Sequence Variables for Routing Problems

Augustin Delecluse
TRAIL, ICTEAM, UCLouvain, Belgium

Pierre Schaus
ICTEAM, UCLouvain, Belgium

Pascal Van Hentenryck
Georgia Institute of Technology, USA

Abstract

Constraint Programming (CP) is one of the most flexible approaches for modeling and solving vehicle routing problems (VRP). This paper proposes the *sequence variable* domain, that is inspired by the insertion graph introduced in [4] and the subset bound domain for set variables. This domain representation, which targets VRP applications, allows for an efficient insertion-based search on a partial tour and the implementation of simple, yet efficient filtering algorithms for constraints that enforce time-windows on the visits and capacities on the vehicles. Experiment results demonstrate the efficiency and flexibility of this CP domain for solving some hard VRP problems, including the Dial-A-Ride, the Patient Transportation, and the asymmetric TSP with time windows.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Programming, Dial-A-Ride, Patient Transportation, TSPTW, Vehicle Routing, Sequence Variables, Insertion Variables

Digital Object Identifier 10.4230/LIPIcs.CP.2022.38

Funding *Augustin Delecluse*: This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.A

1 Introduction

Vehicle routing problems (VRP) [27] are of great importance for the distribution of goods in the supply chain. In order to cope with increasing urbanization and ecological challenges, it is also expected that flexible transport offers for people, such as on-demand transport, will have to be further developed in the future [12]. This raises new challenges for optimization, in particular the development of generic and reusable tools in many contexts and variants of VRP.

Constraint Programming (CP) is one of the most flexible approaches for modeling vehicle routing problems (VRP). One standard model consists in using the so-called successor model with one variable for each visit that represents the next visit in the tour of a vehicle [3]. Due to its simplicity, this model has two practical limitations involving both the modeling and research components of CP. At the modeling level, it is not straightforward to represent the optional aspect of some visits in the successor model. This requires the introduction of a fake vehicle visiting all excluded visits. At the search level, sub-chains formed by fixed successors do not allow any insertion in the middle of a partial tour during the search. Sub-chains that are formed close to the root during the search are fixed with little information and hardly reconsidered in large search spaces explored with a backtracking search. The goal of the sequence variable is to address those two limitations.

1. It can easily model the exclusion of visits not inserted in a tour similarly to a set variable.
2. Inspired by the idea of the *insertion graph* [4], it allows the insertion of a visit in the middle of the partial tour enabling the implementation of a depth first tree search insertion

exploration algorithm similar to the ones used in [4, 15] to reinsert optimally a set of relaxed visits in a large neighborhood search (LNS).

We introduce the *Sequence Variable* domain as well as the mechanism to make the domain reversible in a trail-based solver. Two important constraints and their associated filtering algorithms are described: The *TransitionTimes* constraint links the time window constraints of the visits to a distance matrix, removing insertions that would process a request outside of its time window; The *Cumulative* constraint ensures that the load change performed in each visit never exceeds the maximum capacity of a vehicle. We model three constrained VRP problems with the *Sequence Variable* that are illustrative of the functionality offered for both the modeling and the search flexibility: The Dial-A-Ride Problem (DARP) [8, 15], the Patient Transportation Problem (PTP) [5, 19] and the Traveling Salesman Problem with time windows (TSPTW). Experimental results demonstrate the effectiveness of the approach. It obtains better results than baseline models with sequences of conditional task interval variables in CP Optimizer [18] and obtains results competitive with the state-of-the art results published in the literature on the DARP and PTP. For TSPTW we improve the best know solutions for 32 out of the 205 instances tested in the standard benchmark suite [20].

The rest of the paper is organised as follows. Section 2 details related work on VRP and existing Sequence Variables. Section 3 dives into the definition of a Sequence Variable, its interface and implementation. Section 4 shows how VRP constraints are implemented using the variable. Lastly, our models and experimental results for DARP, PTP and TSPTW are presented in Section 5.

2 Related Work

In [26], the authors introduced a Sequence Variable for scheduling and routing problems. This domain representation directly extends the subset bound domain representation for set variables [14] by partitioning the visits into a required, possible and excluded set plus a partial sequence and a set of insertion points. In this work we simplify this idea by getting rid of the required set. As a consequence, a possible visit must be directly scheduled in the partial sequence but cannot be required without being inserted in the sequence. This modification, despite its simplicity, greatly eases the reasoning made by the constraints, their time complexity and the implementation of search heuristics, while losing little to no flexibility in practice. The proposed sequence domain can be seen as the making of the *insertion graph* idea introduced in [4] more generic and encapsulated as the internal implementation of the sequence variable domain.

Although not published, IBM ILOG CP Optimizer [18] also proposes sequence variables to decide the order of visits. The functionalities and constraints of this sequence variable are briefly described [16, 17], no details are given about the exact implementation. According to their documentation [6, 7], they use a Head-Tail structure, maintaining a separate growing head and tail to add new Interval variables to the beginning or end of the sequence, respectively. The head and tail merge to form the final sequence once no Interval can be added to either of them. This implementation appears to be similar to Google OR-Tools [23] and its own Sequence Variables [24]. Although more targeted to scheduling problems, this sequence was used for solving the PTP in [5] and [19], and is suitable for VRP.

3 Sequence Variable

We first introduce useful notations related to sequences and operations on these. We then formally define the domain of a sequence variable before considering the practical algorithmic details for implementing this domain in a constraint programming solver.

