Insertion Sequence Variables for Hybrid Routing and Scheduling Problems

Charles Thomas[1][0000-0002-7360-5372], Roger Kameugne[2][0000-0003-1809-9822], and Pierre Schaus[1][0000-0002-3153-8941]

[1] UCLouvain, Belgium {name.surname}@uclouvain.be
[2] Faculty of Sciences, University of Maroua, Cameroon rkameugne@gmail.com

Abstract. The Dial a Ride family of Problems (DARP) consists in routing a fleet of vehicles to satisfy transportation requests with time-windows. This problem is at the frontier between routing and scheduling. The most successful approaches in dealing with DARP are often tailored to specific variants. A generic state-of-the-art constraint programming model consists in using a sequence variable to represent the ordering of visits in a route. We introduce a possible representation for the domain called *Insertion Sequence Variable* that naturally extends the standard subset bound for set variables with an additional insertion operator after any element already sequenced. We describe the important constraints on the sequence variable and their filtering algorithms required to model the classical DARP and one of its variants called the Patient Transportation Problem (PTP). Our experimental results on a large variety of instances show that the proposed approach is competitive with existing sequence based approaches.

1 Introduction

Door-to-door transportation services and on demand public transport are increasingly important due to the flexibility it offers to the customers. Two such problems are the Dial a Ride Problem (DARP) [7] and the Patient Transportation Problem (PTP) [3] which consist in transporting a maximum of patients to and from medical appointments. These problems often involve large number of requests and thus require efficient algorithmic solutions. Many approaches have been proposed and applied successfully to different variants of the DARP [11]. However, these solutions are often tailored towards specific use-cases and are difficult to adapt to other variants of the problem. It is thus crucial to develop approaches to model and solve efficiently such problems while remaining generic and easily adaptable.

We describe an *Insertion Sequence Variable* (ISV) for modeling and solving DARPs. Our domain representation includes the subset bound domain [9] for set variables. This allows to represent optional elements in the domain and prevents a repetition of the same element at different positions in the sequence. The set domain is extended with an internal sequence that can be grown with arbitrary insertions available from a set of possible insertions. By letting the constraints remove impossible insertions, the search space is pruned by restricting the set of possible sequences. We describe two important global constraints on the Insertion Sequence Variable for modeling the DARP and PTP: 1) The `TransitionTimes` constraint links a sequence variable with time interval

variables to take into account a transition time matrix between consecutive elements in the sequence. 2) The `Cumulative` constraint ensures that the load profile does not exceed a fixed capacity when pairs of elements in the sequence represent the load and discharge on a vehicle. We experimentally test the performances of the Insertion Sequence Variable on the two problems and show that it is competitive with the state-of-the-art CP approaches.

2 Related Work

In [14], the authors propose a constraint-based approach called *LNS-FFPA* to solve DARPs with a cost objective and show that it outperforms other state-of-the-art approaches. While highly efficient, the LNS-FFPA algorithm is difficult to adapt to other variants of the DARP such as the PTP. Indeed, the approach is not declarative since some constraints are enforced with the search. Furthermore, the model is not able to deal with optional visits that occur in the PTP and similar problems.

Two recent approaches for solving the PTP are [3] and [20]. The approach proposed in [3] consists in representing the problem with a scheduling model where trips are represented by activities. The approach of [20] is based on IBM ILOG CP Optimizer solver [19]. It makes use of the sequence variables from CP Optimizer to decide the order of visits in each vehicle.

The high level functionalities and constraints related to sequence variables of CP Optimizer have been briefly described in [17, 18]. Unfortunately, no details are given on the implementation of such variables and the filtering algorithms of the constraints in the literature. According to the API and documentation available at [12, 13], the sequence variable of CP Optimizer is based on a Head-Tail Sequence Graph structure. It consists of maintaining separate growing head and tail sub-sequences. Interval variables not yet sequenced can be added either at the end of the head or at the beginning of the tail. When no more interval variable can be added and all members of the head and tail are decided, both sub-sequences are joined to form the final sequence. Google OR-Tools [22] also propose sequence variables [23] with the same approach as CP Optimizer. The approach proposed in this paper differs from the one of CP Optimizer in the following ways: 1) the insertion sequence variable is generic and usable in a large variety of problems. In particular, the variable is independent of the notion of time intervals; 2) insertions are allowed at any point in the sequence which allows flexible modeling and search; 3) the variable proposed keeps track of the possible insertions for each element inside its domain which allows advanced propagation techniques.

In [10], the authors discuss the usage of a path variable in the context of Segment Routing Problems. Their implementation is based on a growing prefix to which candidates elements can be appended.

3 Preliminary

Let $X = \{0, ..., n\}$ be a finite set and $\mathcal{P}(X)$ the set of subsets (power set) of X. The inclusion $\subseteq$ relation defines a partial order over $\mathcal{P}(X)$ and the structure $(\mathcal{P}(X), \subseteq)$ is a lattice generally used to represent the domain of a finite set variable. To avoid

explicit exhaustive enumeration of set domain, three disjoint subsets of X are used to represent the current state of the set domain (see [9]). The domain is defined as $\langle P, R, E\rangle \equiv \{S' \mid S' \subseteq X \wedge R \subseteq S' \subseteq R \cup P\}$ where P, R, and E denote respectively the set of Possible, Required and Excluded elements of X. At any time we have that P, R and E form a partition of X. The variable S with domain $\langle P, R, E\rangle$ is bound if P is empty. Table 1 contains the supported operations on a set variable S of domain $\langle P, R, E\rangle$ with their complexity.

