Sequence Variables: A Constraint Programming Computational Domain for Routing and Sequencing

Augustin Delecluse
KU Leuven, augustin.delecluse@kuleuven.be,

Pierre Schaus
UCLouvain, pierre.schaus@uclouvain.be,

Pascal Van Hentenryck
GATECH, pascal.vanhentenryck@isye.gatech.edu,

Abstract. Constraint Programming (CP) offers an intuitive, declarative framework for modeling Vehicle Routing Problems (VRP), yet classical CP models based on successor variables cannot always deal with optional visits or insertion based heuristics. To address these limitations, this paper formalizes sequence variables within CP. Unlike the classical successor models, this computational domain handle optional visits and support insertion heuristics, including insertion-based Large Neighborhood Search. We provide a clear definition of their domain, update operations, and introduce consistency levels for constraints on this domain. An implementation is described with the underlying data structures required for integrating sequence variables into existing trail-based CP solvers. Furthermore, global constraints specifically designed for sequence variables and vehicle routing are introduced. Finally, the effectiveness of sequence variables is demonstrated by simplifying problem modeling and achieving competitive computational performance on the Dial-a-Ride Problem.

1. Introduction

The Vehicle Routing Problem (VRP) has many variants (Braekers et al. 2016). Constraint Programming (CP) is a widely used approach for solving them (Kilby and Shaw 2006), as its declarative modeling paradigm allows for flexible adaptation to constraints and objectives. However, existing CP approaches struggle to handle optional visits and do not support insertion-based search strategies, which are crucial to quickly obtain high-quality solutions. To address this, we introduce a sequence-based computational domain that enables both optional visits and insertion-based searches. As a motivating example, we consider the Dial-A-Ride Problem (DARP). The DARP is to schedule a fleet of K vehicles to fulfill a set of transportation requests R, where each request specifies a pickup location and a drop-off location. The objective is to minimize the total distance traveled while respecting various constraints, such as vehicle capacity, time windows, and user

ride-time limits. The model and the insertion-based search are given next, illustrating the $CP =$ *model* + *search* paradigm. The precise semantics of each constraint does not need to be understood in detail at this stage.

The model The declarative CP model is

$$\min \sum_{k \in K} \boldsymbol{Dist}_k \tag{1}$$

subject to:

$$\text{Distance}(\boldsymbol{Route}_k, \boldsymbol{d}, \boldsymbol{Dist}_k) \qquad \forall k \in K \tag{2}$$

$$\text{TransitionTimes}(\boldsymbol{Route}_k, (\boldsymbol{Time}), \boldsymbol{s}, \boldsymbol{d}) \qquad \forall k \in K \tag{3}$$

$$\text{Cumulative}(\boldsymbol{Route}_k, \boldsymbol{r}^+, \boldsymbol{r}^-, \boldsymbol{q}, c) \qquad \forall k \in K \tag{4}$$

$$\sum_{k \in K} \mathcal{R}_v(\boldsymbol{Route}_k) = 1 \qquad \forall v \in V \tag{5}$$

$$\boldsymbol{Time}_{r_i^-} - \boldsymbol{Time}_{r_i^+} - \boldsymbol{s}_{r_i^+} \le t_i \qquad \forall r_i \in R \tag{6}$$

$$\boldsymbol{Time}_{\omega_k} - \boldsymbol{Time}_{\alpha_k} \le t^d \qquad \forall k \in K \tag{7}$$

It relies on an insertion-based sequence variable $Route_k$ for each vehicle $k \in K$. It represents the sequence of nodes visited by the vehicle starting at the depot and ending at the depot. The objective consists in minimizing the sum of the distances traveled (1). The travel distance of each vehicle $k \in K$ is linked in (2) to a variable $\boldsymbol{Dist}_k$, while constraint (3) enforces visits during valid time windows. The capacity available within a vehicle is constrained through (4). The constraint (5) ensures that every node is visited exactly once. Lastly, (6) and (7) enforce the maximum ride time and the maximum duration of the route, respectively.

The search In Constraint Programming (CP), a backtracking depth-first search (DFS) is typically used. This search can be customized by defining a branching procedure, which is responsible for generating child nodes at each step. In vehicle routing, two main search strategies are commonly used. The nearest neighbor heuristic sequentially appends the closest unvisited node to the current sequence. The insertion-based heuristic selects a node and inserts it in the position that minimizes the cost increase. As shown in Rosenkrantz et al. (1977), insertion-based construction heuristics are particularly effective at avoiding the long edge effect, where a very long final connection is required to close a tour, a common issue with nearest neighbor strategies. An insertion-based branching

procedure for the DARP is presented in Algorithm 1. This procedure generates the alternative child nodes of a search node. Those are obtained by considering all the possible insertions of a chosen request in the different routes.

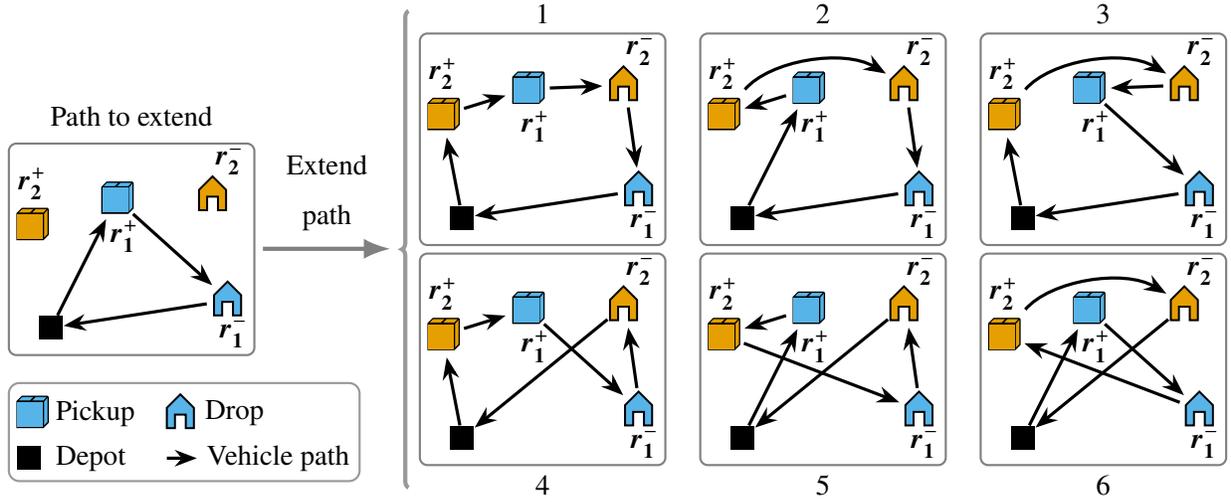**Algorithm 1:** Creation of the branching points for the DARP.

---

1 **if** $\bigwedge_{k \in K}$ **$Route_k$**.isFixed() **then**

2 | **return** solution

3 $r_i \leftarrow \underset{r_j \in R \mid r_j \text{ not yet inserted}}{\text{argmin}} \sum_{k \in K}$ **$Route_k$**.nInsert(r_j^+) $\times$ **$Route_k$**.nInsert(r_j^-)

4 $branches \leftarrow \{\}$

5 **for** $k \in K$ **do**

6 | $I^+ \leftarrow$ **$Route_k$**.getInsert(r_i^+)

7 | **for** $p^+ \in I^+$ **do**

8 | | $I^- \leftarrow$ **$Route_k$**.getInsertAfter(r_i^-, p^+) $\cup \{r_i^+\}$

9 | | **for** $p^- \in I^-$ **do**

10 | | | $branches \leftarrow branches \cup \big\{ ($**$Route_k$**.insert($p^+, r_i^+$) $\wedge$ **$Route_k$**.insert(p^-, r_i^-))$\big\}$

11 sort $branches$ by increasing heuristic value

12 **return** $branches$

The branching proceeds in two steps. First it prioritizes the unvisited request r_i with the fewest remaining feasible insertions (line 3). This is in line with the *first-fail* principle, which aims to create the shallowest possible search tree. Given this request, all insertion points I^+ for its pickup r_i^+ in a given sequence variable **$Route_k$** are retrieved (line 6). Suitable insertions I^- for its delivery r_i^- in the current path are also retrieved, including the pickup r_i^+ as a predecessor candidate (line 8). All valid combinations of predecessors are considered as an alternative decision to explore (line 10). Since DFS is used, the most promising insertions should be explored first. This can, for instance, be achieved by sorting all candidate insertions based on their impact on tour length, prioritizing those that minimize the increase in distance (line 11). A solution is reached when no further insertions are possible in any vehicle: all paths are fixed (line 2).

EXAMPLE 1. The example refers to Figure 1 with only one vehicle. A route serves both visits of r_1. The request r_2 is selected, resulting in six possible sequences for inserting its two visits, each corresponding to a possible branching decision in the search tree. These insertion options are sorted by their cost.

Figure 1 Paths generated by Algorithm 1 from an initial one (left). Numbers show which are considered first.

A DFS using the branching of Algorithm 1 can be used to find an initial solution by stopping at the first feasible solution encountered. A limited discrepancy DFS search, keeping only the leftmost few branches, as was done in Jain and Van Hentenryck (2011) can also be used to quickly discover good solutions. It can also be used into a Large Neighborhood Search (LNS) strategy, as introduced in Shaw (1998), whose main iteration is depicted in Algorithm 2. A set of requests to relax from a previous solution S is first selected (line 1). The paths represented in the previous solution are then enforced, except for nodes belonging to relaxed requests, which are omitted (line 7). A search is then performed (line 8) to insert those remaining relaxed requests, leading to a new solution $\hat{S}$. This process is repeated until a given stopping criterion is met.

Algorithm 2: A LNS iteration for DARP

Input: $S = [S_1, \ldots, S_K]$: a sequence of visits for each vehicle

1   $\mathcal{R} \leftarrow$ relaxedRequests(S)

2   $\mathcal{V}^{\mathcal{R}} \leftarrow \bigcup_{r_i \in \mathcal{R}} \{r_i^+, r_i^-\}$

3   **for** $k \in K$ **do**

4      $\textbf{\textit{Route}}_k \leftarrow$ empty sequence variable `// empty the vehicle path`

5   **for** $v \in S_k$ **do** `// iterate over the previous path `S_k`, in order`

6   **if** $v \notin \mathcal{V}^{\mathcal{R}}$ **then**

7            $\textbf{\textit{Route}}_k$.insertAtEnd($v$) `// append the visit in same order as in `$S$

8   $S \leftarrow$ Solve problem starting with $\textbf{\textit{Route}}$ `// Using branching from Algorithm 1`

1.1. Limitations of Existing Approaches in CP

The model and search strategy introduced for the DARP in the previous section rely on insertion-based sequence variables. However, the most common approach for modeling a VRP in CP is based on the successor model, which does not support insertion-based search strategies. A successor model uses an integer variable $succ_i$ for representing the direct successor of customer i in a tour. To ensure that all customers are visited on a tour, the (Hamiltonian) *circuit* global constraints (Lauriere 1978) and its weighted variants (Benchimol et al. 2012) can be used. This kind of model has two main limitations. First, representing the optional nature of visits with the successor model is not straightforward[1]. Second, at the search level, the first solution along the leftmost branch of the search tree typically relies on a *nearest neighbor* heuristic. These nearest neighbor heuristics tend to add small edges near the root of the search tree, but as decisions progress, they tend to add very long edges at the end, making it difficult to quickly find good solutions with CP.

An advanced CP alternative to the successor model, which enables dealing with optional visits more naturally, is to use the *head-tail sequence variables* implemented in IBM CP Optimizer (Laborie et al. 2018, 2009). According to the documentation and in the context of VRP, the domain consists of two growing sequences of nodes. One is the head (prefix) and the other the tail (suffix) of nodes to be visited in that order. The domain update operations are to append a node to the tail, at the head, or merging the two to form the final sequence. These variables handle optionality by design since not every node needs to be added in the sequence. However, similarly to the successor model, a heuristic on those variables would also rely on a nearest neighbor strategy.

Since both the successor and the head-tail sequence models do not easily support insertion heuristics, a new type of variable, the insertion-based sequence variable, was recently introduced in (Thomas et al. 2020, Delecluse et al. 2022) to address these limitations. These variables have a domain composed of the possible insertions for each node within a partial sequence. They thus consume more space than the head-tail sequence variables. A summary of the main properties of the modeling variables for VRP in CP is given in Table 1.

In the rest of the article, insertion-based sequence variables are simply denoted *sequence variables*. This work significantly extends the previous work on sequence variables from Thomas et al. (2020), Delecluse et al. (2022) by:

---

[1] Some modeling languages (Nethercote et al. 2007, Boussemart et al. 2016, Hentenryck 2002) offer a subcircuit version that allows a node to be excluded from the circuit by assigning it a self-loop ($succ_i = i$) but this complicates the model semantics, as constraints need to be aware that self-loops are permitted and designate an unvisited node

Table 1   CP Variables characteristics.

	Type of variable		
Feature	Successor	Head-tail sequence	Insertion-based sequence
Nearest neighbor heuristics	✓	✓	✓
Insertion based heuristics			✓
Optional visits		✓	✓
Memory complexity on VRP with n nodes and k vehicles	$O(n^2)$	$O(k \cdot n)$	$O(k \cdot n^2)$

- **Formalizing the computational model for sequence variables**, providing a robust theoretical foundation.

- **Introducing consistency levels**, including the novel concept of *insert consistency*.

- **Proposing an implementation** along with the underlying data structures to integrate sequence variables into existing trail-based CP solvers.

- **Developing global constraints** for modeling VRP with sequence variables.

- **Demonstrating the effectiveness of sequence variables** on the Dial-a-Ride problem.

This paper is organized as follows. Section 2 provides a brief overview of VRP solving with CP, and reviews work related to sequence variables. Section 3 details the computational model for sequence variables, including domain representations, operations and consistency definitions. Section 6 discusses the global constraints tailored for sequence variables and their integration with vehicle routing problems. Section 7 describes search strategies to efficiently solve CP models involving sequence variables. Finally, Section 8 presents experimental results on the Dial-A-Ride Problem, validating the theoretical contributions and practical implementations.

2.   Related Work

Constraint Programming (CP) is a declarative paradigm where problems are modeled through variables and constraints. Solutions are then found via a systematic search combined with constraint propagation. In CP, each variable x has an associated domain $D(x)$ of possible values, and constraints actively filter these domains by removing inconsistent values. At each node of the search tree, a variable is selected, a value from its domain is chosen and the constraints are propagated until a fix-point is reached. The search backtracks if an inconsistency is detected. We refer to Michel et al. (2021) for a more detailed background on CP solving.

