeze heuristic: {random, propagation-guided (PG), reverse propagation-guided (RPG), variable-objective (VO), cost-impact guided (CIG), variable-relationship-guided (VRG)}.
- Dependency-based curation of the LNS freeze set: {on, off} because curation does not always pay off.

Multi-Armed Bandit

One **arm** for each configuration of our two LNS hyperparameters:

- LNS freeze heuristic: {random, propagation-guided (PG), reverse propagation-guided (RPG), variable-objective (VO), cost-impact guided (CIG), variable-relationship-guided (VRG)}.
- Dependency-based curation of the LNS freeze set: {on, off} because curation does not always pay off.

The **reward** of an arm is its number of non-worsening (violating) solutions during 5 s.

Multi-Armed Bandit

One **arm** for each configuration of our two LNS hyperparameters:

- LNS freeze heuristic: {random, propagation-guided (PG), reverse propagation-guided (RPG), variable-objective (VO), cost-impact guided (CIG), variable-relationship-guided (VRG)}.
- Dependency-based curation of the LNS freeze set: {on, off} because curation does not always pay off.

The **reward** of an arm is its number of non-worsening (violating) solutions during 5 s.

The **probability** of pulling an arm is under the normalised exponential function, also known as soft-max or Boltzmann exploration: it increases with its reward.

Multi-Armed Bandit

One **arm** for each configuration of our two LNS hyperparameters:

- LNS freeze heuristic: {random, propagation-guided (PG), reverse propagation-guided (RPG), variable-objective (VO), cost-impact guided (CIG), variable-relationship-guided (VRG)}.
- Dependency-based curation of the LNS freeze set: {on, off} because curation does not always pay off.

The **reward** of an arm is its number of non-worsening (violating) solutions during 5 s.

The **probability** of pulling an arm is under the normalised exponential function, also known as soft-max or Boltzmann exploration: it increases with its reward.

Parallelism: 7 threads of the bandit + 1 CP thread, all solution-sharing.

Evaluation — Contestants



Baseline: only Gecode-based solution-sharing portfolio solvers:

Evaluation — Contestants



Baseline: only Gecode-based solution-sharing portfolio solvers:

- $\text{Gecode-par} = \{\text{Gecode}, \text{random-LNS}\}$

[MiniZinc Challenge, since 2017]

Evaluation — Contestants



Baseline: only Gecode-based solution-sharing portfolio solvers:

- $\text{Gecode-par} = \{\text{Gecode}, \text{random-LNS}\}$ [MiniZinc Challenge, since 2017]
- $\text{Gecode-Dexter} = \{\text{Gecode}\} \cup \{f\text{-LNS} \mid f \in \{\text{random}, \text{PG}, \text{RPG}, \text{VO}, \text{CIG}, \text{VRG}\}\}$
[Leander, 2024]

Evaluation — Contestants



Baseline: only Gecode-based solution-sharing portfolio solvers:

- $\text{Gecode-par} = \{\text{Gecode}, \text{random-LNS}\}$ [MiniZinc Challenge, since 2017]
- $\text{Gecode-Dexter} = \{\text{Gecode}\} \cup \{f\text{-LNS} \mid f \in \{\text{random}, \text{PG}, \text{RPG}, \text{VO}, \text{CIG}, \text{VRG}\}\}$
[Leander, 2024]

Ablation study:

Evaluation — Contestants



Baseline: only Gecode-based solution-sharing portfolio solvers:

- $\text{Gecode-par} = \{\text{Gecode}, \text{random-LNS}\}$ [MiniZinc Challenge, since 2017]
- $\text{Gecode-Dexter} = \{\text{Gecode}\} \cup \{f\text{-LNS} \mid f \in \{\text{random}, \text{PG}, \text{RPG}, \text{VO}, \text{CIG}, \text{VRG}\}\}$
[Leander, 2024]

Ablation study:

- $\text{Gecode-depLNS} = \text{improved Gecode-Dexter} + \text{dependency-based model relaxation}$

Evaluation — Contestants



Baseline: only Gecode-based solution-sharing portfolio solvers:

- $\text{Gecode-par} = \{\text{Gecode}, \text{random-LNS}\}$ [MiniZinc Challenge, since 2017]
- $\text{Gecode-Dexter} = \{\text{Gecode}\} \cup \{f\text{-LNS} \mid f \in \{\text{random}, \text{PG}, \text{RPG}, \text{VO}, \text{CIG}, \text{VRG}\}\}$
[Leander, 2024]