Table 1: Operations supported by set variables

Operation	Description	Complexity
`requires`(S, e)	move e to R, fails if $e \in E$	$\Theta(1)$
`excludes`(S, e)	move e to E, fails if $e \in R$	$\Theta(1)$
`isBound`(S)	return true iff S is bound	$\Theta(1)$
`is`$\{$`Possible/Required/Excluded`$\}(S, e)$	return true iff $e \in \{P/R/E\}$	$\Theta(1)$
`all`$\{$`Possible/Required/Excluded`$\}(S)$	enumerate $\{P/R/E\}$	$\Theta(\lvert\{P/R/E\}\rvert)$

We denote by $\overrightarrow{S}$ a *sequence* without duplicates over X ($S \subseteq X$). The sequence $\overrightarrow{S}$ defines an order over the elements of S. Each element of X is unique and can appear only once in S. The set of all sequences of X is denoted by $\overrightarrow{\mathcal{P}}(X)$. Let a and b be two elements of S. The relation a *precedes* b *in* $\overrightarrow{S}$ is noted $a \overset{\overrightarrow{S}}{\prec} b$ or $a \prec b$ when it is clear from the context that the relation applies in regards to S. The relation a *directly precedes* b *in* $\overrightarrow{S}$ is noted $a \xrightarrow{\overrightarrow{S}} b$ or or $a \rightarrow b$ when clear from the context. In this case, b is called the *successor* of a and a is called the *predecessor* of b in $\overrightarrow{S}$. A sequence $\overrightarrow{S}'$ is a *super-sequence* of $\overrightarrow{S}$ if $S \subseteq S'$ and $\forall a, b \in S, a \overset{\overrightarrow{S}}{\prec} b \implies a \overset{\overrightarrow{S'}}{\prec} b$. This relationship is noted $\overrightarrow{S} \subseteq \overrightarrow{S}'$. Conversely, $\overrightarrow{S}$ is a *sub-sequence* of $\overrightarrow{S}'$.

The *insertion* operation $insert(\overrightarrow{S}, e, p)$ consists in inserting the element e in the sequence $\overrightarrow{S}$ after the element p where $e \in X \setminus S$ and $p \in S$. Performing this operation results in a super-sequence $\overrightarrow{S}'$ of $\overrightarrow{S}$ such that $S' = S \cup \{e\}$ and $p \xrightarrow{\overrightarrow{S'}} e$. The operation is also noted $\overrightarrow{S} \underset{(e,p)}{\Longrightarrow} \overrightarrow{S}'$.

The insertion of an element e at the beginning of a sequence or in an empty sequence is defined as $insert(\overrightarrow{S}, e, \bot)$. An insertion in a sequence $\overrightarrow{S}$ is thus characterized by a tuple (e, p) where $e \notin S$ and $p \in S \vee p = \bot$.

Given I, a set of tuples, each corresponding to a potential insertion in $\overrightarrow{S}$, the *one-step derivation* $\overrightarrow{S} \underset{I}{\Longrightarrow} \overrightarrow{S}'$ between a sequence $\overrightarrow{S}$ and its super-sequence S' is defined as $\overrightarrow{S} \underset{I}{\Longrightarrow} \overrightarrow{S}' \iff \exists i = (e, p) \in I \mid \overrightarrow{S} \underset{i=(e,p)}{\Longrightarrow} \overrightarrow{S}'$. In other words, the sequence S is transformed into S' by applying one possible insertion from I. More generally *zero or more steps derivation* is defined as $\overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' \equiv \overrightarrow{S} = \overrightarrow{S}' \vee \left(\exists i \in I \mid \overrightarrow{S} \underset{i}{\Longrightarrow} \overrightarrow{S}'' \wedge \overrightarrow{S}'' \overset{*}{\underset{I \setminus \{i\}}{\Longrightarrow}} \overrightarrow{S}'\right)$. 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 a same super-sequence.

Example 1. Let us consider the sequence $\overrightarrow{S} = (1, 2, 3)$ and the set of insertions $I = \{(4, 2), (5, 2), (5, 4)\}$. We have that $\overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' = (1, 2, 4, 5, 3)$ since it can be obtained with consecutive derivations over I: (1, 2, 3) $\underset{(4,2)}{\Longrightarrow}$ (1, 2, 4, 3) $\underset{(5,4)}{\Longrightarrow}$ (1, 2, 4, 5, 3).

4 Insertion Sequence Variable

Definition 1. *An insertion sequence variable Sq on a set X is a variable whose domain is represented by a tuple $\langle \overrightarrow{S}, I, P, R, E\rangle$ where $\langle P, R, E\rangle$ is the domain of a set variable on X, $\overrightarrow{S}$ is a sequence $\in \overrightarrow{\mathcal{P}}(R)$ and I is a set of tuples (e, p), each corresponding to a possible insertion. The domain of Sq, also noted $D(Sq)$, is defined as*

$$\langle \overrightarrow{S}, I, P, R, E\rangle \equiv \left\{ \overrightarrow{S}' \in \overrightarrow{\mathcal{P}}(P \cup R) \mid R \subseteq S' \wedge \overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' \right\} \tag{1}$$

Sq is bound if P is empty and $|S| = |R|$. Initially, all elements of the domain are optional ($\in P$). During the search, elements can be set as mandatory or excluded (moved to R or E) and possible insertions can be removed from I.