The successor model described in Kilby and Shaw (2006) is commonly used for modeling VRPs in CP, where each node is associated with an integer variable representing its next visited node. An attempt to extend this model to support insertion-based search was proposed in Kilby et al.

(2000), where a reversible data structure called *insertion schedule* represents the direct predecessor and successor of the nodes already inserted by the branching procedure. A channeling mechanism is then used to maintain consistency with a new set of variables, called the *insertion variables* π, which represent the possible insertion positions for visit i. Although this approach enables advanced insertion procedures, such as ours, it does not provide a well-defined sequence variable with a computation domain that can be exploited to filter external constraints.

Another CP alternative relies on dedicated structured domains. They have been successfully introduced for representing set variables in (Gervet 1997, Puget 1993) and graph variables in (Dooms et al. 2005). These two domains rely on the concept of subset-bounds, which maintain a lower bound of mandatory elements and an upper bound of potential elements in the set or graph. In Pesant et al. (1997), the authors propose an extension of the successor model by introducing, for each visit i, two sets: $\mathcal{A}_i$ and $\mathcal{B}_i$, representing respectively the set of (not necessarily direct) predecessors and successors in the tour. These sets are maintained through channeling constraints with the successor and predecessor variables. This formulation makes it possible to add precedence constraints to the model and, by introducing them dynamically, one can also implement insertion-based search strategies. However, as with the *insertion variables* π of Kilby et al. (2000), this approach does not provide a well-defined sequence variable with a computation domain that can be directly exploited by external constraints.

The closest related work to the insertion-based sequence variable domain presented in this paper is the head-tail sequence variable explained in the previous section and introduced in (Laborie et al. 2018, 2009). A similar approach is implemented in OR-Tools by Google (Perron and Furnon 2019), which also provides sequence variable modeling. While primarily targeted at scheduling problems, these sequence variables have been successfully applied to solving hybrid routing scheduling problems in works such as (Liu et al. 2018, Cappart et al. 2018). A related idea of growing prefix sequences was utilized for path variable representation in traffic engineering problems in the field of computer networks (Hartert et al. 2015).

The approach proposed in this paper is more flexible, allowing elements to be inserted at arbitrary positions of the sequence. This insertion capability can be viewed as a domain implementation of the insertion graph concept introduced in Bent and Van Hentenryck (2004). This domain representation was previously introduced in (Thomas et al. 2020), along with its simplified variant that excludes required visits, as detailed in (Delecluse et al. 2022). This paper builds upon and refines these earlier works, providing a cleaner and more detailed version. It formally defines the computation domain and its semantics, and introduces the notion of consistency.

3. Sequence Variable

We first introduce some notations before defining the domain of sequence variables.

Notations A sequence $\overrightarrow{S}$ is defined as an ordered set of nodes belonging to a graph, without repetition. Let $\overrightarrow{S}$ be a sequence with the form $\overrightarrow{S} = \overrightarrow{S}_1 \cdot v_1 \cdot \overrightarrow{S}_2$ ($\overrightarrow{S}_1$ and $\overrightarrow{S}_2$ being sequences, possibly empty). An insertion operation $\underset{(v_1, v_2)}{\longmapsto}$ defined by the pair of nodes (v_1, v_2) produces a new sequence $\overrightarrow{S}' = \overrightarrow{S}_1 \cdot v_1 \cdot v_2 \cdot \overrightarrow{S}_2$. We denote with $\overrightarrow{S} \subset \overrightarrow{S}'$ that the sequence $\overrightarrow{S}$ is a subsequence from $\overrightarrow{S}'$, where $\overrightarrow{S}'$ preserves the order from $\overrightarrow{S}$ and has strictly more nodes. Then $\overrightarrow{S}'$ is called a super sequence of $\overrightarrow{S}$. If the two sequences may be the same, the relation is written $\overrightarrow{S} \subseteq \overrightarrow{S}'$. We denote $v_i \xrightarrow{\overrightarrow{S}} v_j$ to indicate that v_i directly precedes v_j in the sequence $\overrightarrow{S}$ and $v_i \overset{\overrightarrow{S}}{\prec} v_j$ when v_i precedes (not necessarily directly) v_j in $\overrightarrow{S}$. Those relations are simply written $v_i \rightarrow v_j$ and $v_i \prec v_j$ when clear from the context. If the nodes can be the same, the relation is written $v_i \preceq v_j$. Given a sequence $\overrightarrow{S} = v_1 \ldots v_i \ldots v_n$, $\mathrm{prefix}(\overrightarrow{S}, v_i) = v_1 \ldots v_i$ and $\mathrm{suffix}(\overrightarrow{S}, v_i) = v_i \ldots v_n$. Lastly, given a node set V, a start node $\alpha \in V$ and an end node $\omega \in V$, $\mathcal{P}(V)$ denotes all sequences $\overrightarrow{S}$ over a subset of nodes V, starting at node $\alpha \in V$ and ending at node $\omega \in V$.

Domain A sequence domain denoted $\mathcal{D} \subseteq \mathcal{P}(V)$ contains a set of sequences over a subset of the nodes V without repetition, starting at node $\alpha \in V$ and ending at node $\omega \in V$. Four elementary restrictions can be imposed on a sequence domain:

1. **Require** a node to be visited, without explicitly stating the position of the node.
2. **Exclude** a node from being visited.
3. **Enforce a subsequence** $\overrightarrow{S}'$ to be included (i.e., enforce $\overrightarrow{S}' \subseteq \overrightarrow{S}$).
4. **NotBetween**, which forbids a node v_2 to be placed between two other nodes v_1 and v_3 regardless of any additional nodes that may be present between them.

Those restrictions were chosen to enable branching decisions, to allow for large neighborhood searches, and to facilitate domain filtering through constraints.

To define the sequence domain $\mathcal{D}$, we first define one subdomain for each of the four possible restrictions:

1. $\mathcal{D}^{\textcircled{r}}(R)$ is specified by a set of required nodes $R \subseteq V$. It denotes all sequences including all nodes in R. More formally,

$$\mathcal{D}^{\textcircled{r}}(R) = \left\{ \overrightarrow{S} \mid \forall v \in R : v \in \overrightarrow{S} \right\} \tag{8}$$

2. $\mathcal{D}^{\otimes}(X)$ is specified by a set of excluded nodes $X \subseteq V$. It denotes all sequences where no node in X is included. More formally,

$$\mathcal{D}^{\otimes}(X) = \left\{ \overrightarrow{S} \mid \forall v \in X : v \notin \overrightarrow{S} \right\} \tag{9}$$

3. $\mathcal{D}^{\circledS}(\vec{s})$ is specified by a partial sequence of nodes $\vec{s}$. It denotes all sequences $\vec{S}$ such that $\vec{s}$ is a subsequence of $\vec{S}$. More formally,

$$\mathcal{D}^{\circledS}(\vec{s}) = \left\{\vec{S} \mid \vec{s} \subseteq \vec{S}\right\} \tag{10}$$

4. $\mathcal{D}^{\circledn}(N)$ is specified by a set of NotBetween triples $N \subseteq V \times V \times V$. $\mathcal{D}^{\circledn}(N)$ denotes all sequences in which no NotBetween triple from N appears. More formally,

$$\mathcal{D}^{\circledn}(N) = \left\{\vec{S} \mid \forall (v_i \cdot v_j \cdot v_k) \in N : (v_i \cdot v_j \cdot v_k) \nsubseteq \vec{S}\right\} \tag{11}$$

The intersection of those four subdomains defines the sequence domain.

DEFINITION 1. A sequence domain is defined as

$$\mathcal{D}(R, X, \vec{s}, N) = \mathcal{D}^{\circledT}(R) \cap \mathcal{D}^{\circledX}(X) \cap \mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(N) \tag{12}$$

DEFINITION 2. $\mathcal{D}(R, X, \vec{s}, N)$ is said to be fixed when $|\mathcal{D}(R, X, \vec{s}, N)| = 1$.

By abuse of notations, $\mathcal{D}(R, X, \vec{s}, N)$ will be simply written $\mathcal{D}$ in the following.

EXAMPLE 2. $V = \{\alpha, v_1, v_2, v_3, \omega\}$, $R = \{\alpha, \omega, v_2\}$, $X = \{v_3\}$, $\vec{s} = \alpha \cdot v_1 \cdot \omega$, $N = \{(\alpha \cdot v_2 \cdot v_1), (v_3 \cdot v_1 \cdot v_2)\}$. $\mathcal{D} = \mathcal{D}^{\circledT}(R) \cap \mathcal{D}^{\circledX}(X) \cap \mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(N) = \{(\alpha \cdot v_1 \cdot v_2 \cdot \omega)\}$. The compositions of the subdomains are represented in Table 2. Given that $|\mathcal{D}| = 1$, the domain is fixed.

$\mathcal{P}(V)$	$\mathcal{D}^{\circledT}(R)$	$\mathcal{D}^{\circledX}(X)$	$\mathcal{D}^{\circledS}(\vec{s})$	$\mathcal{D}^{\circledn}(N)$	$\mathcal{P}(V)$	$\mathcal{D}^{\circledT}(R)$	$\mathcal{D}^{\circledX}(X)$	$\mathcal{D}^{\circledS}(\vec{s})$	$\mathcal{D}^{\circledn}(N)$
$\alpha \cdot \omega$		✓		✓	$\alpha \cdot v_3 \cdot v_1 \cdot \omega$			✓	✓
$\alpha \cdot v_1 \cdot \omega$		✓	✓	✓	$\alpha \cdot v_3 \cdot v_2 \cdot \omega$	✓			✓
$\alpha \cdot v_2 \cdot \omega$	✓	✓		✓	$\alpha \cdot v_1 \cdot v_2 \cdot v_3 \cdot \omega$	✓		✓	✓
$\alpha \cdot v_3 \cdot \omega$				✓	$\alpha \cdot v_1 \cdot v_3 \cdot v_2 \cdot \omega$	✓		✓	✓
$\boldsymbol{\alpha \cdot v_1 \cdot v_2 \cdot \omega}$	✓	✓	✓	✓	$\alpha \cdot v_2 \cdot v_1 \cdot v_3 \cdot \omega$	✓		✓	
$\alpha \cdot v_1 \cdot v_3 \cdot \omega$			✓	✓	$\alpha \cdot v_2 \cdot v_3 \cdot v_1 \cdot \omega$	✓		✓	
$\alpha \cdot v_2 \cdot v_1 \cdot \omega$	✓	✓	✓		$\alpha \cdot v_3 \cdot v_1 \cdot v_2 \cdot \omega$	✓		✓	
$\alpha \cdot v_2 \cdot v_3 \cdot \omega$	✓			✓	$\alpha \cdot v_3 \cdot v_2 \cdot v_1 \cdot \omega$	✓		✓	

Table 2 The compositions of subdomains in Example 2. A check mark in one of the last 4 columns indicates that the corresponding sequence in the first column is included within the subdomain.

Domain updates over $\mathcal{D}$ are defined by growing the sets R, X and N, adding more elements to them, or by inserting a node within the partial sequence $\vec{s}$. Notice that elementary restrictions can be combined to enforce a *between* restriction, that is forcing a node v_2 to be visited between nodes v_1, v_3. This can be achieved by adding $(\alpha \cdot v_2 \cdot v_1), (v_3 \cdot v_2 \cdot \omega)$ to the NotBetween N given that any

sequence $\vec{s} \in \mathcal{D}$ begins at α and ends at ω. Moreover, if node v_2 must be included in the sequences of the domain, v_2 can be added to the required nodes R.

Each sequence variable $\vec{s}$ is associated with a sequence domain, and represents an unknown sequence. Such a variable is particularly convenient for modeling a route from an origin node α to a destination node ω in a VRP, where the nodes visited and their ordering represents the path performed by a vehicle.

We introduce a compact encoding of the domain $\mathcal{D}$ in the next section, that only consumes $O(n^2)$ memory, with $n = |V|$, and enables efficient domain updates.

4.   An $O(n^2)$ encoding of the sequence variable domain

This section introduces a compact domain representation for the entire domain $\mathcal{D}(R, X, \vec{s}, N)$ as a set of pairs of nodes, requiring only $O(n^2)$ memory, with the limitation that for each NotBetween $(v_1 \cdot v_2 \cdot v_3) \in N$, the two extremities v_1, v_3 are within the partial sequence $\vec{s}$. This representation exploits the representation of the intersection of the subdomains. First the representation of $\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(N)$ is introduced in Section 4.1, then it is extended with the intersection of the excluded nodes subdomain $\mathcal{D}^{\circledX}(X)$ in Section 4.2 and finally with the intersection of the required nodes subdomain $\mathcal{D}^{\circledr}(R)$ in Section 4.3.

4.1.   Partial Sequence and NotBetween

A naive representation of the set of NotBetween $N \subseteq V \times V \times V$ requires a cubic space complexity. This can be reduced to a quadratic one by representing directly $\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(N)$ and by restricting N to only contains NotBetween with the extremities belonging to the partial sequence $\vec{s}$ and in the order in which they appear in $\vec{s}$: $N \subseteq \{(v_1 \cdot v_2 \cdot v_3) \mid v_1 \overset{\vec{s}}{\prec} v_3\}$. The set of NotBetween with this restriction is written $\hat{N}$. The set $\hat{N}$ can then be translated into a set of pairs of nodes (v_i, v_2) corresponding to *forbidden insertions* of v_2 just after v_i in $\vec{s}$, due to a NotBetween:

$$\mathcal{I}^{\circledf}(\vec{s}, \hat{N}) = \{(v_i, v_2) \mid (v_1 \cdot v_2 \cdot v_3) \in \hat{N} \wedge v_1 \preceq v_i \wedge v_i \prec v_3\} \tag{13}$$

THEOREM 1.   *The forbidden insertions $\mathcal{I}^{\circledf}(\vec{s}, \hat{N})$ can be used instead of $\hat{N}$ in the representation of the domain $\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(\hat{N})$.*

Proof.   One can enumerate all super sequences $\vec{s}'$ of $\vec{s}$ such that no forbidden pair of $\mathcal{I}^{\circledf}(\vec{s}, \hat{N})$ appears in that order in $\vec{s}'$:

$$\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(\hat{N}) = \{\vec{s}' \mid (\vec{s} \subseteq \vec{s}') \wedge (\forall v \in (\vec{s}' \setminus \vec{s}) : (\text{prev}(v, \vec{s}, \vec{s}'), v) \notin \mathcal{I}^{\circledf}(\vec{s}, \hat{N}))\} \tag{14}$$

where $\text{prev}(v, \vec{s}, \vec{s}')$ denotes the first node preceding v in $\vec{s}'$ and also appearing in $\vec{s}$:

$$\text{prev}(v, \vec{s}, \vec{s}') = \begin{cases} v & \text{if } v \in \vec{s} \\ \text{prev}(v_i, \vec{s}, \vec{s}') \mid v_i \xrightarrow{\vec{s}'} v & \text{otherwise .} \end{cases} \tag{15}$$

$Q.E.D.$

Thanks to Theorem 1, $\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(\hat{N})$ can be represented solely with the partial sequence $\vec{s}$ and the pairs of nodes $\mathcal{I}^{\circledf}(\vec{s}, \hat{N})$, computed in $O(n^2)$ time and also consuming $O(n^2)$ in memory.