3.1 Sequence notations

The notations are largely borrowed from [26] but reintroduced in this paper for reading convenience. The set of locations that can be visited by a vehicle are referred to as *nodes* and the set of all nodes is denoted as $\mathcal{X}$. A sequence over $\mathcal{X}$ is denoted by $\overrightarrow{S}$ and the set of all sequences of $\mathcal{X}$ by $\overrightarrow{\mathcal{P}}(\mathcal{X})$. The sequence $\overrightarrow{S}$ defines an order over the elements of S. The set of elements in the sequence $\overrightarrow{S}$ is denoted S. All the entries of the sequence are different and therefore $|\overrightarrow{S}| = |S|$. The notation $p \overset{\overrightarrow{S}}{\prec} q$ means that the node p precedes node q in $\overrightarrow{S}$ and $p \xrightarrow{\overrightarrow{S}} q$ means that p directly precedes q in $\overrightarrow{S}$. Those notations are simply written $p \prec q$ and $p \rightarrow q$ when clear from the context. If the nodes can be equal, the relation is written $p \preceq q$. A sequence can be grown by using an insertion operation $insert(\overrightarrow{S}, p, q)$ with $p \in S, q \notin S$ that results in inserting q just right after p in the sequence. More exactly assuming $\overrightarrow{S} = \overrightarrow{S}_1 \cdot p \cdot \overrightarrow{S}_2$ the resulting super-sequence is $\overrightarrow{S}' = \overrightarrow{S}_1 \cdot p \cdot q \cdot \overrightarrow{S}_2$. The operation is also noted $\overrightarrow{S} \underset{(p,q)}{\Longrightarrow} \overrightarrow{S}'$. Given I, a set of pairs of type (p, q), each corresponding to a potential insertion in $\overrightarrow{S}$, $\overrightarrow{S} \underset{I}{\Longrightarrow} \overrightarrow{S}'$ means that $\overrightarrow{S}'$ could be produced by applying one insertion from I on $\overrightarrow{S}$: $\exists (p, q) \in I \mid \overrightarrow{S} \underset{(p,q)}{\Longrightarrow} \overrightarrow{S}'$. More generally the *zero or more derivation steps* is defined as $\overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' \equiv \overrightarrow{S} = \overrightarrow{S}' \vee \left(\exists (p, q) \in I \mid \overrightarrow{S} \underset{(p,q)}{\Longrightarrow} \overrightarrow{S}'' \wedge \overrightarrow{S}'' \overset{*}{\underset{I \setminus \{(p,q)\}}{\Longrightarrow}} \overrightarrow{S}' \right)$. Note that I may contain tuples that do not correspond to a possible insertion in $\overrightarrow{S}$ but instead to a possible insertion in a super-sequence of $\overrightarrow{S}$. Also note that several sequences of insertions in I may lead to an identical super-sequence.

3.2 The sequence domain

The formal definition of a sequence domain is given next.

▶ **Definition 1.** *The domain of a Sequence Variable Sq is represented as* $\langle \overrightarrow{S}, I, P, E \rangle$*, with* $\overrightarrow{S}$ *a sequence of nodes forming a partial tour, insertion points* $I \subseteq \mathcal{X} \times \mathcal{X}$ *and two subsets of nodes* $P, E \subseteq \mathcal{X}$ *for nodes that can possibly be inserted and nodes that are excluded from the sequence, respectively. The domain of Sq, also noted* $D(Sq)$*, is defined as* $\langle \overrightarrow{S}, I, P, E \rangle \equiv \left\{ \overrightarrow{S}' \in \overrightarrow{\mathcal{P}}(P \cup S) \mid \overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' \right\}$ *and capture all the possible valid derivations from the partial tour* $\overrightarrow{S}$ *using insertions of* I*.*

At its initialization the Sequence Variable is composed of a partial tour of two nodes $\alpha \cdot \omega$ for the beginning α and ending ω of the tour and no insertions are allowed after ω to ensure ω remains the last visited node. P is thus equal to $\mathcal{X} \setminus \{\alpha, \omega\}$, $E = \phi$ and the set of insertions is $I = \{(p, q) \in P \times \mathcal{X} \mid p \neq \omega\}$. Imposing a first and last node in the sequence conveniently allows the modeling of problems where the route taken by a vehicle needs to end at its starting point (α lies at the same location as ω) or at another location ($\alpha \neq \omega$) and prevent the API to deal with the special case of empty sequences that require the introduction of a dummy symbol as in [26]. This use of beginning and ending nodes is also used in CP Optimizer and is described as *sinks* in their API [18].

Different forms of consistency could be imagined ensuring for instance that, for all pairs of nodes $(p, q) \in I$, both p and q are reachable from α by using the arcs defined in I and $\overrightarrow{S}$. Checking this consistency would relax the constraint that sequences only visit each node at most once.

In practice we use an even weaker form of consistency on the sequence domain, that is cheap to compute and is captured by the following invariant:

$$S \cup P \cup E = \mathcal{X} \wedge S \cap P = S \cap E = P \cap E = \phi \tag{1}$$

$$\forall (p, q) \in I : p \notin E \wedge q \in P \tag{2}$$

$$\forall q \in P : \exists p \in S \cup P \mid (p, q) \in I \tag{3}$$

(1) Nodes in the partial sequence S, in the possible set P and the set of excluded E form a partition of $\mathcal{X}$; (2) valid insertions are constituted of possible nodes after non excluded nodes (thus not necessarily present in the partial sequence); any excluded node cannot be inserted in $\overrightarrow{S}$ and is not a valid predecessor; any possible element can be inserted after a node (3). This weak consistency can for instance not detect situations where all the edges in I are disconnected from the partial sequence S, forming a disconnected cluster of nodes whose members should be excluded.

3.3 Implementation and data-structures

The implementation of the domain $\langle \overrightarrow{S}, I, P, E \rangle$ should be reversible for trail-based solver such as MiniCP [21] and most of the update and domain iteration operations should be as efficient as possible.

The set partitioning between the sets S, P, E is implemented using a single reversible sparse-sets data-structure as described in [10] ensuring removal and state restoration in constant time.

The insertion points set I is partitioned with one set $I^x = \{p \in (S \cup P) : (p, x) \in I\}$ for each node $x \in \mathcal{X}$ composed of the valid predecessors for node x. Those sets are implemented using reversible sparse-sets ensuring removal and state restoration in constant time. The lower-level consistency invariant expressed in terms of these data structures are given next in equations (4) to (7).

$$S \cup P \cup E = X \wedge S \cap P = S \cap E = P \cap E = \phi \tag{4}$$

$$p \in E \implies I^p = \phi \wedge \forall x : p \notin I^x \tag{5}$$

$$p \in S \implies I^p = \phi \tag{6}$$

$$I^p = \phi \implies p \in S \vee p \in E \tag{7}$$

Through the use of the reversible sparse-set, (1) is directly equivalent to (4). (2) is respected through (5) ($\forall (p, q) \in I : p \notin E$) and through (4), (5), (6) and (7) ($q \in E \cup S \Leftrightarrow q \notin P \implies I^q = \phi \implies \forall (p_1, q_1) \in I : q_1 \neq q$). Finally (3) is retrieved by combining (4) to (7) ($I^x \neq \phi \implies x \notin (S \cup E) \iff x \in P$).

The internal partial sequence $\overrightarrow{S}$ of nodes is implemented using an array of reversible integers, as in [26]. This array stores the current successor of a node, and an element without successor points towards itself.

Additionally, the implementation maintains two reversible integers for every node $x \in \mathcal{X}$: n_s^x for the size of $I^x \cap S$ and n_p^x for the size of $I^x \cap P$. Those values are useful during the search to implement heuristics, for instance when searching the node $i \in P \mid n_s^i \leq n_s^j \; \forall j \in P$ having the least predecessors in the current ordering $\overrightarrow{S}$.