Lemma 1. *Checking the consistency of the domain $\langle \overrightarrow{S}, I, P, R, E\rangle$ is NP-complete.*

Proof. It requires verifying the following properties: $\exists \overrightarrow{S}' \mid \overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' \wedge R \subseteq S'$ and $\forall e \in P, \exists S' \mid \overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' \wedge R \cup \{e\} \subseteq S'$. The Hamiltonian path problem for a directed graph $G = (\mathcal{V}, \mathcal{E})$ can be reduced to checking the consistency of the domain $D(Sq) = \langle \overrightarrow{S} = (), I = \mathcal{E}_{reverse} \cup \{(v, \perp) \mid \forall v \in \mathcal{V}\}, P = \emptyset, R = \mathcal{V}, E = \emptyset\rangle$ where $\mathcal{E}_{reverse}$ is the result of applying the *reverse* operation on each edge $(i, j) \in \mathcal{E}$ defined as $(i, j)_{reverse} = (j, i)$. □

Consequently, instead of checking the full domain consistency at each change in the domain, the following invariant is maintained internally by the sequence variable:

$$P \cup R \cup E = X \wedge P \cap R = R \cap E = P \cap E = \phi \tag{2}$$

$$S \subseteq R \tag{3}$$

$$\forall (e, p) \in I, e \notin S \wedge e \notin E \wedge p \notin E \tag{4}$$

$$\forall p \in S, \nexists (e, p) \in I \implies e \in E \tag{5}$$

At any moment: $P \cup R \cup E$ form a partition of X (2); any member of $\overrightarrow{S}$ is required (3); any member of $\overrightarrow{S}$ cannot be inserted in $\overrightarrow{S}$; any excluded element cannot be inserted in $\overrightarrow{S}$ and is not a valid predecessor (4); any element that cannot be inserted at any position in $\overrightarrow{S}$ is excluded (5).

Example 2. Let us consider $X = \{a, b, c, d, e, f\}$, the variable Sq of domain $\langle \overrightarrow{S} = (f, b), I = \{(c, \perp), (c, e), (c, f), (e, c), (e, f)\}, P = \{c\}, R = \{b, e, f\}, E = \{a, d\}\rangle$ corresponds to the sequences $\{(f, e, b), (c, f, e, b), (f, c, e, b), (f, e, c, b)\}$. The sequences $\{(f, b), (c, f, b), (f, c, b)\}$ are not valid as they do not contain e which is required.

The insertion sequence variable inherits all the operations defined on the set variable (see Table 1) and supports the additional operations summarized in Table 2.

Table 2: Operations supported by insertion sequence variables

Operation	Description	Complexity
isBound(Sq)	return true iff Sq is bound	$\Theta(1)$
isMember(Sq, e)	return true iff e is present in $\overrightarrow{S}$	$\Theta(1)$
allMembers(Sq)	enumerate $\overrightarrow{S}$	$\Theta(\|S\|)$
allCurrentInserts(Sq)	enumerate $\{(e,p) \in I \mid p \in S\}$	$\mathcal{O}(\min(\|I\|, \|S\|))$
nextMember(Sq, e)	return the successor of e in $\overrightarrow{S}$	$\Theta(1)$
insert(Sq, e, p)	insert e in $\overrightarrow{S}$ after p, update P, R and I, fail if $e \in E \vee p \notin S$	$\Theta(1)$
canInsert(Sq, e, p)	return true iff $(e,p) \in I$	$\Theta(1)$
allInserts(Sq)	enumerate I	$\Theta(\|I\|)$
remInsert(Sq, e, p)	remove (e,p) from I	$\Theta(1)$

4.1 Implementation

The implementation of the internal set variable $\langle P, R, E\rangle$ uses array-based sparse sets as in [24] to ensure efficient update and reversibility during a backtracking depth-first-search. It consists of an array of length $|X|$ called `elems` and two reversible integers: `r` and `p`. The position of the elements of X in `elems` indicates in which subset the element is. Elements before the position `r` are part of R while elements starting from position `p` are part of E. Elements in between are part of P. An array called `elemPos` maps each element of X with its position in `elems`, allowing access in $\Theta(1)$.

The internal partial sequence $\overrightarrow{S}$ is implemented using a reversible chained structure. An array of reversible integers called `succ` indicates for each element its successor in the partial sequence. An element which is not part of the partial sequence points towards itself. An additional dummy element $\perp$ marks the start and end of the partial sequence. It can be specified as predecessor in the insertion operation to insert an element at the beginning of the sequence or in an empty sequence. Inserting an element e in the partial sequence after p consists in modifying the successor of e to point to the previous successor of p and modifying the successor of p to point to e.

The set of possible insertions I is implemented using an array of sparse sets called `posPreds`. For each element, the corresponding sparse set contains all the possible predecessors after which the element can be inserted. If the element is a member of the sequence $\overrightarrow{S}$ or excluded, its set is empty. The sparse sets are initialized with the following domain: $R \cup P \cup \{\perp\}$. Constraints may remove possible insertions during their propagation. If doing so results in an empty set, the corresponding element is excluded according to the invariant (5).