EXAMPLE 3. Consider the following set of nodes, partial sequence and NotBetween's:

- $V = \{\alpha, v_1, v_2, v_3, \omega\}$
- $\vec{s} = \alpha \cdot v_1 \cdot \omega$
- $N = \{(\alpha \cdot v_2 \cdot v_1), (v_1 \cdot v_3 \cdot \omega)\}$.

The set of forbidden insertions is then $\mathcal{I}^{\circledf}(\vec{s}, N) = \{(\alpha, v_2), (v_1, v_3)\}$. The domain is $\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(N) = \{(\alpha \cdot v_1 \cdot \omega), (\alpha \cdot v_1 \cdot v_2 \cdot \omega), (\alpha \cdot v_3 \cdot v_1 \cdot \omega), (\alpha \cdot v_3 \cdot v_1 \cdot v_2 \cdot \omega)\}$. The composition of the sub-domains are represented in Table 3.

Table 3   **Sub-domains composition in Example 3. A check mark in one of the last 2 columns indicates that the corresponding sequence in the first column is included within the sub-domain.**

$\mathcal{P}(V)$	$\mathcal{D}^{\circledS}(\vec{s})$	$\mathcal{D}^{\circledn}(\hat{N})$	$\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(\hat{N})$	$\mathcal{P}(V)$	$\mathcal{D}^{\circledS}(\vec{s})$	$\mathcal{D}^{\circledn}(\hat{N})$	$\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(\hat{N})$
$\alpha \cdot \omega$		✓		$\boldsymbol{\alpha \cdot v_3 \cdot v_1 \cdot \omega}$	✓	✓	✓
$\boldsymbol{\alpha \cdot v_1 \cdot \omega}$	✓	✓	✓	$\alpha \cdot v_3 \cdot v_2 \cdot \omega$		✓	
$\alpha \cdot v_2 \cdot \omega$		✓		$\alpha \cdot v_1 \cdot v_2 \cdot v_3 \cdot \omega$	✓		
$\alpha \cdot v_3 \cdot \omega$		✓		$\alpha \cdot v_1 \cdot v_3 \cdot v_2 \cdot \omega$	✓		
$\boldsymbol{\alpha \cdot v_1 \cdot v_2 \cdot \omega}$	✓	✓	✓	$\alpha \cdot v_2 \cdot v_1 \cdot v_3 \cdot \omega$	✓		
$\alpha \cdot v_1 \cdot v_3 \cdot \omega$	✓			$\alpha \cdot v_2 \cdot v_3 \cdot v_1 \cdot \omega$	✓		
$\alpha \cdot v_2 \cdot v_1 \cdot \omega$	✓			$\boldsymbol{\alpha \cdot v_3 \cdot v_1 \cdot v_2 \cdot \omega}$	✓	✓	✓
$\alpha \cdot v_2 \cdot v_3 \cdot \omega$		✓		$\alpha \cdot v_3 \cdot v_2 \cdot v_1 \cdot \omega$	✓		

Domain updates Two domains updates are possible on $\mathcal{D}^{\circledS}(\vec{s}) \cap \mathcal{D}^{\circledn}(\hat{N})$: insertion of a node within $\vec{s}$, or adding a new NotBetween into $\hat{N}$. Two possible strategies to update $\mathcal{I}^{\circledf}(\vec{s}, \hat{N})$ over those domain modifications are

1. Recompute $\mathcal{I}^{\circledf}(\vec{s}, \hat{N})$, which still requires to maintain $\hat{N}$, taking $O(n^3)$ in memory.
2. Store the set $\mathcal{I}^{\circledf}(\vec{s}, \hat{N})$, consuming $O(n^2)$ in memory, and incrementally update it.

We follow the more efficient second strategy and show how to perform such updates in $O(n)$ time, keeping an $O(n^2)$ memory complexity:

- If a new NotBetween $(v_1 \cdot v_2 \cdot v_3)$ is added onto $\hat{N}$, the newly forbidden insertions are:

$$\Delta \mathcal{I}^{\textcircled{f}}(\overrightarrow{s}, (v_1 \cdot v_2 \cdot v_3)) = \{(v_i, v_2) \mid v_1 \preceq v_i \wedge v_i \prec v_3\} \tag{16}$$

  Which uses the same rule as (13). At most $O(|\overrightarrow{s}|) = O(n)$ pairs can be added in this manner.

- If the sequence $\overrightarrow{s}$ is expanded into $\overrightarrow{s}'$ through an insertion $\overrightarrow{s} \underset{(v_1, v_2)}{\longmapsto} \overrightarrow{s}'$, the newly forbidden insertions are:

$$\Delta \mathcal{I}^{\textcircled{f}}(\overrightarrow{s}, (v_1, v_2)) = \{(v_2, v_i) \mid (v_1, v_i) \in \mathcal{I}^{\textcircled{f}}(\overrightarrow{s}, \hat{N})\} \tag{17}$$

  Denoting that a node v_i that could not be placed directly after v_1 now cannot be placed directly after v_2 as well. The set (17) can be created in $O(n)$ time and contains $O(n)$ pairs.

Whenever a NotBetween $(v_1 \cdot v_2 \cdot v_3)$ is added to $\hat{N}$ while it is a subsequence of the partial sequence $((v_1 \cdot v_2 \cdot v_3) \subseteq \overrightarrow{s})$, it results in a domain wipeout. Similarly, expanding $\overrightarrow{s}$ into $\overrightarrow{s}'$ using a forbidden insertion $(v_1, v_2) \in \mathcal{I}^{\textcircled{f}}(\overrightarrow{s}, \hat{N})$ (i.e., performing $\overrightarrow{s} \underset{(v_1, v_2)}{\longmapsto} \overrightarrow{s}'$) results in a domain wipeout. Next, we show how to compute the intersection with the remaining subdomains.

4.2. Excluded Nodes

The subdomain $\mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{n}}(\hat{N})$ presented previously only accounted for a partial sequence $\overrightarrow{s}$ and a set of NotBetween's $\hat{N}$. Excluding a node v from the sequence can be reduced to adding the NotBetween $(\alpha \cdot v \cdot \omega)$ to $\hat{N}$ given that all sequences begins at α and end at ω. Therefore for a set of excluded nodes X we have:

$$\forall v \in X : (\alpha \cdot v \cdot \omega) \in \hat{N} \tag{18}$$

When an update $X \leftarrow X \cup \{v\}$ occurs, it suffices to add the NotBetween (α, v, ω), and use (16) to trigger the incremental update of $\mathcal{I}^{\textcircled{f}}(\overrightarrow{s}, \hat{N})$. Thus, there is no need to store the excluded nodes X.

4.3. Required Nodes

To take into account a set of required nodes R, we do not have other options than storing them. This requires an additional $O(n)$ memory. The entire domain $\mathcal{D}$ can be represented in $O(n^2)$ with i) the set of pairs of nodes $\mathcal{I}^{\textcircled{f}}(\overrightarrow{s}, \hat{N})$, ii) the partial sequence $\overrightarrow{s}$, and iii) the set of required nodes R. Together, those elements implicitly represent all super sequences $\overrightarrow{s}'$ of $\overrightarrow{s}$ such that no forbidden pair of $\mathcal{I}^{\textcircled{f}}(\overrightarrow{s}, \hat{N})$ appears in that order in $\overrightarrow{s}'$ and all the required nodes are present in s':

$$\mathcal{D} = \{\overrightarrow{s}' \mid (\overrightarrow{s} \subseteq \overrightarrow{s}') \wedge (\forall v \in (\overrightarrow{s}' \setminus \overrightarrow{s}) : (\mathrm{prev}(v, \overrightarrow{s}, \overrightarrow{s}'), v) \notin \mathcal{I}^{\textcircled{f}}(\overrightarrow{s}, \hat{N})) \wedge (\forall v \in R : v \in \overrightarrow{s}')\} \tag{19}$$

The next section proposes an implementation of those elements using a graph data structure.

5. Compact domain implementation

As explained in the previous section, a domain implementation requires representing the forbidden insertions $\mathcal{I}^{\oplus}(\overrightarrow{s}, \hat{N})$, the partial sequence $\overrightarrow{s}$, and the required nodes R.

Rather than representing $\mathcal{I}^{\oplus}(\overrightarrow{s}, \hat{N})$, we instead represent the complementary set $(V \times V) \setminus \mathcal{I}^{\oplus}(\overrightarrow{s}, \hat{N}) \setminus \{(v_i, v_j) \mid v_i, v_j \in \overrightarrow{s} \wedge v_i \not\rightarrow v_j\}$ as a graph $G(V, E)$. Each directed edge $(v_i, v_j) \in E$ with $v_i \in \overrightarrow{s}$ represents a possible insertion of node v_j in $\overrightarrow{s}$ just after v_i in the partial sequence $\overrightarrow{s}$. Notice that every pair of nodes in the partial sequence that do not directly follow each other are also removed from the set of edges. As the partial sequence $\overrightarrow{s}$ grows and the set of NotBetween's $\hat{N}$ grows, the set of edges E can only shrink monotonically. Eventually, when the variable is fixed, the set of edges forms a path that corresponds to the partial sequence $\overrightarrow{s}$. This complementary view enables the enumeration of all feasible insertions of a node into the partial sequence $\overrightarrow{s}$. It also allows for counting feasible insertions for each node in the partial sequence. These are two common operations used by filtering algorithms and for implementing first-fail branching heuristics.
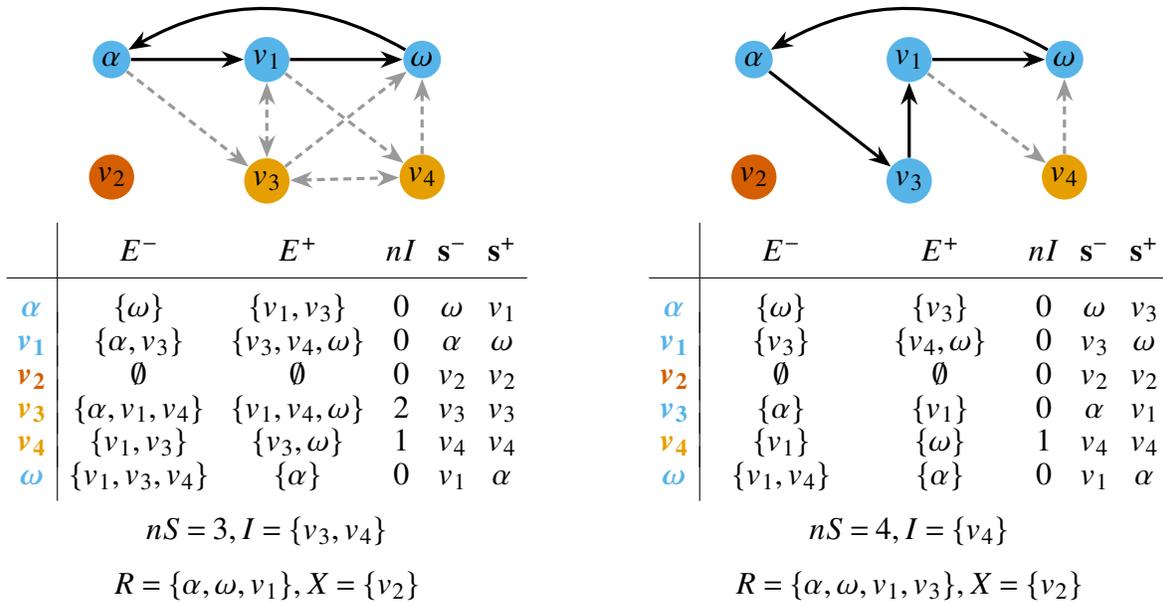
5.1. A Reversible Data Structure

The main mechanism to restore the state of the domains and constraints on backtrack in CP solver is called trailing (we refer to Michel et al. (2021) for an introduction to trailing in CP solvers). To be integrated in a standard trailed-based CP solver, the compact domain implementation also needs to be reversible. The graph $G(V, E)$ is implemented as an adjacency list where each list is a reversible set of nodes and the partial sequence $\overrightarrow{s}$ uses a reversible successor array. In more detail, the implementation relies on the following reversible data structures, most of which were already present in Thomas et al. (2020), Delecluse et al. (2022):

- The partial sequence $\overrightarrow{s}$ is encoded using a successor array of reversible integers $\mathbf{s}^+$. It stores a pointer to the current successor of each node belonging to the partial sequence $\overrightarrow{s}$, and an element without a successor (i.e., a node $v \in V \setminus \overrightarrow{s}$) points towards itself (self-loop). Similarly, an additional array of reversible integers $\mathbf{s}^-$ tracks the current predecessors of each node.
- The set of edges E is maintained by two adjacency sets per node $v \in V$: E_v^- (incoming) and E_v^+ (outgoing) edges: $E_v^- = \{(v_i, v) \mid v_i \in V, (v_i, v) \in E\}$, $E_v^+ = \{(v, v_i) \mid v_i \in V, (v, v_i) \in E\}$. Those sets reversible sparse-sets introduced in de Saint-Marcq et al. (2013), allowing deletion and state restoration in constant time in a trail-based solver.
- A set I, tracking the insertable nodes $\{v \in V \mid v \notin \overrightarrow{s} \wedge v \notin X\}$, as a reversible sparse-set.
- The size of the partial sequence $\overrightarrow{s}$ is tracked by a reversible integer nS.

- Each $v_j \in V$ has a counter nI_j tracking insertions, updated with domain changes, aiding heuristics (e.g., retrieving the node with fewest insertions).