A domain representation example for a Sequence Variable is depicted in Figure 1.

Table 1 highlights the most important operations available on a Sequence Variable and their associated time complexity.

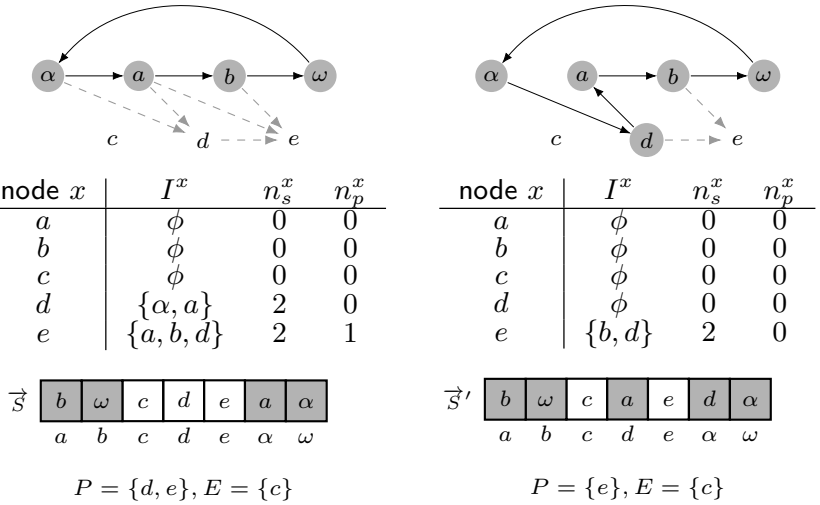

Figure 1 Representation of the implementation of a Sequence Variable. The left part shows a sequence ordering as well as possible insertions (dashed lines) for nodes $x \in P$. Below is a table showing the insertions for the nodes, the successor of the sequence (only relevant for nodes $\notin P \cup E$) and the split of nodes between S, P and E. The right part shows a modification $\overrightarrow{S} \underset{(\alpha,d)}{\Longrightarrow} \overrightarrow{S}'$ where node d has been inserted after node α, changing its status and removing insertion a from I^e due to some constraint.

Operation	Description	Complexity
`isBound(Sq)`	return true iff $\|P\| = 0$	$\Theta(1)$
`is{Member/Possible/Excluded}(Sq, x)`	return true iff $x \in \{S/P/E\}$	$\Theta(1)$
`get{Member/Possible/Excluded}(Sq)`	enumerate over $\{\overrightarrow{S}/P/E\}$	$\Theta(\|\{\overrightarrow{S}/P/E\}\|)$
`succ(Sq, x)`	return $q \mid x \to q$	$\Theta(1)$
`pred(Sq, x)`	return $p \mid p \to x$	$\Theta(1)$
`insert(Sq, p, x)`	insert x into Sq such that $p \to x$ holds	$\Theta(P)$
`exclude(Sq, x)`	exclude x from Sq	$\Theta(P)$
`nMemberInserts(Sq, x)`	return $n_s^x = \|I^x \cap S\|$	$\Theta(1)$
`nPossibleInserts(Sq, x)`	return $n_p^x = \|I^x \cap P\|$	$\Theta(1)$
`getMemberInserts(Sq, x)`	enumerate over $I^x \cap S$	$\Theta(\min(\|I^x\|, \|S\|))$
`getPossibleInserts(Sq, x)`	enumerate over $I^x \cap P$	$\Theta(\min(\|I^x\|, \|P\|))$
`canInsert(Sq, p, x)`	return true iff $p \in I^x$	$\Theta(1)$
`removeInsert(Sq, p, x)`	remove p from I^x	$\mathcal{O}(P)$

Table 1 Operations on a Sequence Variable `Sq` with domain $\langle \overrightarrow{S}, I, P, E \rangle$.

Any global constraint interested to be notified on domain modification of the Sequence Variable can do it using the three following hookup events:

- The sequence is bound, that is the set of possible nodes is empty;
- A node has been inserted / excluded and this node is provided as a parameter of the event to allow incremental updates;
- The number of elements within a set I^x has changed, and the corresponding node is provided as a parameter of the event to allow incremental updates.

In the next section, we describe the filtering algorithms of some important constraints on the sequence variables.

4 Global constraints

This section defines and explains the filtering algorithm for some constraints on the Sequence variables. Some were already introduced in [26] but the filtering algorithms are adapted to reflect the removal of the required set. Some constraints reason over a list of values, written $[x]$ when x is a list, or a matrix of values, denoted by $[[x]]$.

Dependency

Despite the removal of the required set, one might still want to require a particular node to be inserted in one specific sequence. The `Dependency` constraint allows this. This constraint takes a list Dep of nodes as parameter that must all be member of the sequence or excluded from it. It is formally defined as

$$\texttt{Dependency}(Sq, [Dep]) \equiv \left\{ \overrightarrow{S} \in D(Sq) \mid Dep \cap S \neq \phi \Leftrightarrow Dep \cap S = Dep \right\}. \tag{8}$$

Filtering The filtering is triggered when a node $d \in Dep$ is either excluded or inserted into the sequence. If it is excluded, it excludes all other nodes $d' \in Dep \mid d \neq d'$. If it is inserted, it ensures that excluding any other node $d' \in Dep \mid d \neq d'$ results in a failure. The complexity of this filtering is $\mathcal{O}(|Dep|)$.

Disjoint

This constraint ensures that every node $x \in \mathcal{X}$ can be inserted once and only once across all sequences $Sq \in SQ$:

$$\begin{aligned}&\texttt{Disjoint}([SQ], \mathcal{X}) \equiv \\ &\forall Sq, Sq' \in [SQ], \forall x \in \mathcal{X}, Sq \neq Sq' \implies x \in Sq \implies x \notin Sq'\end{aligned} \tag{9}$$

Filtering The filtering is triggered when a node $i \in \mathcal{X}$ is inserted in a Sequence $Sq \in SQ$. It excludes i from all other Sequences $Sq' \in SQ \mid Sq \neq Sq$. As the constraint can be notified of the value of i when an insertion occurs through the hookups events, we only need to iterate over SQ for checking the consistency. The complexity of the filtering when a single node is inserted is therefore $\mathcal{O}(|SQ|)$.

This constraint can optionally enforce that nodes must be inserted in exactly one of the sequences: $\forall x \in \mathcal{X}\ \exists Sq \in SQ \mid x \in D(Sq)$. If this is the case, the constraint fails if a mandatory node is excluded from all sequences .