An illustration of the domain representation for the variable Sq with a domain of $\langle \overrightarrow{S} = (f,b), I = \{(c,\perp),(c,e),(c,f),(e,c),(e,f)\}, P = \{c\}, R = \{b,e,f\}, E = \{a,d\}\rangle$ is given in Figure 1.

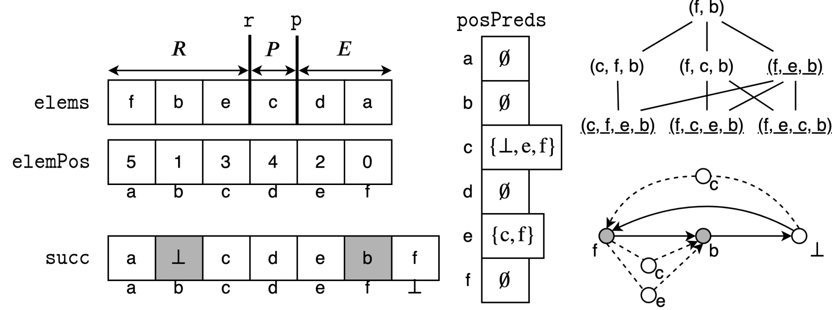


Fig. 1: The insertion sequence variable domain $\langle\langle \overrightarrow{S} = (f, b), I = \{(c, \bot), (c, f), (c, e), (e, c), (e, f)\}, P = \{c\}, R = \{b, e, f\}, E = \{a, d\}\rangle$ (left and middle) and the corresponding lattice (with valid sequences underlined) and graphical representation (right)

5 Global Constraints on Insertion Sequence Variables

5.1 Transition Times constraint

In a scheduling context, the elements to sequence correspond to activities performed over time, each associated with a time window and requiring a minimum transition time to move to the next that depends on the pair of consecutive activities. The `Transition Times` constraint links the sequenced elements with their time window to make sure that transition time constraints are satisfied between any two consecutive elements of the sequence. More formally, each element $i \in X$ is associated with an activity defined by a start $start_i$ and a duration variable dur_i. A matrix $trans_{i,j}$, satisfying the triangle inequality, specifies transition times associated to each couple of activities (i, j). The `TransitionTimes` constraint is then defined as

$$\texttt{TransitionTimes}(Sq, [start], [dur], [[trans]]) \equiv$$
$$\left\{ \overrightarrow{S}' \in D(Sq) \mid \forall a, b \in S', a \overset{\overrightarrow{s'}}{\prec} b \implies start_b \geq start_a + dur_a + trans_{a,b} \right\} \quad (6)$$

Filtering The filtering algorithm is triggered whenever an element is either inserted in $\overrightarrow{S}'$ or required or if one of the bounds of a time window changes. The algorithm is split into three parts: *time windows update*, *insertion update* and *feasible path checking and filtering*.

Time window update This filtering algorithm is used to adjust the start and duration of the activities already present in $\overrightarrow{S}$. This update is done in linear time by iterating over the elements of the sequence and updating their time windows depending on the time needed to transition from the previous element and to the next element. If the time window of an element is shrunk outside its domain, this leads to a failure.

Insertion update This filtering algorithm is used to filter out the invalid insertions in I based on the current state of $\overrightarrow{S}$ and the transition times of the activities. The algorithm is linear and consists in iterating over I. For each possible insertion, if the transition times between the inserted activity and its predecessor and successor lead to a violation of a time window, the insertion is removed.

Feasible Path checking and filtering The problem of verifying that there exists at least one *transition time feasible* extension of the current sequence composed of the required activities not yet inserted is NP-Complete [8] by a reduction from the TSP. Algorithm 1 is a recursive depth first search used to check that there exists at least one feasible extension of the current sequence composed of the required activities not yet inserted (i.e. in the set $R \setminus S$). Given the current sequence $\overrightarrow{S}$, the recursive call `feasiblePath`(ℓ, p, Ω, t, d) checks that it is possible to build a sequence starting from ℓ at time t that contains at least d elements of Ω and is a super-sequence of the sub-sequence of $\overrightarrow{S}$ starting in p. The parameter ℓ indicates the last element visited at time t whereas the parameter p indicates the last element of S that has been visited (possibly several steps before ℓ). The initial call `feasiblePath`$(\ell = \bot, p = \bot, \Omega = R \setminus S, t = 0, d)$ thus checks that there exists a super-sequence of $\overrightarrow{S}$ containing d elements of $R \setminus S$.

Algorithm 1: `feasiblePath`(ℓ, p, Ω, t, d)

Input: ℓ: last element reached, p: previous element reached in $\overrightarrow{S}$, Ω: set of elements to reach, t: departure time from ℓ, d: depth, $Sq = \langle \overrightarrow{S}, I, P, R, E \rangle$: Sequence variable, cache: memoization map