- Given that R and X are disjointed and subsets of V, we use a sparse-set with two reversible size markers as introduced in de Saint-Marcq et al. (2013), ensuring removal of nodes and state restoration in constant time, and enabling iteration over R, X in $\mathcal{O}(|R|), \mathcal{O}(|X|)$, respectively. We also identify with $P = V \setminus X \setminus R$, the *possible* nodes, being not required nor excluded.

The data structures used for implementing a compact domain are depicted in Figure 2.

Figure 2 Compact domain implementation. On the left, the partial sequence $\overrightarrow{s}$ and the graphs $G(V, E)$ are shown. Below them is a table showing the edges E^-, E^+, the counters of insertions nI and the successors s^+ and the predecessors s^- of the nodes (only relevant for nodes $v \in \overrightarrow{s}$). The right part shows the domain after performing an insertion with (α, v_3), extending the partial sequence.

	E^-	E^+	nI	$\mathbf{s}^-$	$\mathbf{s}^+$
α	$\{\omega\}$	$\{v_1, v_3\}$	0	ω	v_1
v_1	$\{\alpha, v_3\}$	$\{v_3, v_4, \omega\}$	0	α	ω
v_2	$\emptyset$	$\emptyset$	0	v_2	v_2
v_3	$\{\alpha, v_1, v_4\}$	$\{v_1, v_4, \omega\}$	2	v_3	v_3
v_4	$\{v_1, v_3\}$	$\{v_3, \omega\}$	1	v_4	v_4
ω	$\{v_1, v_3, v_4\}$	$\{\alpha\}$	0	v_1	α

$$nS = 3, I = \{v_3, v_4\}$$

$$R = \{\alpha, \omega, v_1\}, X = \{v_2\}$$

	E^-	E^+	nI	$\mathbf{s}^-$	$\mathbf{s}^+$
α	$\{\omega\}$	$\{v_3\}$	0	ω	v_3
v_1	$\{v_3\}$	$\{v_4, \omega\}$	0	v_3	ω
v_2	$\emptyset$	$\emptyset$	0	v_2	v_2
v_3	$\{\alpha\}$	$\{v_1\}$	0	α	v_1
v_4	$\{v_1\}$	$\{\omega\}$	1	v_4	v_4
ω	$\{v_1, v_4\}$	$\{\alpha\}$	0	v_1	α

$$nS = 4, I = \{v_4\}$$

$$R = \{\alpha, \omega, v_1, v_3\}, X = \{v_2\}$$

Invariants on the domain are presented in the Appendix A and the implementation of the update operations in Appendix B. Some coherence is enforced on the domain representation. For instance, given that all nodes in the partial sequence will always be visited, we enforce $\overrightarrow{s} \subseteq R$. We also exploit the counter of insertion nI_j of node $v_j \in V$ to automatically insert a node when it is required and has only one insertion remaining (similarly to Kilby et al. (2000)), or to exclude v_j from the sequence when no more insertion remains for it. This process is also detailed in Appendix B.

5.2. API

A sequence domain is fixed whenever, for each node that does not belong to the partial sequence, no insertion remains. The worst-case time complexity for constructing a sequence is thus $O(|E|)$. In the worst case, all edges are removed except those that form the sequence.

Table 4 lists all query operations in a sequence variable domain, along with their associated time complexities. Domain updates are described in Table 5.

Table 4   Queries on a compact sequence domain.

Operation	Description	Complexity
isFixed()	Returns true if there are no remaining insertions	$O(1)$
isMember(v_i)	Returns true if $v_i \in \overrightarrow{s}$	$O(1)$
isRequired(v_i)	Returns true if the node v_i is required	$O(1)$
isExcluded(v_i)	Returns true if the node v_i is excluded	$O(1)$
isPossible(v_i)	Returns true if the node v_i is possible	$O(1)$
isInsertable(v_i)	Returns true if the node v_i is insertable	$O(1)$
getNext(v_i)	Returns the successor $\mathbf{s}_i^+$ of node v_i	$O(1)$
getPrev(v_i)	Returns the predecessor $\mathbf{s}_i^-$ of node v_i	$O(1)$
nInsert(v_i)	Returns the number of feasible insertions for v_i	$O(1)$
nMember()	Returns the length of the partial sequence	$O(1)$
getMember()	Enumerates nodes in the partial sequence $\overrightarrow{s}$	$\Theta(|\overrightarrow{s}|)$
getRequired()	Enumerates the required nodes R	$\Theta(|R|)$
getExcluded()	Enumerates the excluded nodes X	$\Theta(|X|)$
getPossible()	Enumerates the possible nodes P	$\Theta(|P|)$
getInsertable()	Enumerates the insertable nodes I	$\Theta(|I|)$
getEdgesTo(v_i)	Enumerates E_i^-	$\Theta(|E_i^-|)$
getEdgesFrom(v_i)	Enumerates E_i^+	$\Theta(|E_i^+|)$
canInsert(v_i, v_j)	Returns true if (v_i, v_j) is a feasible insertion	$O(1)$
getInsert(v_j)	Enumerates the insertions of v_j: $\{v_i \mid v_i \in E_j^- \wedge \text{canInsert}(v_i, v_j)\}$	$\Theta(\min(|\overrightarrow{s}|, |E_j^-|))$
getInsertAfter(v_j, p)	Enumerates the insertions of v_j after p: $\{v_i \mid p \prec v_i \wedge \text{canInsert}(v_i, v_j)\}$	$O(|\overrightarrow{s}|)$

Table 5   Updates on a compact sequence domain.

Operation	Update	Complexity
notBetween(v_i, v_j, v_k)	$N \leftarrow N \cup (v_i \cdot v_j \cdot v_k)$	$\begin{cases} O(|I|) & \text{if } v_i \to v_k \\ O(n) & \text{otherwise} \end{cases}$
insert(v_i, v_j)	$\underset{(v_i, v_j)}{\longmapsto}$	$\Theta(|E_j^-|)$
insertAtEnd(v_i)	$\underset{(\text{getPrev}(\omega), v_i)}{\longmapsto}$	$\Theta(|E_i^-|)$
require(v_i)	$R \leftarrow R \cup \{v_i\}$	$O(|E_i^-|)$
exclude(v_i)	$X \leftarrow X \cup \{v_i\}$	$O(|E_i^-|)$

5.3. Visit of nodes as boolean variables

Given the set of mandatory nodes R, one can easily create a binary variable $\mathcal{R}_i(\overrightarrow{s})$ for any node $v_i \in V$, telling if the node is visited (value 1: $v_i \in R$) or not (value 0: $v_i \notin R$) by a sequence variable $\overrightarrow{s}$ with a compact domain. In CP, this can be implemented as a *view* over the compact sequence domain, making the usage of such variables cheap once a sequence variable has been created, as in Schulte and Tack (2013), Van Hentenryck and Michel (2014), Michel et al. (2021). Fixing a Boolean variable $\mathcal{R}_i(\overrightarrow{s})$ to 1 may automatically insert the node v_i within the partial sequence if only one insertion point remains for it, as explained in the Appendix B.

Thanks to the usage of those binary variables, one can easily enforce logical constraints over sequence variables. For instance, to force a sequence $\overrightarrow{s}$ to visit at least n nodes ($\sum_{v_i \in V} \mathcal{R}_i(\overrightarrow{s}) \geq n$), enforce two nodes $v_i, v_j \in V$ to always be visited together ($\mathcal{R}_i(\overrightarrow{s}) = \mathcal{R}_j(\overrightarrow{s})$), etc. More complex constraints, with specific propagators, are presented next.

Table 6   **Correspondence between operations on a boolean variable $\mathcal{R}_i(Sq)$ defined over a node v_i in a sequence variable $\overrightarrow{s}$. The left part shows queries over the domains, while the right part shows updates of the domains.**

Boolean variable		Sequence variable	Boolean variable		Sequence variable
$\lvert \mathcal{D}(\mathcal{R}_i(\overrightarrow{s})) \rvert = 1$	$\Longleftrightarrow$	$\neg \overrightarrow{s}.\mathrm{isPossible}(v_i)$	$\mathcal{D}(\mathcal{R}_i(\overrightarrow{s})) \leftarrow \{\mathit{true}\}$	$\Longleftrightarrow$	$\overrightarrow{s}.\mathrm{require}(v_i)$
$\mathit{false} \in \mathcal{D}(\mathcal{R}_i(\overrightarrow{s}))$	$\Longleftrightarrow$	$\neg \overrightarrow{s}.\mathrm{isRequired}(v_i)$	$\mathcal{D}(\mathcal{R}_i(\overrightarrow{s})) \leftarrow \{\mathit{false}\}$	$\Longleftrightarrow$	$\overrightarrow{s}.\mathrm{exclude}(v_i)$
$\mathit{true} \in \mathcal{D}(\mathcal{R}_i(\overrightarrow{s}))$	$\Longleftrightarrow$	$\neg \overrightarrow{s}.\mathrm{isExcluded}(v_i)$			

6. Global Constraints

Many useful global constraints and their filtering algorithms can be defined on sequence variables. To limit the scope of this paper, we focus on those required by a wide range of Vehicle Routing Problem (VRP) applications, such as the Dial-a-Ride Problem. Properly characterizing the amount of filtering theoretically is of great interest. Similar to the notions of bound-consistency and domain consistency for global constraints involving integer variables (see Michel et al. (2021)), various consistency levels can be defined for sequence variables.

Consistency of a constraint on a sequence variable.

DEFINITION 3. A constraint C over a sequence domain $\mathcal{D}(R, X, \overrightarrow{s}, N)$ is *insert consistent*, if and only if for every remaining insertion (v_1, v_2) we have $\mathcal{D}(R, X, \overrightarrow{s}', N) \cap C \neq \emptyset$ where $\overrightarrow{s} \underset{(v_1, v_2)}{\longmapsto} \overrightarrow{s}'$.

DEFINITION 4. Given a sequence domain $\mathcal{D}(R, X, \overrightarrow{s}, N)$, we define its relaxed sequence domain as $\mathcal{D}(\{v \mid v \in \overrightarrow{s}\}, X, \overrightarrow{s}, N)$, which considers that only nodes in the partial sequence are required.

DEFINITION 5. A constraint C over a sequence domain $\mathcal{D}(R, X, \overrightarrow{s}, N)$ is *relaxed insert consistent*, if and only if for every remaining insertion (v_1, v_2) on $\mathcal{D}$ we have $\mathcal{D}(\{v \mid v \in \overrightarrow{s}'\}, X, \overrightarrow{s}', N) \cap C \neq \emptyset$ where $\overrightarrow{s} \underset{(v_1, v_2)}{\longmapsto} \overrightarrow{s}'$.

Given that $\{v \mid v \in \overrightarrow{s}\} \subseteq R$, the *relaxed-insert consistency* property for a constraint C is thus a relaxed form of *insert consistency*. The filtering algorithms that we introduce are relatively basic and aim to reach *relaxed-insert consistency* rather than *insert consistency*, which may be NP-hard to reach in polynomial time for some constraints.

6.1. Distance

The Distance constraint is used to represent the travel length in a sequence of nodes. It links an integer variable *Dist* to the traveled distance between nodes in a SV, through transitions defined in a matrix $\boldsymbol{d} \in \mathbb{Z}^{|V| \times |V|}$ satisfying the triangular inequality (20).

$$\text{Distance}(\overrightarrow{S}, \boldsymbol{d}, Dist) \leftrightarrow \sum_{v_i \xrightarrow{\overrightarrow{s}} v_j} \boldsymbol{d}_{i,j} = Dist \tag{20}$$

Filtering The filtering is presented in Algorithm 3. First, it computes the traveled distance over the partial sequence $\overrightarrow{s}$ (line 1). If the SV is fixed, this fixes the value of the integer variable *Dist* (line 3). Otherwise, this is used to set its lower bound (line 5) and compute the length of the largest insertion still possible (line 6). All insertions for every insertable node $v_j \in I$ are then looked, and if the cost for inserting a node v_j between two consecutive nodes v_i and v_k is too high, it is removed (lines 10 to 12). In the worst case, the filtering empties all the edges from E, except those in the current sequence $\overrightarrow{s}$, resulting in a time complexity of $O(|E|)$.

Algorithm 3 follows a structure used in most filtering algorithms over sequence variables. The partial sequence $\overrightarrow{s}$ is first traversed and potentially used to filter other variables. Then, all insertions are inspected and possibly removed if they cannot be performed.

Regarding constraints in the next sections, the reader is referred to Delecluse et al. (2022), Thomas et al. (2020) and Appendix C for the presentation of filtering algorithms.

6.2. TransitionTimes

The TransitionTimes constraint is used for problems where node visits involve a service duration and are restricted by time windows, with a transition time required to move from one node to the next. More formally, each node $v_i \in V$ is attached to an integer variable **Start**$_i$ representing the start

---

Algorithm 3: Distance($\vec{s}, \boldsymbol{d}, Dist$) constraint filtering.

---

1   $length \leftarrow \displaystyle\sum_{v_i \xrightarrow{\vec{s}} v_j} \boldsymbol{d}_{i,j}$

2   **if** $\vec{s}$.isFixed() **then**

3      $Dist \leftarrow length$

4   **else**

5      $\lfloor Dist \rfloor \leftarrow \max(length, \lfloor Dist \rfloor)$

6      $maxDetour \leftarrow \lceil Dist \rceil - length$

7   **for** $v_j \in \vec{s}$.getInsertable() **do**

8   **for** $v_i \in \vec{s}$.getInsert(v_j) **do**

9            $v_k \leftarrow \vec{s}$.getNext(v_i)

10            $cost \leftarrow \boldsymbol{d}_{i,j} + \boldsymbol{d}_{j,k} - \boldsymbol{d}_{i,k}$

11   **if** $cost > maxDetour$ **then**

12               $\vec{s}$.notBetween(v_i, v_j, v_k)

---

of the service at that node and a service duration value s_i. A matrix $\boldsymbol{d} \in \mathbb{Z}^{|V| \times |V|}$ defines the transition times between elements and satisfies the triangular inequality. The definition of the constraint is:

$$\text{TransitionTimes}(\vec{s}, \textbf{Start}, s, \boldsymbol{d}) \leftrightarrow \forall v_i \overset{\vec{s}}{\prec} v_j : \textbf{Start}_i + s_i + \boldsymbol{d}_{i,j} \leq \textbf{Start}_j \tag{21}$$

We consider that waiting at a given node (i.e., reaching it before its time window without beginning the task related to it) is possible, which is why (21) uses inequalities. Moreover, the start variable $\textbf{Start}_v$ of an unvisited node $v \notin \vec{s}$ is not constrained.