Precedence

For some applications, visiting a set of nodes in a specific order is important and those nodes must all be inserted or not at all. This constraint is similar to the `Dependence` constraint but is done over an ordered set $\overrightarrow{D}$ of nodes, ensuring that the order in the set appears within the sequence if some node $n \in \overrightarrow{D}$ belongs to the Sequence. It is formally defined as

$$\texttt{Precedence}(Sq, \overrightarrow{D}) \equiv \left\{ \overrightarrow{S} \in D(Sq) \mid \overrightarrow{S} \cap \overrightarrow{D} \neq \phi \implies \forall i, j \in \overrightarrow{D} : i \overset{\overrightarrow{D}}{\prec} j \implies i \overset{\overrightarrow{S}}{\prec} j \right\} \tag{10}$$

Filtering The filtering is triggered whenever a node is inserted into the sequence or excluded from it. It iterates over the nodes in $\overrightarrow{D}$ and ensures that they appear in the same order in $\overrightarrow{S}$ if they belong to $D(Sq)$. It then removes the insertions that would prevent the order from being respected:

$$\forall i,j \in \overrightarrow{D}\ \forall p \in \overrightarrow{S} \mid \left(i \overset{\overrightarrow{D}}{\prec} j \wedge p \overset{\overrightarrow{S}}{\prec} i \implies p \notin I^j\right) \wedge \left(i \overset{\overrightarrow{D}}{\prec} j \wedge j \overset{\overrightarrow{S}}{\prec} p \implies p \notin I^i\right) \tag{11}$$

Furthermore, if some node in $\overrightarrow{D}$ is excluded from the Sequence, all nodes from $\overrightarrow{D}$ are excluded as well. The time complexity is $\mathcal{O}(|D| \cdot |S|)$ as the constraint considers the insertions $I^x \cap S$ for every node $x \in \overrightarrow{D}$.

Transition Times

The `TransitionTimes` constraint is used for problems where nodes are related to a time window and where transitions from one node to another take a certain duration specified in a distance matrix. This constraint ensures that the order defined by the successor set $\overrightarrow{S}$ is feasible: all nodes in S must be visited within their time window.

More formally, each node $x \in \mathcal{X}$ is attached to a time window variable $[start_x]$ and a duration $duration_x$. A matrix $trans \in R^{n \times n}$ with n nodes defines the transition times between elements and satisfies the triangular inequality. The sum of transition times when following the path described by the sequence is defined by a variable $transitionTime$. The constraint is defined as

$$\texttt{TransitionTimes}(Sq, [start], [duration], [[trans]], transitionTime) \equiv$$
$$\left\{ \overrightarrow{S} \in D(Sq) \;\middle|\; \begin{array}{c} \forall i,j \in \overrightarrow{S}, i \overset{\overrightarrow{S}}{\prec} j \implies start_i + duration_i + trans_{i,j} \leq start_j \\ transitionTime = \sum_{i,j \in \overrightarrow{S} \mid i \to j} trans_{i,j} \end{array} \right\} \tag{12}$$

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 (12) uses inequalities.

Filtering The pseudo code for the filtering is shown in Algorithm 1. It first ensures that a Sequence respects its time windows: an iteration over $\overrightarrow{S}$ is done, updating the bounds for $start_i\ \forall i \in \overrightarrow{S}$ (line 2). No time window update is done for nodes $\notin \overrightarrow{S}$. Afterwards, it computes the current length of the sequence as the sum of transitions between elements in S and uses it to update the bounds of $transitionTime$ (line 4). Only the lower bound is updated, as we could exclude all remaining nodes in P and still get a valid solution. Then, the algorithm starts removing invalid insertions. An insertion $p \in I^x \cap S$ for a node $x \in P$ is invalid if reaching x through p would violate its time window (line 8), prevent the current successor $q \mid p \to q$ of p to be reached within its own time window (line 13) or exceed the maximum traveled distance (line 17). Line 12 uses a max because reaching a node before its time window is possible: if $reaching_x < \min(start_x)$, the departure occurs at $\min(start_x)$, otherwise it happens at $reaching_x$.

The time complexity of this filtering is $\mathcal{O}(|P| \cdot |S|)$. However, the effective complexity is slightly lower as $I^x \cap S$ is retrieved in $\Theta(\min(|S|, |I^x|))$. A similar pruning can also be defined for predecessors $p \in I^x \cap P$ of $x \in P$, ensuring that doing the transition from p to x would not exceed $start_x$. Because we do not reason over a set of Required nodes as in [26], we do not need to ensure that a valid transition exists among those Required nodes, removing the NP-complete problem of checking such transition.

Algorithm 1 `TransitionTimes`$(Sq = \langle \overrightarrow{S}, I, P, E\rangle, [start], [duration], [[trans]], transitionTime)$ filtering.

```
1  for i ∈ S⃗ do
2  |  update time windows start_i
3  length ← current distance of the sequence
4  min (transitionTime) ← length
5  for x ∈ P do
6  |  for p ∈ I^x ∩ S⃗ do
7  |  |  reaching_x ← min(start_p) + duration_p + trans_{p,x}
8  |  |  if reaching_x > max(start_x) then
9  |  |  |  remove p from I^x
10 |  |  else
11 |  |  |  q ← succ(Sq, p)
12 |  |  |  reaching_q ← max (reaching_x, min(start_x)) + duration_x + trans_{x,q}
13 |  |  |  if reaching_q > max(start_q) then
14 |  |  |  |  remove p from I^x
15 |  |  |  else
16 |  |  |  |  detour ← trans_{p,x} + trans_{x,q} − trans_{p,q}
17 |  |  |  |  if detour + length > max(transitionTime) then
18 |  |  |  |  |  remove p from I^x
```

Cumulative

Common variations of VRP include pickup and delivery occurring at nodes, consuming a certain amount of load available in a vehicle. By analogy to scheduling problems, this constraint is called the `Cumulative` constraint: when providing a set of activity consuming a certain load $load_x$, it ensures that a maximum capacity is never exceeded and filters insertions that would exceed the available capacity. As our filtering is close to the one presented in [26] but more enhanced, we will borrow their notation.

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

$$\texttt{Cumulative}(Sq, [start], [end], [load], C) \equiv$$
$$\left\{ \overrightarrow{S} \in D(Sq) \mid \forall e \in \overrightarrow{S}, \sum_{i \in A | start_i \preceq e \preceq end_i} load_i \leq C \right\} \tag{13}$$

Checking The checking consists of verifying that an optimistic load profile does not exceed the vehicle capacity. We introduce two sets of values that represent the accumulated capacity at each node visited in the order of the partial sequence instead of one as in [26]. This allows computing a more realistic load profile and filtering more insertion points. Those two sets are denoted $C^b = \{C^b_x \; \forall x \in \overrightarrow{S}\}$ for the accumulated capacity just *before* visiting a given node and $C^a = \{C^a_x \; \forall x \in \overrightarrow{S}\}$ for the accumulated capacity just *after* leaving a given node.