1. $n \leftarrow$ `nextMember`(Sq, p) ;
2. **if** $n \neq \bot$ *and* $t + trans_{\ell,n} > \max(start_n)$ **then**
3.   **return** false;
4. **if** $\Omega = \emptyset$ **then**
5.   **return** true;
6. $(t_f, t_i) \leftarrow$ cache.`getOrElse`$((\ell, p, \Omega), (-\infty, +\infty))$;
7. **if** $t \leq t_f$ **then return** true;
8. **if** $t \geq t_i$ **then return** false;
9. **for** $a \in \Omega$ **do**
10.   **if** $t + trans_{\ell,a} > \max(start_a)$ **then**
11.   cache.`update`$((\ell, p, \Omega), (t_f, \min(t_i, t)))$;
12.   **return** false ; `// pruning (infeasible sequence)`
13. **if** $d \leq 0$ **then**
14.   **return** true ; `// pruning (maximum depth reached)`
15. **else**
16.   **for** $a \in \Omega \mid (a, \ell) \in I$, *sorted in increasing* $(\min(start_a) + \min(dur_a))$ **do**
17.   **if** *feasiblePath*$(a, p, \Omega \setminus \{a\}, \max(t + trans_{\ell,a}, \min(start_a) + \min(dur_a)), d - 1)$ **then**
18.   cache.`update`$((\ell, p, \Omega), (\max(t_f, t), t_i))$;
19.   **return** true;
20.   **if** $n \neq \bot$ *and* *feasiblePath*$(n, n, \Omega, \max(t + trans_{\ell,n}, \min(start_n) + \min(dur_a)), d)$ **then**
21.   cache.`update`$((\ell, p, \Omega), (\max(t_f, t), t_i))$;
22.   **return** true;
23.   **return** false;

At each node, the algorithm either explores the insertion of a new element after ℓ which corresponds to branching over an element of Ω (line 16) or follows the current sequence $\overrightarrow{S}$ which consists in branching over the successor of p in $\overrightarrow{S}$ (line 20). A pruning is done at lines 2 and 9 if one realizes that at least one activity cannot be reached. By the triangle inequality assumption of the transition times, if either the successor of p or at least one activity of Ω cannot be reached directly after ℓ, then it can surely not be reached later in time if some activities were visited in between. Therefore `false` is returned in such case which corresponds to the infeasibility pruning. The possible extensions considered recursively at line 16 are based on the current state of I and the value of p. The maximum depth is controlled by the parameter d to avoid prohibitive computation. The algorithm can thus return a false positive result by returning `true` at line 14 if this limit is reached.

The time complexity of Algorithm 1 is $\mathcal{O}(|S| \cdot |\Omega|^d)$ in worse case as it corresponds to an iteration over $\overrightarrow{S}$ with a depth-first search of depth d and branching factor $|\Omega|$ at each step. In practice, as the branching is based on I, the search tree will often be smaller. In order to reduce the time complexity of the successive calls to `feasiblePath`, a cache is used to avoid exploring several times a partial extension that can be proven infeasible or feasible based on previous executions. A global map called `cache` is assumed to contain keys composed of the arguments of the function, that is a tuple with (Ω, ℓ, p). At each key, the map associates a couple of integer values (t_f, t_i) where t_f is the latest known time at which it is possible to depart from ℓ and find a feasible path among the sub-sequence starting after p and the activities of Ω and t_i is the earliest known time at which the departure from ℓ is too late and there exists no feasible path. Line 6 is called to find if a corresponding entry exists in the map. If it is the case, the departure time t is compared to the couple (t_f, t_i) of the map. If $t \leq t_f$, the value *true* is immediately returned. If $t \geq t_i$, *false* is returned. If $t_f < t < t_i$, the algorithm continues its exploration. The cache is updated at lines 11, 18 and 21 depending on the result found. Usage of the cache is highlighted in gray in Algorithm 1.

This checking algorithm can be used in a shaving-like fashion into Algorithm 2. A value is filtered out from the possible set if its requirement made the sequencing infeasible according to the transition times. This `TransitionTimesFiltering` algorithm executes in $\mathcal{O}(|P| \cdot (|S| \cdot |R \setminus S|)^d)$. Notice that the cache is shared and reused along the calls in order to avoid many subtree explorations. Due to the extensive nature of the algorithm, a parameter ρ defines a threshold for the size of P above which the `feasiblePath` algorithm is not executed for each element of P (line 4).

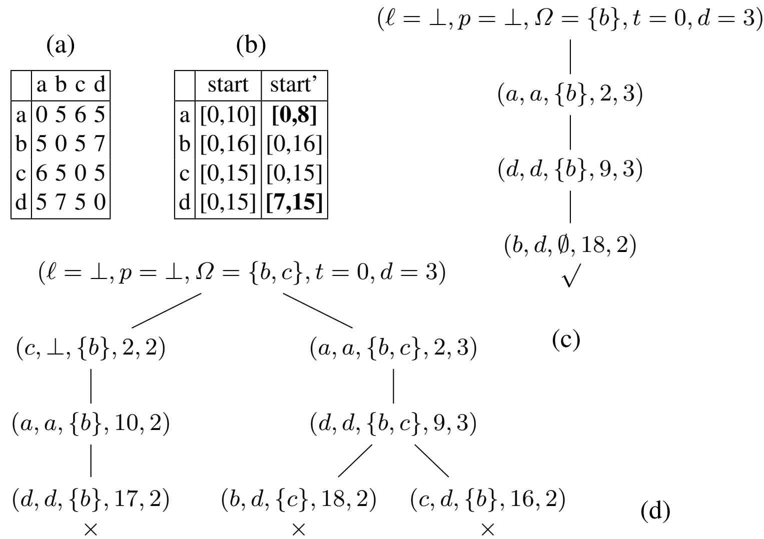
Algorithm 2: `TransitionTimesFiltering`(Sq, d, ρ)