6.3. Precedence

For some applications, visiting a set of nodes in a specific order is important, such as the visit of a pickup that must be done before visiting the corresponding drop-off. The Precedence constraint can be used in such scenarios, ensuring that an ordered set of nodes $\vec{o}$ appears in the same order in a sequence variable ($\vec{o}$ being a fixed sequence, not a variable). It is formally defined as

$$\text{Precedence}(\vec{s}, \vec{o}) \leftrightarrow \forall v_i \overset{\vec{o}}{\prec} v_j : v_i, v_j \in \vec{s} \implies v_i \overset{\vec{s}}{\prec} v_j \tag{22}$$

Note that some or all nodes in $\vec{o}$ may be absent from the SV. If the nodes from the set $\vec{o}$ must be all present or all absent, one can easily enforce this with $\forall v_i, v_j \in \vec{o} : \mathcal{R}_i(\vec{s}) = \mathcal{R}_j(\vec{s})$.
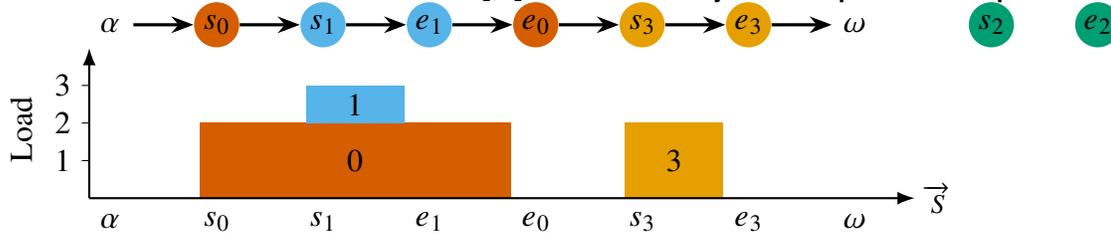
6.4. Cumulative

Some variations of VRP involve pickup and deliveries, transporting goods or people. The Cumulative constraint can be used to represent those scenarios. It ensures that going through all pickups and deliveries visited in a sequence respects an assigned capacity.

More specifically, let us define an *activity i* as a pair of nodes (s_i, e_i) corresponding to its start (pickup) and end (delivery), respectively. The set of all activities is written A. An activity $i \in A$ consumes a certain load l_i during its execution and can be in one of three states with respect to the partial sequence $\overrightarrow{s}$: *fully inserted* if $s_i \in \overrightarrow{s} \wedge e_i \in \overrightarrow{s}$, *non-inserted* if $s_i \notin \overrightarrow{s} \wedge e_i \notin \overrightarrow{s}$, and *partially inserted* otherwise (the start or the end is inserted but not both). The Cumulative constraint with a maximum capacity c, with starts s, corresponding ends e and loads l is defined as:

$$\text{Cumulative}(\overrightarrow{S}, s, e, l, c) \leftrightarrow \begin{cases} \left(\forall v \in \overrightarrow{S} : \sum_{i \in A | s_i \preceq v \prec e_i} l_i \leq c \right) \wedge \\ \left(\forall i \in A : s_i \in \overrightarrow{S} \iff e_i \in \overrightarrow{S} \right) \wedge \\ \left(\forall i \in A : \text{Precedence}(\overrightarrow{S}, (s_i, e_i)) \right) \end{cases} \tag{23}$$

This constraint implies that the start s_i of an activity $i \in A$ is visited before its end s_i ($\forall i \in A :$ Precedence$(\overrightarrow{S}, (s_i, e_i))$), and that its nodes are either all present or all absent ($\forall i \in A : \mathcal{R}_{s_i}(\overrightarrow{S}) = \mathcal{R}_{e_i}(\overrightarrow{S})$). An example of sequence over which the constraint holds is presented in Figure 3.

Figure 3   Cumulative$(\overrightarrow{S}, (s_0, s_1, s_2, s_3), (e_0, e_1, e_2, e_3), (2, 1, 1, 2), 3)$ **over a fixed sequence variable** $\overrightarrow{s} = \alpha \cdot s_0 \cdot s_1 \cdot e_1 \cdot e_0 \cdot s_3 \cdot e_3 \cdot \omega$. **The sequence is shown at the top, and the graph below shows the accumulated load for each node** $v \in \overrightarrow{s}$. **Note how nodes** s_2, e_2 **related to activity 2 are not part of the sequence.**

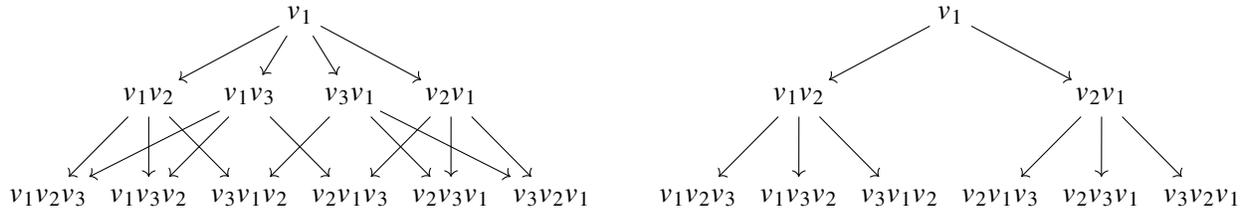## 7. Search on a sequence variable

This section presents the basic search procedure used in conjunction with one or more sequence variables to explore the search space.

7.1. Branching

Filtering of the constraints is generally not enough to terminate with fixed sequences. A search procedure is needed to explore the search space. When working with sequence variables, this corresponds to iteratively choosing an unfixed sequence and applying alternative decisions further constraining its domain, through the use of *insert* and *notBetween* operations. Once all sequences in the problem are fixed and the constraints are satisfied, a solution to the problem has been found.

As shown in the left part of Figure 4, different intermediate sequences can ultimately lead to the same one through different insertion steps. This symmetry, induced by branching decisions, can cause inefficiencies. Ideally, we would explore search trees corresponding to disjoint search spaces.

Figure 4 **Sequences created from a fully connected graph $G(V, E)$ with $V = \{\alpha, v_1, v_2, v_3, \omega\}$, where all nodes are required ($V = R$). Nodes α and ω are implicit and not shown. Left: each sequence is extended by inserting any node at each step. Right: a node is first selected for insertion, then all its feasible positions are considered for insertion before moving to the next node (first v_1, then v_2, then v_3).**

Disjoint search spaces A simple branching strategy that guarantees the generation of disjoint search spaces is the two-step *n*-ary branching:

1. **Node selection**: Choose an insertable node $v_i \in I$ within a sequence variable $\overrightarrow{S}$.

2. **Node branching**: For each insertion of v_i in $\overrightarrow{S}$, create a corresponding branching decision.

The first step is similar to the variable selection used with integer variables. It allows the integration of first-fail strategies, such as selecting nodes with the fewest possible insertion points. The second step is conceptually similar to value selection; therefore, the most promising insertions should be attempted first. It allows the integration of insertion-based heuristics, such as selecting the insertion that results in the smallest increase in tour length.

As can be observed in the right part of Figure 4, this strategy generates only distinct sequences.

Another simple binary branching strategy that offers the same guarantees is to replace the second step with only two branches. An insertion point is selected, with the insertion performed on the left branch, while the corresponding notBetween constraint is enforced on the right branch.

7.2. Large Neighborhood Search

The usage of LNS with sequence variables was already presented in Algorithm 2. In the context of VRP, this algorithm consists of relaxing some tours by removing nodes from sequences. This approach maintains partial tours, similar in spirit to the partial-order scheduling method introduced for scheduling in Godard et al. (2005). It also accurately corresponds to the original LNS presented in Shaw (1998), where partial tours are extended through insertions.

The reconstruction phase uses CP and its search capabilities to reinsert the removed nodes into the restricted problem, possibly within a time limit, before restarting the process.

The set of nodes to relax can be selected in various ways, such as randomly or with more advanced strategies based on relatedness criteria, such as geographical proximity or time-based considerations between nodes, as in Bent and Van Hentenryck (2004), Christiaens and Vanden Berghe (2020).

8. Experimental results on the Dial-A-Ride Problem

We previously presented sequence variables, several global constraints and considerations on the search procedure. We will now use that information to solve the DARP model from (1)-(7).

The state-of-the-art, to the best of our knowledge, is the Adaptive LNS proposed by Gschwind and Drexl (2019), combining the operators proposed by Ropke and Pisinger (2006) (random, worst and Shaw removal with greedy and regret based insertions) with additional relaxations, exploiting 9 removal and 5 insertion operators in total. Insertions are evaluated based on a feasibility check specific to the DARP, and obtained solutions close to the best found so far are further optimized using an adaptation of the neighborhood proposed in Balas and Simonetti (2001). For further readings on the DARP, the reader is referred to the literature review from Ho et al. (2018).

8.1. Search

We use a simple LNS that always relaxes 10 requests chosen randomly and uses Algorithm 1 for branching. The request to insert is the one that has the minimum number of insertions, defined as the sum of insertions for its pickup and drop. Insertions with a small increase in distance and high-preserved time slack are considered first, as in Jain and Van Hentenryck (2011). When considering an insertion of node v_j between nodes v_i, v_k in a vehicle, the heuristic cost is defined as:

$$C_1(d_{i,j} + d_{j,k} - d_{i,k}) - C_2(\lceil \textbf{\textit{Time}}_k \rceil - \lfloor \textbf{\textit{Time}}_i \rfloor - s_i - d_{i,j} - s_j - d_{j,k}) \qquad (24)$$

Where constants C_1, C_2 control the importance of the detour cost and the preserved time slack, respectively. The heuristic cost for inserting a request $\textbf{\textit{r}}_i \in R$ is the sum of cost for inserting its pickup $\textbf{\textit{i}}^+$ and drop $\textbf{\textit{i}}^-$.

8.2. Computational results

We compare the sequence variable approach with other approaches suitable for modeling VRP:

- **A successor model** written in Minizinc (Nethercote et al. 2007) and run with the Gecode solver (Schulte et al. 2010), which performed best on the DARP across all Minizinc backend. The underlying CP model is essentially the same as in Berbeglia et al. (2011). Both exhaustive search (**Succ**) and LNS (**Succ-LNS**) are reported.

- **OR-Tools** and its routing library, which relies on local search (Perron and Furnon 2019). All combinations of first solution strategy and local search meta heuristics in the solver have been tried, and we report the best overall configuration.

- **Hexaly** (version 13.0), a commercial solver specialized in routing (Benoist et al. 2024), using a model provided by the Hexaly team.

- **CP Optimizer (CPO)** (version 22.1.1.0), a commercial scheduling solver, whose model is written using head-tail sequence variables.

Details on the models along with the full DARP definition are in Appendix D. This list voluntary omits the state-of-the-art methods on the DARP, described earlier, as they are specialized towards the DARP only, and cannot be directly adapted to cope with other kinds of VRP. Nevertheless, we make use of the best known solutions found by such methods in our comparisons.

All experiments were conducted using two Intel® Xeon® CPU E5-2687W in single-threaded mode, with 15 minutes of run time. The sequence variable was implemented in Java using MaxiCP Schaus et al. (2024), an extended and open-source version of the MiniCP solver from Michel et al. (2021).
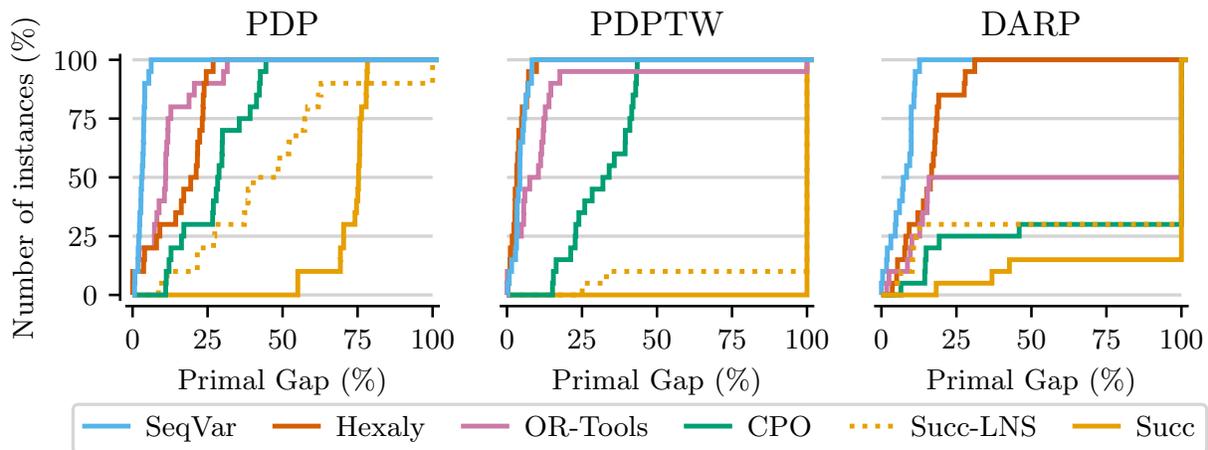
The results for the instances from Cordeau and Laporte (2003) are shown in Figure 5. It shows the proportion of instances (on the y-axis) that achieved a solution with a gap less than or equal to each value τ (on the x-axis) for each solver s: $\gamma_s(\tau) = |\{ p \in \mathcal{P} : \gamma_{p,s} \leq \tau \}|/|\mathcal{P}|$ where $\mathcal{P}$ denotes the set of problem instances, and $\gamma_{p,s}$ denotes the primal gap to the best known solution of the best solution obtained by solver s within the computation time limit. This gap is set to 1 if no incumbent solution is found within the time limit.

The approach using a sequence variable, although based on a simple LNS, consistently achieves solutions within 15% of the best-known results for the DARP. Detailed results per instance are provided in Appendix D.2.

We also report performance when (i) the ride time constraints and maximum route duration of the DARP are removed—reducing it to a Pickup and Delivery Problem with Time Windows

(PDPTW)—and (ii) when time windows are additionally removed—reducing it further to a Pickup and Delivery Problem (PDP). These variants illustrate the adaptability of the proposed models. For these cases as well, the best results are obtained using the sequence variable approach.

Figure 5   Number of instances solved for each primal gap value. Curves on the top left are the best.

9.   Conclusion

This paper enhances previous proposals of Sequence Variables, which are used in CP to tackle vehicle routing and sequencing problems. Their domain representation, implementation, and interactions with Boolean variables are proposed and formalized. They ease the modeling of complex VRP such as the DARP while staying close to the state-of-the-art in terms of performances. These variables are compatible with optional visits, insertion-based heuristics, and can be easily combined with Large Neighborhood Search, making them practical for solving complex VRP in a CP framework.