The computing of those values is presented in Algorithm 2. It looks at the positions of the start and end of fully inserted activities, and increases C^a from the start until the node before the end node (line 7). For C^b, it is increased from the node after the start until the end node, included (line 5). When encountering a partially inserted activity i, our optimistic load profile considers that a sequence can be formed where $start_i \rightarrow end_i$ and thus only increases the value of C^a (line 10) or C^b (line 12) at one node. This setting for the load profile implies $C^b_s < C^a_s$ for every inserted start s and $C^b_e > C^a_e$ for every inserted end e. An example load profile is shown in Figure 2. Note that we do not necessarily have $C^a_i = C^b_j \mid i \rightarrow j$, as illustrated in Figure 3.

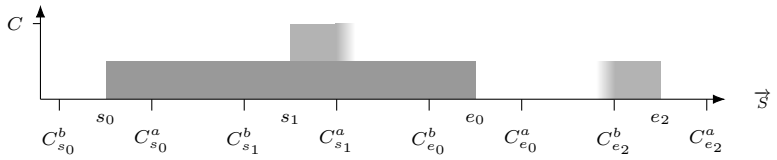

Figure 2 Load profile for the `Cumulative` constraint with $C = 2$ and $\vec{S} = \{\alpha, s_0, s_1, e_0, e_2, \omega\}$. Each activity has a load of 1, activity 0 (s_0, e_0) is fully inserted (dark gray) and both activity 1 and 2 are partially inserted (light gray). e_1 is considered to be inserted right after s_1, whose load only affects $C^a_{s_1}$. For activity 2, s_2 is consider to be inserted before e_2, affecting the value $C^b_{e_2}$.

Algorithm 2 `LoadProfile`$(Sq, start, end, load, C)$ computation.

Input : $start, end, load$: start, end and load of activities, C: capacity, $Sq = \langle \vec{S}, I, P, E \rangle$: Sequence Variable.
Output : C^b, C^a: capacity before arriving at a node and after leaving a node, respectively.

```
1  C^b, C^a ← 0
2  for i | start_i ∈ S ∨ end_i ∈ S do
3      if start_i ∈ S ∧ end_i ∈ S then
4          for x ∈ S⃗ | start_i ≺ x ⪯ end_i do
5              C^b_x ← C^b_x + load_i
6          for x ∈ S⃗ | start_i ⪯ x ≺ end_i do
7              C^a_x ← C^a_x + load_i
8      else
9          if start_i ∈ S then
10             C^a_{start_i} ← C^a_{start_i} + load_i
11         else
12             C^b_{end_i} ← C^b_{end_i} + load_i
13 return C^b, C^a
```

Filtering The filtering is triggered whenever new elements are inserted into the sequence. It uses the load profile computed during the checking to filter two cases: the partially inserted activities first and the non-inserted activities afterwards.

The partially inserted activities are considered first: we remove the insertions points for their non-inserted node that would cause the maximum capacity to be exceeded. A filtering example for removing insertions for starts whose corresponding end is inserted is shown in Algorithm 3. We iterate over the sequence in backward order (line 9) and compute the capacity occurring at the node (line 6 and 10). As soon as the maximum capacity would

be exceeded if the start was inserted there, we remove the corresponding insertions (line 7 and 11). We inspect both the capacity before arriving at a node (line 7) and when leaving it (line 11) to detect invalid insertions. A load profile example where such filtering is used is shown in Figure 3. It detects that the starts of partially inserted activities cannot be inserted everywhere in the sequence. This detection was not possible using the load profile from [26], illustrated in Figure 4: it only includes the capacity when leaving the node, which is always zero when no start is inserted. In this case, their algorithm produces an empty profile, which can be enhanced and more representative, as in Figure 3.

For the non-inserted activities, we use a similar pruning as [26]: we look at every possible insertions for the start of the activities and see if a matching end can be found. Start positions that cannot be closed and end positions for which no corresponding start can be found are removed. The time complexity is dominated by the complexity to check all the activities, which is $\mathcal{O}(|S| \cdot |A|)$.

Algorithm 3 `CumulFiltering`$(Sq, start, end, load, C, C^b, C^a)$ for partially inserted activities with end inserted.

Input : $Sq = \langle \overrightarrow{S}, I, P, E \rangle$: Sequence Variable, $start, end, load$: start, end and load of activities, C: capacity, C^b: minimum capacity before reaching a node, C^a minimum capacity after leaving a node.

```
1  for i | start_i ∉ S ∧ end_i ∈ S do
2      current ← (x ∈ S⃗ | x → end_i)
3      if C^a_current + load_i > C then
4          return failure
5      while current ≠ α do
6          if C^b_current + load_i > C then
7              remove all nodes x ∈ S⃗ | x ≺ current from I^{start_i}
8              break
9          current ← (x ∈ S⃗ | x → current)
10         if C^a_current + load_i > C then
11             remove all nodes x ∈ S⃗ | x ⪯ current from I^{start_i}
12             break
13 return success
```

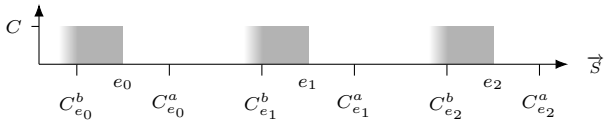

Figure 3 Load profile for a Sequence Variable with $\overrightarrow{S} = \{\alpha, e_0, e_1, e_2, \omega\}$, where only ends are inserted. Thanks to the computation of C^b in addition to C^a and the use of Algorithm 3, we can remove invalid insertions when considering the starts whose corresponding ends are inserted. This is the case for the start of activity 2 s_2 which cannot be inserted right after e_0, as the capacity before arriving at node e_1 would be exceeded. This case would have not been detected using only C^a, resulting in a load profile similar to Figure 4.

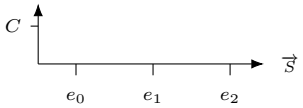

Figure 4 Load profile from [26] for a Sequence Variable with $\overrightarrow{S} = \{e_0, e_1, e_2\}$, where only ends are inserted. Because the accumulated capacity at each node is computed only when leaving a node, partially inserted activities might not contribute to the profile. In this case, it does not allow to detect that trying to insert s_2 after e_0 is invalid.

5 Experimental Results

The experiments reported in this section were conducted using two Intel(R) Xeon(R) CPU E5-2687W with 128GB of RAM. The Sequence Variable was implemented in MiniCP solver [21]. The source code is available for the readers in this anonymous repository [1], or by contacting the authors directly.