Input: d: maximum depth, ρ: filtering threshold, $Sq = \langle \overrightarrow{S}, I, P, R, E \rangle$: seq. variable

```
1 cache ← map() ; // initializing memoization map
2 if !feasiblePath(⊥, ⊥, R \ S, 0, d) then
3 |  return failure ;
4 if |P| ≤ ρ then
5 |  forall a ∈ P do
6 |  |  if !feasiblePath(⊥, ⊥, (R \ S) ∪ {a}, 0, d) then
7 |  |  |  excludes(Sq, a);
```

Example 3. Let us consider the following example where $X = \{a, b, c, d\}$ is the set of activities. The transition times between activities are given in Table (a) of Fig. 2 and the initial time windows (column start) in Table (b) of Fig. 2. We consider the sequence variable Sq of domain $\langle \overrightarrow{S} = (a, d), I = \{(b, a), (b, d), (c, \bot), (c, d)\}, P = \{c\}, R = \{a, b, d\}, E = \emptyset\rangle$. The duration of each activity is fixed at 2. Let us apply the propagation of `TransitionTimes` on this example:

1. *Time window update* is applied. The time windows of a and d are reduced. The updated time windows are displayed in Table (b) (column start').
2. *Insertion update* is applied. The insertion (b, a) is removed from I as b cannot be inserted after a without violation (b would end at the earliest at 9 which implies that d would start at the earliest at 16, outside its time window).
3. *Transition Time Filtering* (Algorithm 2) is applied. The search trees for the checker (c) and the filter (d) are displayed in Figure 2. Failures are denoted with $\times$ and successes with $\surd$. The initial value of the parameter d is 3. The domain of Sq after propagation is $\langle (a, d), \{(b, d)\}, \emptyset, \{a, b, d\}, \{c\}\rangle$ as the filter excludes c.


(a)

	a	b	c	d
a	0	5	6	5
b	5	0	5	7
c	6	5	0	5
d	5	7	5	0

(b)

	start	start'
a	[0,10]	**[0,8]**
b	[0,16]	[0,16]
c	[0,15]	[0,15]
d	[0,15]	**[7,15]**

Fig. 2: Propagation on $Sq = \langle (a, d), \{(b, a), (b, d), (c, \bot), (c, d)\}, \{c\}, \{a, b, d\}, \emptyset\rangle$

5.2 Cumulative constraint

In both the DARP and PTP, one has to satisfy requests that correspond to embarking and disembarking a person in a vehicle. The activities of transport are modeled as pairs of elements in an insertion sequence variable that must occur in this specific order:

embarking before disembarking. Also this pair of elements must both be present or absent from the sequence. During the trip, the person occupies some load in the vehicle. By analogy to scheduling problems, a request is called an activity A_i and is composed of the two elements ($start_i$,end_i) corresponding to the embarking and disembarking. This activity will consume a load $load_i$ while it is on the board of the vehicle. The set of activities is denoted A. Also by analogy to scheduling [1], we call `Cumulative` the constraint that ensures that the capacity C of the resource is respected at any point in the ordering defined by the sequence Sq over X where $\forall i \in A, start_i, end_i \in X$. More formally

$$\texttt{Cumulative}(Sq, [start], [end], [load], C) \equiv \left\{ \overrightarrow{S}' \in D(Sq) \mid \forall e \in S', \sum_{i \in A | start_i \preceq e \preceq end_i} load_i \leq C \right\}. \quad (7)$$

Filtering The propagation is triggered when new elements are inserted in $\overrightarrow{S}$. It consists in filtering insertions in the current sequence $\overrightarrow{S}$ by checking if they are supported. An insertion for the element corresponding to one extremity of an activity is supported if there exists at least one possible insertion for the other extremity of the activity such that the activity load does not overloads the capacity between both insertion positions.

The first step of the propagation algorithm is to build a minimum load profile that maps each element e of the sequence to the minimal load at this point in the sequence based on the activities that are part of $\overrightarrow{S}$. These can be either fully inserted (both the start and end of the activity $\in \overrightarrow{S}$) or partially inserted (only the start or end $\in \overrightarrow{S}$). For partially inserted activities, the position for the element not yet inserted is chosen among the possible insertions in I as the closest one to the inserted element. Note that a violation of the capacity at this point would trigger a failure.

Once the cumulative profile is built, possible insertions for activities that are partially inserted are filtered. The algorithm used consists in iterating over $\overrightarrow{S}$ starting from the inserted element. Possible insertions for the missing element are considered and allowed as long as the load of the activity can be added to the minimal load profile without overloading the capacity. If the capacity is overloaded at some point, the current insertion as well as the insertions not yet reached are removed.

Finally, Algorithm 3 is used to check activities for which neither element is inserted. The loop at line 5 iterates over $\overrightarrow{S}$ starting from the dummy element $\perp$. When a potential start predecessor is encountered, it is added to the `activeStarts` set which maintains potential valid predecessors for the start element that have been encountered so far (line 7). The boolean `canClose` indicates if there exists at least one possible insertion position for the start of the activity that would be valid if the end is inserted at this point. It is set to true whenever a start predecessor is added to `activeStarts`. If adding the activity to the load profile for the current element violates the capacity, `canClose` is set to false and `activeStarts` is emptied as the potential start predecessors will not be matched to a valid insertion for the end element. When a valid predecessor for the end element is encountered, the end predecessor and all the start predecessors in `activeStarts` are validated (lines 13 and 14). The possible pre-

decessors that have not been validated at the end of the loop are removed at line 18.