Acknowledgments

References

Balas E, Simonetti N (2001) Linear time dynamic-programming algorithms for new classes of restricted tsps: A computational study. *INFORMS journal on Computing* 13(1):56–75.

Benchimol P, Hoeve WJv, Régin JC, Rousseau LM, Rueher M (2012) Improved filtering for weighted circuit constraints. *Constraints* 17:205–233.

Benoist T, Gardi F, Julien D, Megel R (2024) Hexaly. URL https://www.hexaly.com.

Bent R, Van Hentenryck P (2004) A two-stage hybrid local search for the vehicle routing problem with time windows. *Transportation Science* 38(4):515–530.

Berbeglia G, Pesant G, Rousseau LM (2011) Checking the feasibility of dial-a-ride instances using constraint programming. *Transportation Science* 45(3):399–412.

Boussemart F, Hemery F, Lecoutre C, Sais L (2004) Boosting systematic search by weighting constraints. *ECAI*, volume 16, 96–97.

Boussemart F, Lecoutre C, Audemard G, Piette C (2016) Xcsp3: an integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398* .

Braekers K, Ramaekers K, Van Nieuwenhuyse I (2016) The vehicle routing problem: State of the art classification and review. *Computers & industrial engineering* 99:300–313.

Cappart Q, Thomas C, Schaus P, Rousseau LM (2018) A constraint programming approach for solving patient transportation problems. Hooker J, ed., *International Conference on Principles and Practice of Constraint Programming (CP)*, 490–506 (Springer).

Christiaens J, Vanden Berghe G (2020) Slack induction by string removals for vehicle routing problems. *Transportation Science* 54(2):417–433.

Cordeau JF, Laporte G (2003) A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological* 37:579–594, URL `http://dx.doi.org/10.1016/S0191-2615(02)00045-0`.

de Saint-Marcq VlC, Schaus P, Solnon C, Lecoutre C (2013) Sparse-sets for domain implementation. *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, 1–10.

Delecluse A, Schaus P, Van Hentenryck P (2022) Sequence variables for routing problems. *International Conference on Principles and Practice of Constraint Programming (CP)*.

Dooms G, Deville Y, Dupont P (2005) Cp(graph): Introducing a graph computation domain in constraint programming. *International Conference on Principles and Practice of Constraint Programming (CP)*, 211–225 (Springer).

Gervet C (1997) Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1:191–244.

Godard D, Laborie P, Nuijten W (2005) Randomized large neighborhood search for cumulative scheduling. *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 5, 81–89 (AAAI-Press).

Gschwind T, Drexl M (2019) Adaptive large neighborhood search with a constant-time feasibility test for the dial-a-ride problem. *Transportation Science* 53(2):480–491.

Hartert R, Schaus P, Vissicchio S, Bonaventure O (2015) Solving segment routing problems with hybrid constraint programming techniques. *International Conference on Principles and Practice of Constraint Programming (CP)*, 592–608 (Springer).

Hentenryck PV (2002) Constraint and integer programming in opl. *INFORMS Journal on Computing* 14(4):345–372.

Ho SC, Szeto WY, Kuo YH, Leung JM, Petering M, Tou TW (2018) A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological* 111:395–421.

Jain S, Van Hentenryck P (2011) Large neighborhood search for dial-a-ride problems. *International Conference on Principles and Practice of Constraint Programming (CP)*, 400–413 (Springer).

Kilby P, Prosser P, Shaw P (2000) A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. *Constraints* 5(4):389–414.

Kilby P, Shaw P (2006) Vehicle routing. *Handbook of Constraint Programming*, 801–836 (Elsevier).

Laborie P, Rogerie J, Shaw P, Vilím P (2009) Reasoning with conditional time-intervals. part ii: An algebraical model for resources. *FLAIRS Conference*.

Laborie P, Rogerie J, Shaw P, Vilím P (2018) Ibm ilog cp optimizer for scheduling. *Constraints* 23(2):210–250, ISSN 1383-7133, URL http://dx.doi.org/10.1007/s10601-018-9281-x.

Lauriere J (1978) A language and a program for stating and solving combinatorial problems. *Artificial Intelligence* 10(1):29–127, URL http://dx.doi.org/10.1016/0004-3702(78)90029-2.

Liu C, Aleman DM, Beck JC (2018) Modelling and solving the senior transportation problem. *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, 412–428 (Springer).

Michel L, Schaus P, Van Hentenryck P (2021) Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation* 13(1):133–184, URL http://dx.doi.org/10.1007/s12532-020-00190-7.

Nethercote N, Stuckey PJ, Becket R, Brand S, Duck GJ, Tack G (2007) Minizinc: Towards a standard cp modelling language. *International Conference on Principles and Practice of Constraint Programming (CP)*, 529–543 (Springer).

Perron L, Furnon V (2019) Or-tools. URL https://developers.google.com/optimization/.

Pesant G, Gendreaul M, Rousseau JM (1997) Genius-cp: A generic single-vehicle routing algorithm. *International Conference on Principles and Practice of Constraint Programming (CP)*, 420–434 (Springer).

Puget JF (1993) Set constraints and cardinality operator: Application to symmetrical combinatorial problems. *Third Workshop on Constraint Logic Programming–WCLP93*, 211.

Ropke S, Pisinger D (2006) An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science* 40(4):455–472.

Rosenkrantz DJ, Stearns RE, Lewis PM II (1977) An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing* 6(3):563–581.

Savelsbergh MW (1985) Local search in routing problems with time windows. *Annals of Operations Research* 4(1):285–305.

Schaus P, Derval G, Delecluse A, Michel L, Hentenryck PV (2024) Maxicp: A constraint programming solver for scheduling and vehicle routing. URL `https://github.com/aia-uclouvain/maxicp`.

Schulte C, Tack G (2013) View-based propagator derivation. *Constraints* 18:75–107.

Schulte C, Tack G, Lagerkvist MZ (2010) Modeling and programming with gecode .

Shaw P (1998) Using constraint programming and local search methods to solve vehicle routing problems. *International Conference on Principles and Practice of Constraint Programming (CP)*, 417–431 (Springer).

Thomas C, Kameugne R, Schaus P (2020) Insertion sequence variables for hybrid routing and scheduling problems. *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, 457–474 (Springer).

Van Hentenryck P, Michel L (2014) Domain views for constraint programming. *International Conference on Principles and Practice of Constraint Programming (CP)*, 705–720 (Springer).

Appendix A: Domain Implementation Details

A.1. Initialization

When initialized, a compact sequence domain is defined on the set of nodes V belonging to the graph $G(V, E)$, and its successor array $\mathbf{s}^+$ contains one entry per node $v \in V$. The two starting and ending nodes, α and ω, are identified upon the initialization of the domain. These nodes constitute the initial sequence of nodes represented by the domain, which form a cycle in the successor array.

$$\mathbf{s}_\alpha^+ = \omega \wedge \mathbf{s}_\omega^- = \alpha \wedge \mathbf{s}_\omega^+ = \alpha \wedge \mathbf{s}_\alpha^- = \omega \tag{25}$$

Note that a link from the end node ω to the first node α is encoded, thus representing a cycle instead of a path, to ease invariant encoding in further equations. The other nodes point to themselves as self-loops. At initialization, the edge set forms a complete graph, with the exception of edges that originate at ω and end at α: $E = \{(v_1, v_2) \mid v_1, v_2 \in V : v_1 \neq \omega \wedge v_2 \neq \alpha \wedge v_1 \neq v_2\} \cup \{(\omega, \alpha)\}$. The set of nodes that may be inserted is defined as $I = V \setminus \{\alpha, \omega\}$

A.2. Invariants

First, we introduce one predicate that will be instrumental in defining the consistency invariants. It determines (in constant time) if a node v_i belongs to the partial sequence (*i.e.* if $v_i \in \vec{s}$ holds) by verifying if it is not a self-loop:

$$\text{isMember}(v_i) \equiv \mathbf{s}_i^+ \neq v_i. \tag{26}$$

The lower-level consistency invariant expressed on the data structures are given next. We first identify the invariants describing the channeling between some data structures.

$$\forall v_i, v_j \in V : v_i \in E_j^- \iff v_j \in E_i^+ \tag{27}$$

$$\forall v_i, v_j \in V : \mathbf{s}_i^+ = v_j \iff \mathbf{s}_j^- = v_i \tag{28}$$

The invariant (27) ensures that each edge (v_i, v_j) appears twice in the data structures: one for the adjacency set of the incoming node, and one for the outgoing node. the invariant (28) ensures that for any two nodes v_i and v_j in the graph, v_i is the predecessor of v_j if and only if v_j is the successor of v_i. It guarantees a consistent and mutual relationship between successor and predecessor links for every node in the graph. In addition to channeling, specific invariants are used to maintain the counters previously introduced.

$$nS = |\{v_i \mid v_i \in V \wedge \text{isMember}(v_i)\}| \tag{29}$$

$$\forall v_j \in I : nI_j = |\{v_i \mid v_i \in E_j^- \wedge \text{isMember}(v_i)\}| \tag{30}$$

$$\forall v_i \in V : (nI_i \geq 1) \leftrightarrow (v_i \in I) \tag{31}$$

The length of the partial sequence is tracked in (29). Invariant (30) tracks how many insertions are feasible for a given node $v_j \in I$. This counter is used to ensure that every node $v_j \in I$ has at least one insertion possible (31): otherwise the node is not insertable and thus has 0 insertion.

Given the array of successors $\mathbf{s}^+$, one can define the set of nodes reachable from a circuit containing node $v_i \in V$:

$$\text{circuit}(v_i) = \text{circuit}(v_i, \emptyset) \tag{32}$$

$$\text{circuit}(v_i, S) = \begin{cases} S & \text{if } v_i \in S \\ \text{circuit}(\mathbf{s}_i^+, S \cup \{v\}) & \text{otherwise} \end{cases} \tag{33}$$

Intuitively from (32), $\text{circuit}(v_i)$ gives all nodes in the sequence from α to ω if v_i is in the sequence; otherwise it returns a set containing only v_i. This is done by following recursively the pointers $\mathbf{s}^+$ of the successor array. Using this definition, the implementation invariants are as follows.

$$\mathbf{s}_\omega^+ = \alpha \wedge \mathbf{s}_\alpha^- = \omega \tag{34}$$

$$\forall v_i, v_j \in V, v_i \neq v_j : \left(\mathbf{s}_i^+ = v_j \implies v_j \in E_i^+ \right) \wedge \left(\mathbf{s}_j^- = v_i \implies v_i \in E_j^- \right) \tag{35}$$

$$\forall v_i, v_j \in V, v_i \neq v_j : \mathbf{s}_i^+ \neq \mathbf{s}_j^+ \tag{36}$$

$$\forall v_i \in V : \text{isMember}(v_i) \iff \text{circuit}(v_i) = \text{circuit}(\alpha) \tag{37}$$

$$\forall v_i, v_j \in V, v_i \neq v_j : \neg\text{isMember}(v_i) \wedge \neg\text{isMember}(v_j) \implies v_i \in E_j^- \tag{38}$$

$$\forall v_i, v_j, v_k \in V, v_i \neq v_j \neq v_k : \mathbf{s}_i^+ = v_k \implies \left(v_i \in E_j^- \iff v_k \in E_j^+ \right) \tag{39}$$

Invariant (34) ensures that the successor of the last node ω being visited always points toward the first node α. Invariant (35) enforces that the current successor of a node $v_i \in \vec{s}$ exists within the outgoing edges of the node v_i. Invariants (36) and (37) ensure that only one sub-circuit is encoded within the successor array. All successors must be different, and if the successor of a node $v_i \in V$ is set (*i.e.* v_i belongs to the partial sequence) then v_i belongs to the circuit of the first node α (and the last node ω given that $\text{circuit}(\alpha) = \text{circuit}(\omega)$). Invariant (38) enforces that all insertable nodes form a clique. Finally, (39) ensures that the two edges $(v_i, v_j), (v_j, v_k)$ needed to insert a node $v_j \in I$ after node v_i (with v_k being the current successor of v_i) are either both absent or both present.

Lastly, some invariants interact specifically with the required R and excluded nodes X. Moreover, the implementation may modify itself the set of required nodes R, so that it always contains nodes from the partial sequence: $\vec{s} \subseteq R$, given that nodes who are part of the partial sequence are always visited in all sequences from the domain. The invariants are:

$$v_i \in X \iff E_i^- = \emptyset \iff E_i^+ = \emptyset \tag{40}$$

$$\forall v_i \in V : \text{isMember}(v_i) \implies v_i \in R \tag{41}$$

$$\forall v_i \in R \cap I : nI_i > 1 \tag{42}$$

Invariant (40) ensures that an excluded node has no edge attached to it. Invariant (41) captures the fact that nodes who are part of the partial sequence $\vec{s}$ are always visited, and thus considered as mandatory. Finally, invariant (42) guarantees that required nodes not part of the sequence have at least two insertions remaining. Otherwise, if only one insertion remained for such a node, it would directly be used to add the node to the partial sequence $\vec{s}$.

Appendix B: Domain updates

B.1.   Insertion

Algorithm 4 is used to perform an Sq.insert(v_1, v_2) operation on a compact sequence domain Sq, provided that (v_1, v_2) is a feasible insertion. The inserted node v_2 is first marked as required, removed from the insertable nodes and the size nS of the partial sequence $\vec{s}$ is increased (lines 1 to 4). Then, every node v_i linked to v_2 is inspected (line 6). If a node v_i belongs to both the partial sequence $\vec{s}$ and to the ingoing edges of v_2, it could previously be used to perform an insertion using (v_i, v_2). Given that v_2 is already inserted, such edges are removed (lines 8 to 11). Otherwise, if v_i does not belong to $\vec{s}$, two situations may occur: either v_i cannot be placed directly after v_1, and thus cannot be placed directly after v_2 either, according to (17) (lines 13 and 14); or the node v_i can be inserted directly after v_1 and can now be inserted directly after v_2 as well, increasing its counter nI_i (line 16). Lastly, the edges, successor and predecessor of v_1, v_2 and v_3 (the previous successor of v_1) are updated to reflect the insertion (lines 17 to 19).