5.1 Dial-A-Ride Problem

We consider the problem described in [9, 15] and borrow the notations from [15]. This problem consists of m vehicles that must process n requests. Each request has a maximum ride time L, the vehicles have a maximum route duration D and the planning time is defined by a value T, representing the time at which the vehicles must be returned back to their origin. Each request i consists of an associated load and 2 nodes: a pickup $pickup_i$ and delivery $drop_i$ that must be visited one before the other. Each node i has an associated service duration $d_j \geq 0$ and belongs to one of two categories: *non-critical nodes*, having a time window $[0, T]$ and *critical nodes* having a tighter time window $[s_i, e_i]$ where $s_i \neq 0 \vee e_i \neq T$. Each activity is composed of exactly one critical vertex and one non-critical vertex and the set of all critical vertices is CV. The nodes define a complete graph: there is always a transition from one node to another.

A solution for the DARP consists of finding a route such that all vehicles begin and end at the depot; all requests are serviced; the maximum capacity of a vehicle is never exceeded; the pickup and corresponding delivery of a request are serviced by the same vehicle; for all requests i the difference between the arrival time at a $drop_i$ and the departure from $pickup_i$ never exceeds L; each node is visited within its time window. The objective consists of minimizing the routing cost: the sum of traveled distance by each vehicle.

This problem can be modeled easily by introducing one Sequence Variable per vehicle. Only a few constraints are required, the most important ones being a `Disjoint` constraint to ensure that nodes are visited once, a `TransitionTimes` to prevent visits of nodes outside of their time window and a `Cumulative` to respect the maximum capacity of each vehicle. We compare our results with [15], a state-of-the art approach for DARP and we use a similar branching strategy, shown in Algorithm 4.

We begin by computing the number of insertions for every request (line 7) as the product between the insertions for its critical node and for its non-critical node. This can be retrieved in constant time for one node x through n_s^x, introduced in section 3.3. We then select the request having the least possible insertions (line 8) and branch on every pair of insertions for its vertices (line 13). Those branching decisions are ordered by increasing value of a heuristic h for inserting a node $x \in P$ between nodes $i, j \in \overrightarrow{S} \mid i \rightarrow j$. This heuristic is defined in equations (14)-(16), and is similar to [15].

$$h(x, i, j) = \alpha \cdot costIncrease(x, i, j) - \beta \cdot slack(x, i, j) \tag{14}$$

$$costIncrease(x, i, j) = dist_{i,x} + dist_{x,j} - dist_{i,j} \tag{15}$$

$$slack(x, i, j) = \max(time_j) - \min(time_i) - dist_{i,x} - dist_{x,j} \tag{16}$$

Where $time_x$ is an integer variable denoting the serving time of node x and $dist_{i,j}$ the distance between nodes i and j. The values for α and β were kept from [15] and are set to 80 and 1, respectively. We also use Large Neighborhood Search with First Feasible Probabilistic Acceptance from [15]. For a fair comparison, we have implemented the COMET source code provided by the authors of [15] in Java. Although not able to run it in COMET, we could

Algorithm 4 Branching for DARP with a set SQ of Sequence Variables.

if *no unassigned requests left* **then**
  return solution
for $r \in$ unassigned request **do**
  $nInsert_r \leftarrow 0$
  $cv_r, ncv_r \leftarrow$ critical node and non-critical node from request r
  for $S \in SQ$ **do**
    $nInsert_r \leftarrow nInsert_r + nMemberInserts(S, cv_r) \cdot nMemberInserts(S, ncv_r)$
$r \leftarrow \text{argmin}\ \{nInsert_r \mid \forall r \in \text{unassigned request}\}$
$branching \leftarrow \{\}$
for $S \in SQ$ **do**
  for $p_{cv} \in getMemberInserts(S, cv_r)$ **do**
    for $p_{ncv} \in getMemberInserts(S, ncv_r)$ **do**
      $branching \leftarrow branching + (insert(S, cv_r, p_{cv}), insert(S, ncv_r, p_{ncv}))$
sort *branching* by increasing order of heuristic
return *branching*

obtain solution quality similar to the ones reported in [15] with the translated source-code. It is worth mentioning that the code for [15] is not generic, but custom and optimized for this sole problem. The filtering of the insertions is done during the search procedure rather than relying on the generic constraints executed in the fix-point of the solver.

We first compare the number of failures and solutions found using the exact search described in [15] without LNS with the Sequence Variable implementation. This comparison was made on a small instance with 2 vehicles and 20 requests and the corresponding results are reported in Table 2. We observe that the search from [15] finds all solutions to the instance in less time compared to the Sequence approach, which is ≈ 1.34 times slower. However, the number of failures is halved using Sequence Variables, resulting in a doubled ratio of solutions found per failure. This means that the approach performs better at removing invalid candidates to insert into the routes, although its filtering is slower.

Statistic	Tree Search [15]	Tree Search with **Sequence**
Time [s]	**974.545**	1307.447
Choices	153 864 380	**120 593 739**
Failures	70 033 356	**35 751 093**
Solutions	**66 700 800**	**66 700 800**
Failures / choices	0.455	**0.296**
Solutions / choices	0.434	**0.553**
Solutions / failure	0.952	**1.866**

Table 2 Statistics for finding all solutions on an instance with $m = 2$ vehicles and $n = 20$ requests, without using LNS. Choices refers to the number of branching decisions created during the search. Best result for each metric are shown in bold.

The next experiment compares the different approaches against instances with more requests and vehicles. It also includes a baseline comparison with a CP Optimizer model described in [26]. The solutions found are reported in Table 3 when an initial solution was provided. From the results, we see that our Sequence Variable does obtain results competitive with the approach from [15] with a slight advantage on smaller instances but not on larger ones. For some instances such as the one with $m = 8$ vehicles and $n = 108$ requests, finding a feasible solution is hard. This is why the Sequence Variables from CP Optimizer using a

black-box search that is not specific to this problem cannot always find a feasible solution. However, even when providing an initial solution, CP Optimizer sometimes fails to improve it, whereas our approach is able to get even better results by using it.