Algorithm 3: `CumulFiltering`$(Sq, start, end, load, C, profile)$

Input: $start, end, load$: starts, ends and loads of activities, C: capacity,
$Sq = \langle \overrightarrow{S}, I, P, R, E \rangle$: Sequence variable, $profile$: minimum load profile

1 **forall** $i \mid start_i \notin S \wedge end_i \notin S$ **do**
2   $activeStarts \leftarrow \emptyset$;
3   $current \leftarrow \perp$;
4   $canClose \leftarrow false$;
5   **do**
6   **if** `canInsert`$(Sq, start_i, current)$ **then**
7       $activeStarts \leftarrow activeStarts \cup \{current\}$;
8       $canClose \leftarrow true$;
9   **if** $profile(current) + load_i > C$ **then**
10       $activeStarts \leftarrow \emptyset$;
11       $canClose \leftarrow false$;
12   **if** `canInsert`$(Sq, end_i, current) \wedge canClose$ **then**
13       $current$ is valid predecessor for end_i ;
14       $\forall p \in activeStarts$, p is valid predecessor for $start_i$;
15       $activeStarts \leftarrow \emptyset$;
16     $current \leftarrow$`nextMember`$(Sq, current)$
17   **while** $current \neq \perp$;
18   remove predecessors for $start_i$ and end_i that have not been validated ;

The complexity to build the minimum load profile is linear. The complexity to check all the activities $\in A$ is $\mathcal{O}(|A| \cdot |S|)$.

Example 4. Let us consider four activities: $A_0 = [a, e], A_1 = [b, f], A_2 = [c, g]$ and $A_3 = [d, h]$. Each activity A_i has a load of 1. The capacity is $C = 3$. The current partial sequence is $\overrightarrow{S} = (a, b, c, e, f)$. Before propagation, the current possible insertions in I are: $\{(d, \perp), (d, a), (d, b), (d, g), (g, d), (g, e), (g, f), (g, h), (h, a), (h, c), (h, e), (h, d), (h, g)\}$. Note that the possible insertions that are not in the current sequence $((d, g), (g, d), (g, h), (h, d), (h, g))$ will be ignored by the filtering algorithm. Let us propagate the `Cumulative` constraint:

1. The minimal load profile is built based on $A_0 = [a, e]$ and $A_1 = [b, f]$ which are both fully inserted and $A_2 = [c, g]$ which is partially inserted (only c is member in $\overrightarrow{S}$). The possible insertion for the end of A_2 (g) that is the closest to its start (c) is (g, e). Thus, A_2 is considered ending after e to compute the minimum load profile which is $\{\perp : 0, a : 1, b : 2, c : 3, e : 2, f : 0\}$.
2. The possible insertions for the partially inserted activity A_2 are filtered. The sequence is iterated over starting from c. As (g, e) is part of the minimal load profile, it is validated. The remaining possible insertion (g, f) is reached without overloading the capacity and thus validated.
3. The possible insertions for non-inserted activity $A_3 = [d, h]$ are filtered. To do so, Algorithm 3 iterates over the elements in $\overrightarrow{S}$, starting from $\perp$. Both $\perp$ and a are added to `activeStart` and `canClose` is set to $true$. When considering a as possible predecessor for h, as `canClose` is true, the insertions $(h, a), (d, \perp)$ and (d, a) are validated. Afterwards b is added to `activeStart`. When consid-

ering c, adding the activity A_3 at this point would overload the capacity C. Thus, `canClose` is set to $false$ and `activeStart` is emptied. c and d are not validated as possible predecessors, as `canClose` is $false$ when they are considered.

At the end of the propagation, the validated insertions are $(g, e), (g, f), (d, \perp), (d, a)$ and (h, a). The possible insertions $(d, b), (h, c)$ and (h, e) are removed from I.

6 Applications of the Insertion Sequence Variable

This section presents the application of the insertion sequence variable on two variants of the Dial a Ride Problem.

6.1 Dial a Ride Problem

The Dial a Ride Problem (DARP) consists at routing a fleet of vehicles in order to transport clients from one place to another. The variant experimented in this paper was proposed by Cordeau and Laporte [5]. The objective is to minimize the total routing cost of the vehicles (defined as the total distance traveled by the vehicles) under various constraints such a maximum trip duration and time-windows. This problem is modeled with one insertion sequence variable for each route. Each request is modeled by two stops (its pickup and drop) that must be part of the same sequence. A `Cumulative` constraint ensures the capacity of the vehicle is satisfied. The time-window and time constraints are enforced with the help of the `TransitionTimes` constraints.

Search A Large Neighborhood Search (LNS) [25] is used. The relaxation procedure randomly selects a subset of requests that must be reinserted into the sequences. If the search tree is completely explored during a given number of consecutive iterations given by a stagnation threshold s, the relaxation size is increased. Two different search heuristics are considered: 1) **A generic First Fail search**. Similarly as in [14], at each step of the search, it selects the element (the stop) not yet decided with the minimal number of possible insertions in all compatible sequences. Then, it branches in a random order over the possible insertions for the element. 2) **A problem specific heuristic** called `Cost Driven` search. It uses a similar approach to the first fail heuristic to select a stop with a minimal number of possible insertions. The cost metric used in [14] for their `LNS-FFPA` algorithm is used to improve the heuristic. The minimum cost between all possible insertions for a stop is used as a tie breaker for the selection of the next stop to insert. Additionally, the branching decisions, each corresponding to a possible insertion for the stop selected, are explored by increasing order of cost.