Ablation study:

- $\text{Gecode-depLNS} = \text{improved Gecode-Dexter} + \text{dependency-based model relaxation}$
- $\text{Gecode-depLNS-MAB} = \text{Gecode-depLNS} + \text{parallelised multi-armed bandit}$

Evaluation — Benchmark



We used existing MiniZinc models and instances for:

Evaluation — Benchmark



We used existing MiniZinc models and instances for:

- car sequencing
- nurse rostering
- rotating workforce rostering
- job shop problem $\rightsquigarrow$ few dependency constraints, so Gecode-par should win
- steel mill slab design
- travelling salesperson problem with time windows $\rightsquigarrow$ cyclic dependency graph

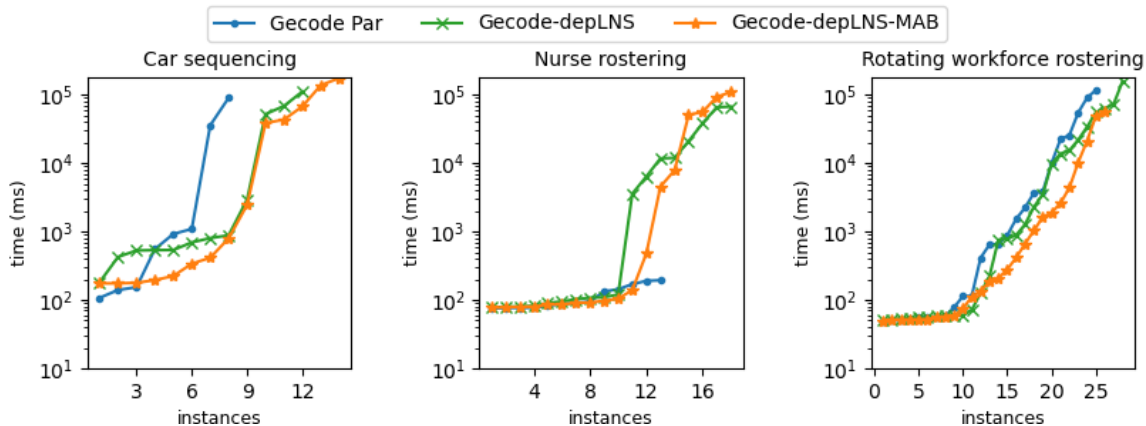
Evaluation — Experiment Setup

- We made 10 independent runs per instance.
- Each run was allocated 3 minutes.
- The **score** (of a function used in other LNS papers) ranges over the interval 0..100:

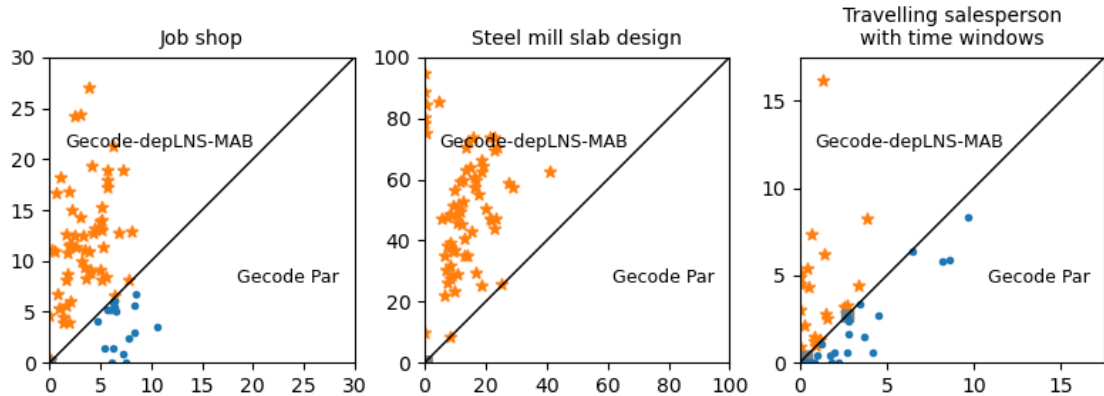
Evaluation — Experiment Setup

- We made 10 independent runs per instance.
- Each run was allocated 3 minutes.
- The **score** (of a function used in other LNS papers) ranges over the interval 0..100:
 - A low score is better than a high score.
 - A score of 0 is given when best: matching the best objective value of all runs of all solvers.
 - A score of 100 is given when worst, including when no solution was found.