15 minutes run - no initial solution provided							
class a		LNS-FFPA		Sequence		CPO	
m	n	Mean	Best	Mean	Best	Mean	Best
3	24	191.59	191.40	**190.99**	**190.79**	198.19	198.19
4	36	**291.71**	**291.71**	294.48	292.75	313.33	313.33
5	48	308.26	305.98	**308.02**	**305.48**	t/o	t/o
6	72	531.59	522.00	**527.80**	**518.94**	t/o	t/o
7	72	553.26	546.63	**551.45**	**544.64**	t/o	t/o
8	108	**741.37**	**719.24**	780.45	758.63	t/o	t/o
9	96	625.09	616.47	**622.86**	**612.74**	t/o	t/o
10	144	**949.72**	**922.32**	1005.12	951.33	t/o	t/o
11	120	**696.33**	**683.64**	715.10	692.71	t/o	t/o
13	144	**878.10**	**863.15**	913.60	899.12	t/o	t/o
Avg.		**576.70**	**566.25**	590.99	576.71	t/o	t/o

15 minutes run - initial solution provided							
class a		LNS-FFPA		Sequence		CPO	
m	n	Mean	Best	Mean	Best	Mean	Best
3	24	191.76	191.40	**190.89**	**190.21**	196.11	196.00
4	36	**291.71**	**291.71**	294.72	292.72	318.97	318.97
5	48	308.95	306.97	**307.09**	**304.38**	327.37	327.00
6	72	532.55	524.97	**531.84**	**519.76**	579.79	579.77
7	72	**554.57**	550.42	554.65	**548.72**	614.02	614.00
8	108	**752.29**	**742.08**	794.86	755.00	924.04	923.86
9	96	**622.19**	614.65	625.68	**611.15**	740.26	740.26
10	144	**950.16**	**929.31**	1011.42	962.21	t/o	t/o
11	120	**699.32**	**687.99**	718.58	709.49	861.74	861.73
13	144	**878.33**	**864.81**	901.71	874.56	1042.82	1042.82
Avg.		**578.18**	**570.43**	593.14	576.82	t/o	t/o

Table 3 Comparison between our LNS-FFPA implementation from [15] (LNS-FFPA), our Sequence Variable implementation (Sequence) and the model using Sequence variables from CPoptimizer (CPO). 10 runs per solver were done on each instance, the best results are shown in bold. The left graph was produced when no initial solution was given, and the right when it was provided. Time-outs or no improving solution found are indicated by 't/o'.

5.2 Patient Transportation Problem

This problem, described in [5], is an extension of the Dial-A-Ride problem introduced in Section 5.1 with a few additional constraints. It considers the transport of patients to a hospital (described as one activity) and possibly back to a given location (another activity) by using a limited number of vehicles. The trip to the hospital must therefore always occur before the return trip and some patients can only be transported in a particular type of vehicle (patients in wheelchairs for instance). The objective consists of maximizing the number of transported patients.

We introduce one Sequence Variable per vehicle. We then use a `Cumulative` constraint to ensure that a vehicle never exceeds its maximum capacity and serves each activity as well as a `Precedence` constraint to guarantee that the trip to the hospital occurs before the transportation back home. As activities must be serviced within a specific time window, we use a `TransitionTimes` constraint and finally a `Disjoint` constraint to ensure that the patients are serviced at most once. For cases where a particular patient i can only be transported in a given type of vehicle t, we simply exclude all nodes n related to i from Sequence Variables whose related vehicle type is different from t. Our search and LNS uses works similarly to the one from Section 5.1, by inserting all nodes related to a patient (for their forward and possibly backward trip) before trying to serve another patient.

The comparison between our model and the results from [5] for the biggest available instances are reported in Table 4. We have used the same time-out as the one reported in their paper (30 minutes) and run their model on our setup, finding better solutions than the ones they reported. We observe that we are able to improve the number of serviced patients on the most difficult instances by using Sequence Variables.

5.3 Traveling Salesman Problem With Time Windows

TSPTW is a variant of the Traveling Salesman Problem (TSP) where all the customers must be visited within given time windows. Even finding a feasible solution was proved NP-complete [25]. As only one vehicle is available, the problem is modeled with a single Sequence Variable, the `TransitionTimes` constraint as well as a `Disjoint` constraint on the variable with the option that all nodes must be inserted.

Instances					SCHED+MSS	Sequence
Difficulty	Name	$\|H\|$	$\|V\|$	$\|R\|$	Sol	Sol
Easy	RAND-E-8	32	12	128	**128**	**128**
Easy	RAND-E-9	36	14	144	**144**	143
Easy	RAND-E-10	40	16	160	**158**	156
Medium	RAND-M-8	64	8	128	89	**91**
Medium	RAND-M-9	72	8	144	89	**93**
Medium	RAND-M-10	80	9	160	109	**113**
Hard	RAND-H-8	128	8	128	77	**87**
Hard	RAND-H-9	144	8	144	78	**84**
Hard	RAND-H-10	160	8	160	76	**84**

Table 4 Experimental results for the Patient Transportation Problem. $|H|$, $|V|$, $|R|$ are the number of hospitals, vehicles and requests, respectively. The objective is the number of patients serviced (Sol). SCHED+MSS refers to the best model from [5] while our own model is denoted as Sequence. Best results are shown in bold, the time-out was set to 30 minutes.

We use LNS to find better solutions over time. The relaxation used by LNS starts from an initial solution and consists of removing a set C of n consecutive nodes from the solution after a given node i. Those nodes are then only allowed to be inserted after node i or after another node in C. To achieve this, we remove the insertions $(p, q) \mid (p \neq i \vee p \notin C \vee q \notin C)$ from the sets of insertions I. Nodes not belonging to C are ordered according to their previous best found ordering.

Our LNS, described in Algorithm 5 uses the same structure as the one from [15]. It starts from an initial solution *initSol* and relaxes an increasing number of nodes $n = i + j$ (line 7) from it. This process is done *numIter* times before increasing the number of relaxed nodes. $minSize, maxSize$ and $range$ provide bounds for the number of nodes that needs to be relaxed. During our experiments, we have set $minSize = 10$, $maxSize =$ number of nodes in the problem, $range = 5$ and $numIter = 300$. The branching procedure at line 8 uses a similar branching as the one from the DARP: the non-inserted node x with the least number of member insertions $n_s^x = |I^x \cap S|$ is selected and branched on according to a heuristic that is the same as equation (14).

Algorithm 5 LNS($Sq = \langle \overrightarrow{S}, I, P, E \rangle, initSol, minSize, maxSize, range, numIter, dist, timeLimit$).

```
1  bestSol ← initSol
2  for i ∈ {minSize ... (maxSize − range)} do
3  |  if i = maxSize-range then
4  |  |  i ← minSize
5  |  for j ∈ {0 ... range − 1} do
6  |  |  for k ∈ {1 ... numIter} do
7  |  |  |  relax(i + j) consecutive nodes from bestSol
8  |  |  |  sol ← optimize(dist)
9  |  |  |  if the solution has been improved then
10 |  |  |  |  bestSol ← sol
11 |  |  |  if timeLimit is reached then
12 |  |  |  |  return bestSol
```

We tested the model on three sets of instances from [20], referred to as *OhlmannThomas*, *AFG* and *GendreauDumasExtended* and adapted from [22, 11] for the first set, [2] for the

second set and from [13, 11] for the third set. The number of nodes in those instances varies from 20 to 232. The LNS was initialized from the best known solutions reported on [20]. The improved solutions were tested using the checker from [20] to ensure their feasibility as well as their cost.