Algorithm 4: Sq.insert(v_1, v_2)

Input: Sq: compact sequence domain, (v_1, v_2) feasible insertion to perform

1   $R \leftarrow R \cup \{v_2\}$

2   $I \leftarrow I \setminus \{v_2\}$

3   $nI_2 \leftarrow 0$

4   $nS \leftarrow nS + 1$

5   $v_3 \leftarrow \mathbf{s}_1^+$

6   **for** $v_i \in E_2^-$ **do**

7   **if** Sq.isMember(v_i) **then**

8   **if** $v_i \neq v_1$ **then**

9         $E_i^+ \leftarrow E_i^+ \setminus \{v_2\}, E_2^- \leftarrow E_2^- \setminus \{v_i\}$

10         $v_j \leftarrow \mathbf{s}_i^+$

11         $E_2^+ \leftarrow E_2^+ \setminus \{v_j\}, E_j^- \leftarrow E_j^- \setminus \{v_2\}$

12   **else if not** Sq.canInsert(v_1, v_i) **then**

13       $E_2^+ \leftarrow E_2^+ \setminus \{v_i\}, E_i^- \leftarrow E_i^- \setminus \{v_2\}$

14       $E_i^+ \leftarrow E_i^+ \setminus \{v_2\}, E_2^- \leftarrow E_2^- \setminus \{v_i\}$

15   **else**

16       $nI_i \leftarrow nI_i + 1$

17   $\mathbf{s}_1^+ \leftarrow v_2, \mathbf{s}_2^+ \leftarrow v_3$

18   $\mathbf{s}_3^- \leftarrow v_2, \mathbf{s}_2^- \leftarrow v_1$

19   $E_1^+ \leftarrow E_1^+ \setminus \{v_3\}, E_3^- \leftarrow E_3^- \setminus \{v_1\}$

In the implementation, if (v_1, v_2) does not define a feasible insertion, two situations may occur:

1. If v_2 is already within the partial sequence and lies after v_1 (*i.e.* $v_1, v_2 \in \overrightarrow{s} \wedge v_1 \prec v_2$), the insertion is considered as having already been performed. Nothing happens in this case.

2. Otherwise, v_2 cannot be inserted. This corresponds to a domain wipeout.

B.2. NotBetween

The algorithm 5 performs a $Sq.\text{notBetween}(v_1, v_2, v_3)$ operation on a compact sequence domain Sq. It iterates over the nodes v_i between the nodes v_1 and v_3 (line 1), removes the edges allowing to insert v_2 after v_i, and decrements its counter of insertion nI_2 (lines 4 to 6). If the insertion counter reaches 0, the node must be excluded due to (31), therefore removing the node v_2 from the set of insertable nodes I and marking it as excluded (lines 8 and 9). In this case, all edges passing through the node are removed. In contrast, if the counter of insertion reaches 1 and the node is required (line 14), it is automatically inserted at its only remaining insertion (lines 15 and 16).

Algorithm 5: $Sq.\text{notBetween}(v_1, v_2, v_3)$

Input: Sq: compact sequence domain, v_1, v_3 two nodes in the partial sequence $\overrightarrow{s}$ between which node $v_2 \in V \setminus \overrightarrow{s}$ cannot appear.

1   **for** $v_i \in \overrightarrow{s} \mid v_1 \preceq v_i \prec v_3$ **do**

2   **if** $Sq.\text{canInsert}(v_i, v_2)$ **then**

3             $v_j \leftarrow \mathbf{s}_i^+$

4             $E_2^- \leftarrow E_2^- \setminus \{v_i\}, E_i^+ \leftarrow E_i^+ \setminus \{v_2\}$

5             $E_j^- \leftarrow E_j^- \setminus \{v_2\}, E_2^+ \leftarrow E_2^+ \setminus \{v_j\}$

6             $nI_2 \leftarrow nI_2 - 1$

7   **if** $nI_2 = 0$ **then**

8                 $X \leftarrow X \cup \{v_2\}$

9                 $I \leftarrow I \setminus \{v_2\}$

10   **for** $v_k \in I$ **do**

11                   $E_k^- \leftarrow E_k^- \setminus \{v_2\}, E_k^+ \leftarrow E_k^+ \setminus \{v_2\}$

12                $E_2^- \leftarrow \emptyset, E_2^+ \leftarrow \emptyset$

13   **return**

14   **if** $nI_2 = 1$ **and** $v_2 \in R$ **then**

15       $\{v_i\} \leftarrow Sq.\text{getInsert}(v_2)$

16       $Sq.\text{insert}(v_i, v_2)$

The algorithm 5 is frequently called with v_3 being the direct successor of v_1: $v_1 \rightarrow v_3$. If so, only one iteration occurs at line 1, with $v_i = v_1$, and edge deletion occurs in constant time if v_2 is not excluded or inserted.

A specific situation must be checked if v_2 already belongs to the partial sequence. In case the nodes are consecutive in the partial sequence ($v_1, v_2, v_3 \in \vec{s} \wedge v_1 \prec v_2 \prec v_3$), a domain wipeout is triggered. No filtering occurs if $v_3 \preceq v_1$.

B.3. Require

Requiring a node v_i is obtained by marking it as required ($R \leftarrow R \cup \{v_i\}$) and inserting it if only one insertion remained for it ($nI_i = 1$), similarly to lines 15 and 16 of Algorithm 5. In case the node was excluded, a domain wipeout occurs.

B.4. Exclude

The exclusion of a node v_i is obtained by marking it as excluded ($X \leftarrow X \cup \{v_i\}$) and removing all edges connected to it. Although edge removal occurs, only the insertion counter nI_i changes. The counters related to other nodes $v_j \in I \wedge v_i \neq v_j$ are not affected by edge removal, since any edge (v_i, v_j) (or (v_j, v_i)) being removed could not define an insertion: no endpoint of the edge is within the partial sequence $\vec{s}$. In case the node was required, a domain wipeout occurs.

Appendix C: Filtering implementations

C.1.   TransitionTimes

The filtering occurs in two steps. First, the bounds of the time windows of visited nodes are updated by iterating over the partial sequence $\vec{s}$:

$$\lfloor \mathbf{Start}_j \rfloor \leftarrow \max \left(\lfloor \mathbf{Start}_j \rfloor, \lfloor \mathbf{Start}_i \rfloor + s_i + d_{i,j} \right) \qquad \forall v_i \rightarrow v_j \tag{43}$$

$$\lceil \mathbf{Start}_i \rceil \leftarrow \min \left(\lceil \mathbf{Start}_i \rceil, \lceil \mathbf{Start}_j \rceil - s_i - d_{i,j} \right) \qquad \forall v_i \rightarrow v_j \tag{44}$$

This step can be implemented in $O(|\vec{s}|)$. Next, all insertions of a node $v_j \in I$ between consecutive nodes $v_i, v_k \in \vec{s}, v_i \rightarrow v_k$ are inspected, similarly to line 8 of Algorithm 3. Each insertion can be used to define *ea* and *la*: the earliest and latest arrival at node v_j, if performed.

$$ea = \lfloor \mathbf{Start}_i \rfloor + s_i + d_{i,j} \tag{45}$$

$$la = \lceil \mathbf{Start}_k \rceil - s_j - d_{jk} \tag{46}$$

Insertion corresponding to time window violations are removed:

$$(ea > \lceil \mathbf{Start}_j \rceil) \vee (la < \lfloor \mathbf{Start}_j \rfloor) \vee (ea > la) \Longrightarrow \vec{s}.\text{notBetween}(v_i, v_j, v_k) \tag{47}$$

In (47), if either reaching node v_j cannot be done within its time window, or if doing a detour through it would violate the time window of v_k, the insertion is removed. This shares some similarities with the work from Savelsbergh (1985), where time windows are used for checking the validity of moves in local search. Finally, a time window update is performed for nodes $v_j \in R \setminus \vec{s}$ that are required but not yet inserted:

$$\lfloor \mathbf{Start}_j \rfloor \leftarrow \max \left(\lfloor \mathbf{Start}_j \rfloor, \min_{v_i \in \vec{s}.\text{getInsert}(v_j)} \lfloor \mathbf{Start}_i \rfloor + s_i + d_{i,j} \right) \tag{48}$$

$$\lceil \mathbf{Start}_j \rceil \leftarrow \min \left(\lceil \mathbf{Start}_j \rceil, \max_{v_i \in \vec{s}.\text{getInsert}(v_j), v_i \rightarrow v_k} \lceil \mathbf{Start}_k \rceil - s_j - d_{j,k} \right) \tag{49}$$

The visit time of such nodes are updated based on their earliest predecessor and latest successor in (48), (49) respectively. The time complexity of the filtering is the same as in the Distance constraint: $O(|E|)$. Although we reason over a set of required nodes as in Thomas et al. (2020), we do not ensure that a valid transition exists among all required nodes: this problem is NP-complete and would be too computationally expensive to perform at every filtering.

C.2. Precedence

The filtering consists of two main steps.

1. First, it considers the partial sequence $\vec{s}$ and ensures that nodes belonging to $\vec{o} \cap \vec{s}$ appears in the same order in both $\vec{o}$ and $\vec{s}$. This step can be implemented in $O(\max(|\vec{o}|, |\vec{s}|))$ and may trigger a failure.

2. Next, it considers the insertable nodes from the ordering $\vec{o}$ to respect, removing insertions that would violate the ordering if performed. This is described in Algorithm 6. A set Q tracks the nodes whose insertions will be filtered. The main loop iterates over each node $v_k \in \vec{o}$, as well as the last node ω. If v_k is insertable (*i.e.*, it is not yet in $\vec{s}$), it is added to Q for potential filtering (line 5). On the contrary, if v_k is already in $\vec{s}$, it serves as a boundary for the nodes in Q. For each node v_j in Q, we enforce that it can only be inserted between v_i and v_k, where:

   - v_i is the previous node in $\vec{o}$ that also belongs to $\vec{s}$, found in a previous iteration.
   - v_k (currently being iterated over) is the next node after v_j in $\vec{o}$ that belongs to $\vec{s}$.

   To enforce this consistency, we forbid all sequences where v_j appears before v_i or after v_k(line 7), preserving only insertions consistent with the required precedence.

Algorithm 6: Precedence$(\vec{S}, \vec{o})$ constraint filtering for invalid insertions.

---

1 $Q \leftarrow \emptyset$

2 $v_i \leftarrow \alpha$

3 **for** $v_k \in \vec{o} \cdot \omega$ **do**

4 $\quad$ **if** $\vec{S}$.isInsertable(v_k) **then**

5 $\quad\quad$ $Q \leftarrow Q \cup \{v_k\}$

6 $\quad$ **else if** $\vec{S}$.isMember(v_k) **then**

7 $\quad\quad$ **for** $v_j \in Q$ **do**

$\quad\quad\quad$ // v_j can only be inserted between v_i and v_k

8 $\quad\quad\quad$ $\vec{S}$.notBetween(α, v_j, v_i)

9 $\quad\quad\quad$ $\vec{S}$.notBetween(v_k, v_j, ω)

10 $\quad\quad$ $Q \leftarrow \emptyset$

11 $\quad\quad$ $v_i \leftarrow v_k$
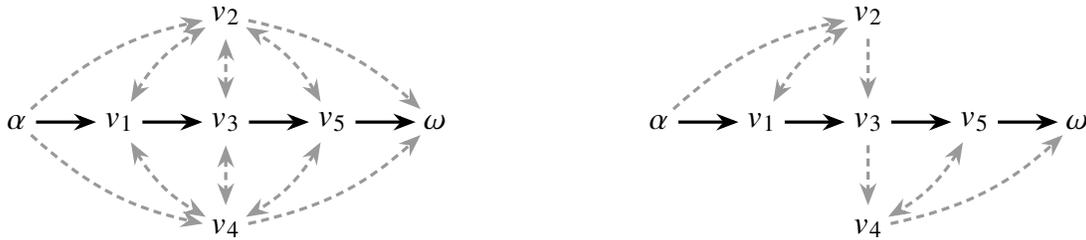

The time complexity of the filtering is dominated by Algorithm 6, running in $O(|\vec{o}| \cdot |V|)$.

EXAMPLE 4. An example of the filtering is illustrated in Figure 6. The current partial sequence is $\vec{s} = \alpha \cdot v_1 \cdot v_3 \cdot v_5 \cdot \omega$. The ordering to enforce is $\vec{o} = v_2 \cdot v_3 \cdot v_4$. The first step of the filtering, checking the ordering, does not trigger any failure. Algorithm 6 is then used for the second step. The four iterations done at line 3 are as follows:

1. $v_k = v_2$. Given that $v_2 \in I$, Q becomes $\{v_2\}$.

2. $v_k = v_3$, which belong to $\vec{s}$. The nodes in the queue Q must be placed between v_i and v_k (here forcing v_2 to be placed between α and v_3). This is done by two calls: $\vec{S}$.notBetween(α, v_2, α) (doing nothing) and $\vec{S}$.notBetween(v_3, v_2, ω). Finally, v_i becomes v_3 and Q is emptied.

3. $v_k = v_4$. Given that $v_4 \in I$, Q becomes $\{v_4\}$.

4. $v_k = \omega$, enforcing nodes in Q to be placed between v_3 and ω. The two calls at lines 8, 9 are $\vec{S}$.notBetween(α, v_4, v_3) and $\vec{S}$.notBetween(ω, v_4, ω) (the latter doing nothing).

Figure 6 **Precedence constraint with $\vec{o} = v_2 \cdot v_3 \cdot v_4$, before filtering (left) and after filtering (right).**

Note. Edges (v_2, v_4) and (v_4, v_2) are present but not drawn for clarity.

C.3. Cumulative

Firstly, to ensure that the start s_i and the end e_i of an activity $i \in A$ are visited together and in a valid order, two constraints are added per request. The constraint $\mathcal{R}_{s_i}(Sq) = \mathcal{R}_{e_i}(Sq)$ ensures that the two nodes s_i, e_i appear together, and Precedence$(Sq, (s_i \cdot e_i))$ ensures that the start s_i appears before the end e_i. The remaining filtering consists of three steps: computing a load profile, filtering the partially inserted activities and then the non-inserted activities.

A lower bound on the load profile is computed based on the activities that are fully or partially inserted, representing the sum of resources consumed at each node. It is described by three values for each inserted node $v \in \vec{s}$:

- l_v^- for the accumulated load before the visit of v ;

- l_v^+ for the accumulated load at the visit of v ;

- l_v^{v+1} for the accumulated load between the visit of v and its successor.

A load profile example is shown in Figure 7. Using three values per node to represent the load profile is needed to accurately represent whether activities may be put between a node and its successor, as shown in Figure 8.