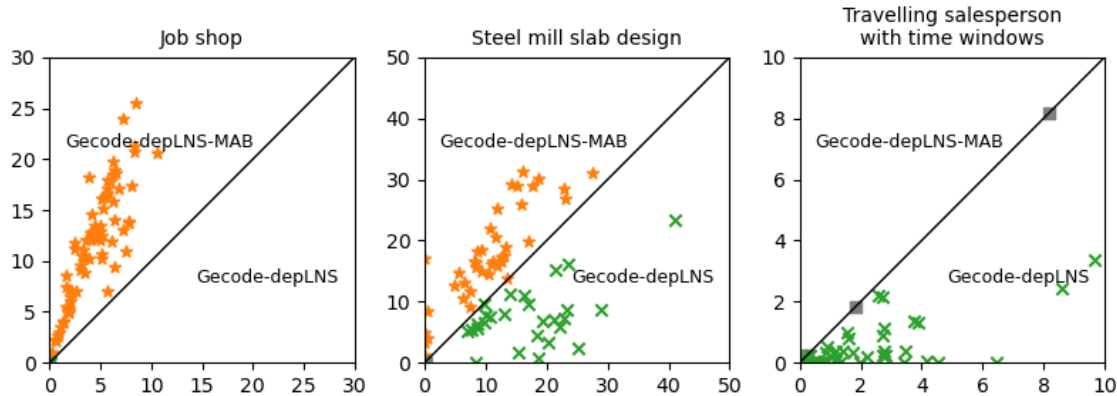
Evaluation — Number of Solved CSP Instances



Evaluation — Score Comparison for COP Instances



Evaluation — Score Comparison for COP Instances



Conclusion

Summary:

- Definition of an automated, generic LNS-friendly relaxation of any (MiniZinc) model, by exploiting the dependency graph & semantics of its constraints.

Conclusion

Summary:

- Definition of an automated, generic LNS-friendly relaxation of any (MiniZinc) model, by exploiting the dependency graph & semantics of its constraints.
- Addition of a parallelised multi-armed bandit (MAB) on two LNS hyperparameters.

Conclusion

Summary:

- Definition of an automated, generic LNS-friendly relaxation of any (MiniZinc) model, by exploiting the dependency graph & semantics of its constraints.
- Addition of a parallelised multi-armed bandit (MAB) on two LNS hyperparameters.
- Implementation as solution-sharing LNS + CP portfolio solvers, with & without MAB.

Conclusion

Summary:

- Definition of an automated, generic LNS-friendly relaxation of any (MiniZinc) model, by exploiting the dependency graph & semantics of its constraints.
- Addition of a parallelised multi-armed bandit (MAB) on two LNS hyperparameters.
- Implementation as solution-sharing LNS + CP portfolio solvers, with & without MAB.

Future work:

- Tweaks of the MAB: better reward strategy? better hyperparameters?

Conclusion

Summary:

- Definition of an automated, generic LNS-friendly relaxation of any (MiniZinc) model, by exploiting the dependency graph & semantics of its constraints.
- Addition of a parallelised multi-armed bandit (MAB) on two LNS hyperparameters.
- Implementation as solution-sharing LNS + CP portfolio solvers, with & without MAB.

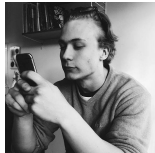
Future work:

- Tweaks of the MAB: better reward strategy? better hyperparameters?
- Exploitation of the dependency graph for other solving technologies: CP? LCG?

Acknowledgements



Gustav
Björdal



Dexter
Leander



Otto
Hammar



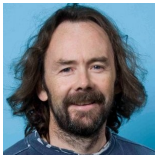
Pierre
Flener



Justin
K.
Pearson



Mikael
Zayenz
Lagerkvist



Peter
J. Stuckey



Guido
Tack



Vetenskapsrådet
Swedish Research Council

Grants 2018-04813 and 2025-03939

Thank You!

Thank you for your attention: any questions?

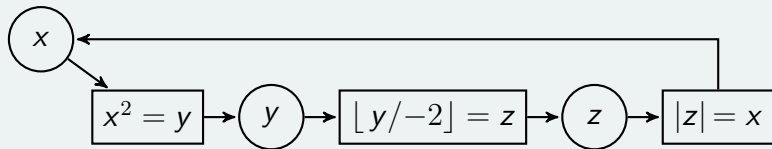
1a. Dependency Constraints, upon Cycle Elimination

Copy all the dependency constraints of the given model, as they are easy to satisfy, but first eliminate the cycles in the dependency graph

[FKL, Pierre Flener, & Pearson @ JAIR, 2025]

as the dependency constraints within a cycle are difficult to satisfy simultaneously.