A set of 20 of the 25 instances from the OhlmannThomas set could be improved, 10 of the 50 instances from the AFG set and 2 of the 130 instances from the GendreauDumasExtended set. The new objective solutions as well as the solving time to reach them are reported in Table 5. From our experiments we observe that we converge to a new solution sometimes rapidly (6 new best solutions are reached in less than 5 seconds and not improved afterwards) whereas some instances benefit more from the increasing number of nodes relaxed in the LNS and are still improved after a longer period of time.

Instance	Previous	New	Time [s]
n150w120.001	735	734	0.50
n150w120.002	683	679	290.86
n150w120.003	748	747	41.45
n150w120.005	692	689	7.84
n150w140.001	767	762	134.96
n150w140.002	757	755	34.28
n150w140.003	620	613	64.28
n150w140.004	677	676	12.94
n150w140.005	665	663	167.10
n150w160.001	708	706	2.64
n150w160.002	712	711	162.35
n150w160.003	610	608	0.49
n200w120.001	801	799	194.56
n200w120.002	725	722	12.12
n200w120.003	885	880	51.44
n200w120.005	843	841	202.69
n200w140.001	837	834	35.23
n200w140.002	768	765	14.77
n200w140.003	764	758	298.95
n200w140.005	827	822	33.21

Instance	Previous	New	Time [s]
rbg132.2	8200	8194	37.76
rbg132	8470	8468	0.76
rbg201a	12 967	12 948	152.53
rbg233.2	14 549	14 523	24.20
rbg092a	7160	7158	2.70
rbg152.3	9797	9796	0.41
rbg193.2	12 167	12 159	242.54
rbg193	12 547	12 538	55.57
rbg233	15 031	14 994	264.70
rbg172a	10 961	10 956	113.83

Instance	Previous	New	Time [s]
n80w120.005	597	591	9.09
n100w160.005	587	586	19.59

Table 5 Improved routing cost values found for the TSPTW instances. Previous best objective values (Previous) are retrieved from [20]. New best objective values discovered are indicated (New) as well as the time to reach them. The time-out was set to 5 minutes. The Table on the left shows the values for the OhlmannThomas instances, the top right for the AFG instances and the bottom right for the GendreauDumasExtended.

6 Conclusions and future work

This paper introduced a simplified version of the Sequence domain introduced in [26] as a flexible and effective approach for modeling and solving VRP with CP. The filtering algorithms for constraints imposing time windows and vehicle capacity are described. Experimental results on three problems show that our models are competitive with existing sequence based approaches while being effective enough to discover new best solutions to a well-studied problem such as the TSPTW. Our proposed filtering algorithms are relatively simple and could most certainly be improved. We plan to enhance them and study the use of Sequence Variables in more vehicle routing problems, as well as scheduling problems.

References

1 MiniCP Sequences - Anonymous GitHub, Sep 2021. [Online; accessed 26. Feb. 2022]. URL: https://anonymous.4open.science/r/minicp-sequences-5EE3/README.md.

2 Norbert Ascheuer. *Hamiltonian Path Problems in the On-line Optimization of Flexible Manufacturing Systems*. PhD thesis, 1996.

3 Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby, and Patrick Prosser. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of heuristics*, 6(4):501–523, 2000.

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

5 Quentin Cappart, Charles Thomas, Pierre Schaus, and Louis-Martin Rousseau. A constraint programming approach for solving patient transportation problems. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 490–506, Cham, 2018. Springer International Publishing.

6 IBM Knowledge Center. Interval variable sequencing in CP Optimizer, Mar 2021. [Online; accessed 13. Jan. 2022]. URL: https://www.ibm.com/docs/en/icos/12.9.0?topic=concepts-interval-variable-sequencing-in-cp-optimizer.

7 IBM Knowledge Center. Search API for scheduling in CP Optimizer, Mar 2021. [Online; accessed 13. Jan. 2022]. URL: https://www.ibm.com/docs/en/icos/12.9.0?topic=c-search-api-scheduling-in-cp-optimizer#85.

8 Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of operations research*, 153(1):29–46, 2007.

9 Jean-François Cordeau and Gilbert Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological*, 37:579–594, 07 2003. doi:10.1016/S0191-2615(02)00045-0.

10 Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

11 Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367–371, 1995.

12 Interreg Europe. Demand-responsive transport. https://www.interregeurope.eu/sites/default/files/2021-12/Policy%20brief%20on%20demand%20responsive%20transport.pdf, June 2018.

13 Michel Gendreau, Alain Hertz, Gilbert Laporte, and Mihnea Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335, 1998.

14 Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1:191–244, 03 1997.

15 Siddhartha Jain and Pascal Van Hentenryck. Large neighborhood search for dial-a-ride problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 400–413. Springer, 2011.

16 Philippe Laborie and Jerome Rogerie. Reasoning with conditional time-intervals. In *FLAIRS conference*, pages 555–560, 2008.

17 Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. Reasoning with conditional time-intervals. part ii: An algebraical model for resources. In *FLAIRS Conference*, 2009.

18 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, apr 2018. doi:10.1007/s10601-018-9281-x.

19 Chang Liu, Dionne M. Aleman, and J. Christopher Beck. Modelling and solving the senior transportation problem. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 412–428, Cham, 2018. Springer International Publishing.

20 Manuel López-Ibáñez. Instances for the TSPTW, Sep 2020. [Online; accessed 15. Feb. 2022]. URL: https://lopez-ibanez.eu/tsptw-instances.

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

22 Jeffrey W Ohlmann and Barrett W Thomas. A compressed-annealing heuristic for the traveling salesman problem with time windows. *INFORMS Journal on Computing*, 19(1):80–90, 2007.

23 Laurent Perron and Vincent Furnon. Or-tools. URL: https://developers.google.com/optimization/.

24 Laurent Perron and Vincent Furnon. Or-tools sequence var. URL: https://developers.google.com/optimization/reference/constraint_solver/constraint_solver/SequenceVar.

25 Martin WP Savelsbergh. Local search in routing problems with time windows. *Annals of Operations research*, 4(1):285–305, 1985.

26 Charles Thomas, Roger Kameugne, and Pierre Schaus. Insertion sequence variables for hybrid routing and scheduling problems. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 457–474. Springer, 2020.

27 Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. SIAM, 2002.