For each fully inserted activity i, the load of i is added between the nodes: $\forall s_i \prec v \preceq e_i : l_v^- \leftarrow l_v^- + l_i$, $\forall s_i \preceq v \prec e_i : l_v^+ \leftarrow l_v^+ + l_i \wedge l_v^{v+1} \leftarrow l_v^{v+1} + l_i$. When considering only the inserted activities, it follows that $\forall v \in \vec{s} : l_v^+ = l_v^{v+1}$. Those equalities do not hold anymore after considering the partially inserted activities.

For partially inserted activities with a start inserted, a node from the partial sequence $\vec{s}$ is used instead of the non-inserted end node to compute the load. It corresponds to the earliest node after the start node or the start node itself,

after which the non-inserted end can be inserted. Partially inserted activities with the end inserted behave similarly, considering the latest node preceding the end, after which the start node can be inserted.

EXAMPLE 5. On Figure 7, activity 0 is partially inserted. The earliest predecessor for e_0 is s_0, which only contributes to the load $l_{s_0}^{s_0+1}$. For activity 1, the earliest predecessor of e_1 is e_2, contributing to $l_{s_1}^+, l_{s_1}^{s_1+1}, l_{e_2}^-, l_{e_2}^+$ (but not $l_{e_2}^{e_2+1}$).

Setting an entry in l^-, l^+ or l^{+1} exceeding the capacity triggers a failure.

Given the load profile, we filter insertions for the partially inserted activities and the non-inserted activities. Algorithm 7 depicts the filtering for every non-inserted activity $i \in A$ whose start s_i is inserted but not its corresponding end e_i. It first finds the closest node v after which the end e_i can be inserted (line 3), triggering a failure if no such node exists (line 6). Note that this closest node v was already used to set the load of activity i during the load profile computation, and therefore is already marked as valid (line 7). The next nodes to inspect are therefore the nodes following v, in order. As soon as the capacity occurring at a node v does not allow inserting the end e_i of the activity, all insertions between this invalid node v and the end ω of the sequence are removed (line 9). A similar filtering is performed in a mirror fashion, considering the non-inserted activities whose ends are inserted but not the corresponding start.

Finally, the filtering from Thomas et al. (2020) is used for non-inserted activities, inspecting every start (and end) of activities, checking if a matching end (and start) can be found, and removing insertions when no match exists.

Algorithm 7: Filtering of the Cumulative($\vec{S}, s, e, l, c$) for activities with start inserted.

Input: $\vec{S}$: sequence variable, s, e, l: start, end and load of activities, c: capacity.

1   **for** $i \in A$ **s.t.** $\left(\vec{S}.\text{isMember}(s_i) \text{ and not } \vec{S}.\text{isMember}(e_i)\right)$ **do**

2 $v \leftarrow s_i$

3   **while not** $\vec{S}.\text{canInsert}(v, e_i)$ **do**

4 $v \leftarrow \vec{S}.\text{getNext}(v)$

5   **if** $v = \omega$ **then**

6   **return failure**

7 $v \leftarrow \vec{S}.\text{getNext}(v)$

8   **while** $v \neq \omega$ **do**

9   **if** $\max(l_v^-, l_v^+) + l_i > c$ **then**

10 $\vec{S}.\text{notBetween}(v, e_i, \omega)$

11   **break**

12 $v \leftarrow \vec{S}.\text{getNext}(v)$
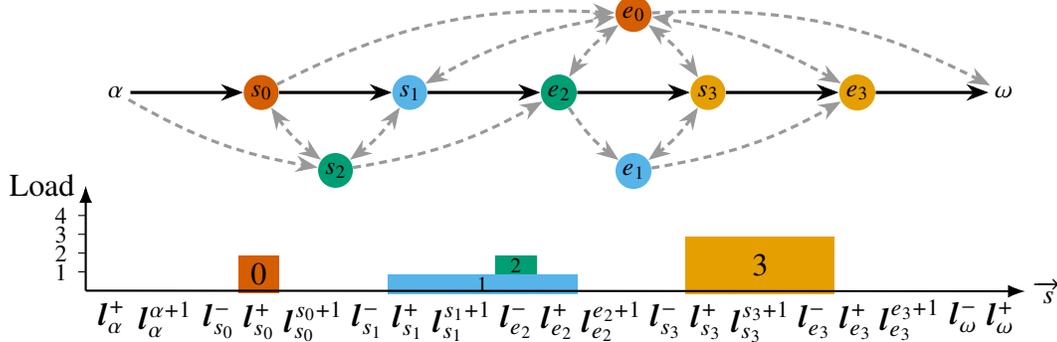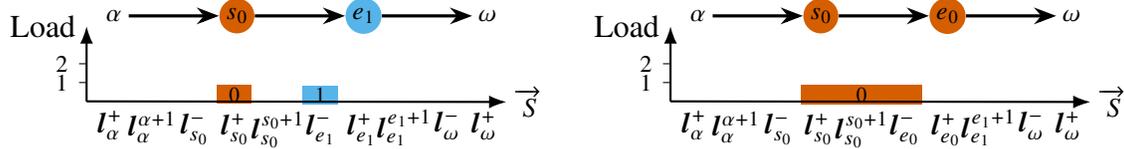

Appendix D: DARP

The DARP is defined over a graph $G = (V, E)$, the nodes being transportation points. A distance matrix d indicates the distance between nodes, and the objective is to minimize the total routing cost. The K available vehicles depart

Figure 7 A sequence variable $\vec{s}$ with partial sequence $\vec{s} = \alpha \cdot s_0 \cdot s_1 \cdot e_2 \cdot s_3 \cdot e_3 \cdot \omega$ **(top) and its corresponding load profile (bottom) when using a** $\mathrm{Cumulative}(\vec{s}, (s_0, s_1, s_2, s_3), (e_0, e_1, e_2, e_3), (2, 1, 1, 3), 4)$ **constraint. Edges between insertable nodes are not shown. For the partially inserted activity** 1, **the closest node after which** e_1 **can be inserted is** e_2 **hence rectangle 1 ends at** $l^+_{e_2}$ **instead of** $l^+_{s_1}$ **on the load profile.**

Figure 8 Two load profiles for different sequences, with a capacity of 2. With $l^{s_0+1}_{s_0}$, we detect that an activity of load 2 can be put between s_0 and e_1 on the left but cannot be put between s_0 and e_0 on the right.

from a common depot, fulfill a subset of the R transportation requests in the problem, and return to the common depot. Each request $r_i \in R$ is composed of a pickup $r^+_i \in V$ and its corresponding drop-off location $r^-_i \in V$. Several constraints restrict the types of travel that can be performed. Each node $v_i \in V$ to be visited (*i.e.* the depot, each pickup and drop location) has a service duration s_i, and a given time window: the visit must occur within $[a_i, b_i]$. Furthermore, the ride time a customer $r_i \in R$ spends in its vehicle is limited, ensuring the time from pickup r^+_i to drop off r^-_i does not exceed a predefined time limit $t_i > 0$. In addition to restricting the ride time, the route duration is also constrained: the total time between departing and returning to the depot cannot exceed a set limit $t^d > 0$. Finally, processing a transportation request $r_i \in R$ consumes a load of $q_i > 0$ in a vehicle, whose limited capacity c cannot be exceeded.

In both the sequence model and the Minizinc model, the set of nodes V and the distance matrix d are extended to introduce nodes representing the start and end vertices of the vehicles, which are duplicates of the depot. The values of C_1, C_2 in (24) were taken from Jain and Van Hentenryck (2011) and set to 80 and 1, respectively.

D.1. Minizinc Successor Model

The successor model is almost the same as in Berbeglia et al. (2011). Similarly to the model using sequence variables in (1)-(7), the set of nodes is extended to introduce the start and end nodes of the vehicles, and the time windows are modeled by an integer variable $\textbf{\textit{Time}}_v$ for each vertex $v \in V$. Their domains are also initialized based on the time windows from the instance. Each node v has an associated successor variable $\textbf{\textit{Succ}}_v$, whose domain is all the nodes of the problem, except for the end depots. For each vehicle $k \in K$, the successor of its end depot ω_k is set to the start depot α_{k+1} of the following vehicle $k + 1$. The load at each node is tracked in an integer variable $\textbf{\textit{Load}}_v$, with domain $0 \ldots c$, and fixed to 0 in case of depot nodes. Moreover, an array $\textbf{\textit{l}}$ for the load change of each node is created as:

$$l_v = \begin{cases} \boldsymbol{q_i} & \text{if } v = \boldsymbol{r_i^+} \\ -\boldsymbol{q_i} & \text{if } v = \boldsymbol{r_i^-} \\ 0 & \text{otherwise} \end{cases} \tag{50}$$

Which describes a positive load change for a pickup, negative for a delivery, and null for a depot node. Finally, an integer variable $\boldsymbol{Vehicle_v}$ represents the vehicle that visits a node $v \in V$. Its domain is initialized in $\{0 \ldots |K| - 1\}$, except for the start and end depots, where it is set to the id of the corresponding vehicle. The model is as follows:

$$\min \sum_{v \in V} \boldsymbol{Dist_v} \tag{51}$$

subject to:

$$\text{circuit}(\boldsymbol{Succ}) \tag{52}$$

$$\boldsymbol{Dist_v} = d_{v,\boldsymbol{Succ_v}} \qquad\qquad \forall v \in V \tag{53}$$

$$\boldsymbol{Time_v} + s_v + \boldsymbol{Dist_v} \le \boldsymbol{Time_{Succ_v}} \qquad\qquad \forall v \in V^{\backslash \omega} \tag{54}$$

$$\boldsymbol{Load_v} + l_v = \boldsymbol{Load_{Succ_v}} \qquad\qquad \forall v \in V \tag{55}$$

$$\boldsymbol{Time_{r_i^-}} - \boldsymbol{Time_{r_i^+}} - s_{r_i^+} \le t_i \qquad\qquad \forall r_i \in R \tag{56}$$

$$\boldsymbol{Time_{\omega_k}} - \boldsymbol{Time_{\alpha_k}} \le t^d \qquad\qquad \forall k \in K \tag{57}$$

$$\boldsymbol{Time_{r_i^+}} + s_{r_i^+} + d_{r_i^+,r_i^-} \le \boldsymbol{Time_{r_i^-}} \qquad\qquad \forall r_i \in R \tag{58}$$

$$\boldsymbol{Vehicle_v} = \boldsymbol{Vehicle_{Succ_v}} \qquad\qquad \forall v \in V^{\backslash \omega} \tag{59}$$

$$\boldsymbol{Vehicle_{r_i^+}} = \boldsymbol{Vehicle_{r_i^-}} \qquad\qquad \forall r_i \in R \tag{60}$$

Where $V^{\backslash \omega}$ denotes all nodes in V, except for nodes corresponding to end depots. The objective is to minimize the total distance (51), which is summed over the distance between each node and its successor (53). Due to the duplications of the start and end depots, the circuit constraint can be used (53). The time between each node and its successor is constrained in (54), except for the successors of the end depots. The load at each node is tracked with (55). The constraints (56) and (57) enforce the max ride and maximum route duration, respectively, and are the exact same constraints as (6) and (7). The constraint (58) ensures that the visit time of a drop is at least the visit time of its pickup and the travel between the two nodes. They are not necessary for correctness, but help in filtering the time windows. Lastly, the vehicle passing at a node also passes at its successor (59) and must be the same between each pickup and delivery pair (60).

Search annotations were provided, as they greatly improve the quality of solutions. They tell to branch first on the successor variables $\boldsymbol{Succ}$ using the DomWDeg heuristics Boussemart et al. (2004), which prioritizes variables associated with small domains and a large number of failures encountered. The LNS consists in keeping 85% of variables to their value in a previous solution, and is restarted after a constant number of 10.000 search nodes.

D.2. Results per instance

Table 7 shows the performance of the different solvers for each instance.

Table 7 **Primal gap to the best known solution (bks) in percentage on the DARP instances. Best results are in bold.**

Instance	k	|R|	bks	Succ		Succ-LNS		CPO		OR-Tools		Hexaly		Seqvar	
				Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min
R1a	3	24	190.02	22.34	22.34	5.11	3.19	6.50	4.05	-	-	3.63	0.97	**0.00**	**0.00**
R1b	3	24	164.46	58.38	58.38	6.47	2.51	14.64	11.64	1.78	1.78	5.08	2.67	**0.40**	**0.00**
R7a	4	36	291.71	-	-	12.68	7.29	14.90	4.09	-	-	5.37	1.62	**1.96**	**1.08**
R7b	4	36	248.21	-	-	11.60	2.83	14.46	9.09	-	-	7.75	5.27	**3.14**	**0.74**
R2a	5	48	301.34	-	-	10.30	3.83	-	-	-	-	8.12	3.79	**1.71**	**0.29**
R2b	5	48	295.66	74.41	74.41	10.41	5.25	19.26	12.66	**2.52**	2.52	9.27	5.56	4.89	**2.33**
R8a	6	72	487.84	-	-	-	-	-	-	-	-	15.09	10.12	**7.28**	**3.07**
R8b	6	72	458.73	-	-	-	-	-	-	10.15	10.15	16.31	11.98	**6.04**	**4.45**
R3a	7	72	532.00	-	-	-	-	-	-	-	-	12.14	7.80	**4.62**	**1.80**
R3b	7	72	484.83	-	-	-	-	-	-	8.67	8.58	13.97	9.41	**7.01**	**2.49**
R9a	8	108	653.94	-	-	-	-	-	-	-	-	31.11	24.29	**9.94**	**6.79**
R9b	8	108	592.23	-	-	-	-	-	-	-	-	18.34	14.68	**9.93**	**7.82**
R4a	9	96	570.25	-	-	-	-	-	-	9.62	9.62	19.10	15.21	**8.40**	**3.54**
R4b	9	96	529.33	-	-	-	-	45.92	28.99	12.78	12.78	17.75	13.71	**9.94**	**7.92**
R10a	10	144	845.47	-	-	-	-	-	-	-	-	27.61	21.05	**11.27**	**9.17**
R10b	10	144	783.81	-	-	-	-	-	-	-	-	27.95	23.14	**12.70**	**9.78**
R5a	11	120	625.64	-	-	-	-	-	-	15.87	15.87	16.73	13.76	**10.82**	**8.00**
R5b	11	120	573.56	-	-	-	-	-	-	13.72	13.72	17.32	14.25	**9.45**	**5.36**
R6a	13	144	783.78	-	-	-	-	-	-	15.33	15.33	17.96	13.76	**11.04**	**7.83**
R6b	13	144	725.22	-	-	-	-	-	-	15.23	15.19	18.60	14.61	**9.95**	**8.40**