Example: cycle elimination in a dependency graph



A dependency graph with one cycle.

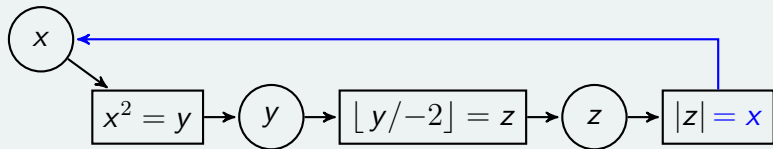
1a. Dependency Constraints, upon Cycle Elimination

Copy all the dependency constraints of the given model, as they are easy to satisfy, but first eliminate the cycles in the dependency graph

[FKL, Pierre Flener, & Pearson @ JAIR, 2025]

as the dependency constraints within a cycle are difficult to satisfy simultaneously.

Example: cycle elimination in a dependency graph



Let's break the cycle at the dependency constraint that functionally defines variable x .

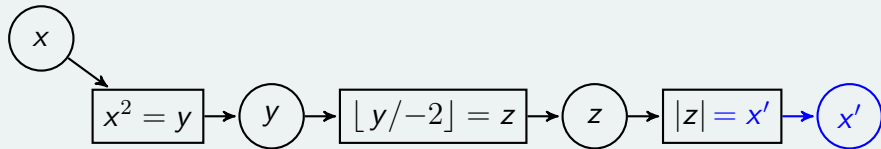
1a. Dependency Constraints, upon Cycle Elimination

Copy all the dependency constraints of the given model, as they are easy to satisfy, but first eliminate the cycles in the dependency graph

[FKL, Pierre Flener, & Pearson @ JAIR, 2025]

as the dependency constraints within a cycle are difficult to satisfy simultaneously.

Example: cycle elimination in a dependency graph



We replace that constraint $|z| = x$ with $|z| = x'$, by introducing the variable x' .

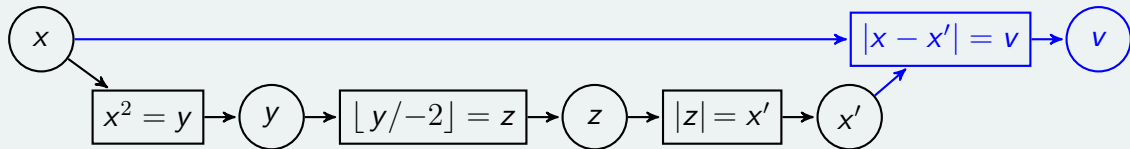
1a. Dependency Constraints, upon Cycle Elimination

Copy all the dependency constraints of the given model, as they are easy to satisfy, but first eliminate the cycles in the dependency graph

[FKL, Pierre Flener, & Pearson @ JAIR, 2025]

as the dependency constraints within a cycle are difficult to satisfy simultaneously.

Example: cycle elimination in a dependency graph



We add a **softening** of the now required $x = x'$, by introducing the violation variable v .

Multi-way Dependency Constraints

Some dependency constraints have multiple sets of input and output variables. Each such constraint could be represented by multiple vertices in the dependency graph.

Example

The constraint

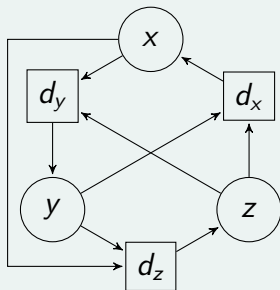
$$x + y = z \quad (d_z)$$

can be reformulated as

$$z - y = x \quad (d_x)$$

and as

$$z - x = y \quad (d_y)$$



Conditional Dependency Constraints

Some constraints are dependency constraints under specific circumstances.

Example: the TABLE constraint

For each car i , we can replace its two ELEMENT constraints with a TABLE constraint:

$$\text{TABLE}([t_i, r_i, w_i], \begin{bmatrix} [\text{truck}, \checkmark, \checkmark], \\ [\text{motorcycle}, \checkmark, \text{thumbs down}], \\ [\text{car}, \text{thumbs down}, \checkmark], \\ [\text{scooter}, \text{thumbs down}, \text{thumbs down}] \end{bmatrix})$$

The values in the first column of the parameter table are distinct.

Therefore, this is a dependency constraint that functionally defines r_i and w_i from t_i .
(Conversely, this is a dependency constraint that functionally defines t_i from r_i and w_i .)