6.2 Patient Transportation Problem (PTP)

The Patient Transportation Problem (PTP) [3] is a variation of the classical DARP where clients are patients that must be transported to medical appointments and possibly brought back to a specified location after their care. This implies that some pairs of requests are dependent from each other. The objective consists in maximizing the

number of requests served instead of minimizing the total distance. Additionally, the problem introduces additional constraints such as categories of patients that can only be taken in charge by specific vehicles. The fleet of vehicles is heterogeneous, each has its own capacity, can only serve some types of patients and departure from different points. Also each vehicle is available in a given time window only.

Search Such as for the DARP, LNS is used and Two search heuristics are considered: 1) The same **generic First Fail heuristic** as the one described in 6.1; 2) **A problem specific heuristic** called `Slack Driven` search. It is similar to the `Cost Driven` heuristic described in section 6.1. The cost metric is replaced by a *slack difference* metric which is defined as the total size difference of the time windows of the predecessor and successor of the stop to insert before and after insertion. The intuition is to minimize this difference in order to keep the sequences as flexible as possible and maximizing potential future insertions.

7 Experimental results

This section reports the comparison of the models presented in section 6 with state-of-the-art CP approaches for the DARP and PTP. The models based on insertion sequences variables are referred as the *Insertion Sequence* (`ISEQ`) approaches. The generic *First Fail* heuristic is referred as `FF`. The *Cost Driven* and *Slack Driven* heuristics are referred as `CDS` and `SDS` respectively.

For the DARP, the insertion sequence approach was compared with 1) the *LNS with First Feasible Probabilistic Acceptance*(`LNS-FFPA`) model and heuristic proposed in [14]; 2) an implementation of our model with the sequence variables and interval variables of *CP Optimizer* is referred as `DARP_CPO`. The approaches were run on 68 DARP instances from [6, 4] and are available at [2].

For the PTP, The insertion sequence model was compared with 1) the model proposed in [3], referred as *Scheduling with Maximum Selection Search* (`SCHED+MSS`); 2) the model proposed in [20], referred as *Liu CP Optimizer model* (`LIU_CPO`). A greedy approach referred as (`GREEDY`) was used to compute the initial PTP solutions given to the compared models in the LNS setting. Tests were performed on the benchmark of instances used in [3]. It contains both real exploitation instances and randomly generated instances which are available at [26].

For the `TransitionTimes` constraint, the maximum depth was fixed to 3 and the filtering threshold to 10. The LNS used an initial relaxation of 20% of the requests, a failure limit of 500, a stagnation threshold of 50 and an increase factor of 20%.

Each approach was run 10 times on each instance, with a time limit of 600 sec. The system used for the experiments is a PowerEdge R630 server (128GB, 2 proc. Intel E5264 6c/12t) running on Linux. The approaches using CP Optimizer were implemented using the Java API of CPLEX Optimization Studio V12.8 [19]. The other models were implemented on OscaR [21] running on Scala 2.12.4.

In order to compare the anytime behavior of the approaches, we define the *relative distance* of an approach at a time t as the current distance from the best known objective (BKO) divided by the distance to the worse initial objective (WSO): $(objective(t) -$

$BKO)/(WSO - BKO)$. If an approach has not found an initial solution, the worse initial objective (WSO) is used as objective value. A relative distance of 1 thus indicates that the approach has not found an initial solution or is stuck at the initial solution while a relative distance of 0 indicates that the best known solution has been reached.

Results Figure 3 shows the evolution of the average relative distance during the search. The DARP results are shown on the left. For the PTP, the approaches are compared in two different settings: 1) in the same experimental setting as in [3] (with a LNS search starting from an initial solution given by a greedy approach) (middle); 2) in a DFS starting without an initial solution (right).

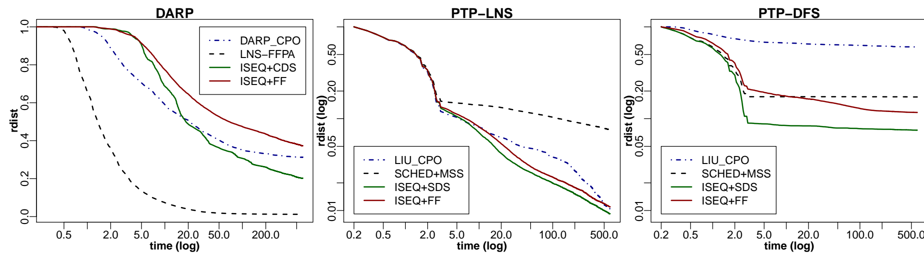


Fig. 3: Average relative distance in function of time

These results suggest that, on the DARP, the sequence based approaches are not able to compete with the dedicated LNS-FFPA algorithm. However, they are able to successfully outperform the dedicated SCHED+MSS approach on the PTP. As can be observed, the approaches using the insertion sequence variable obtain slightly better result than the approach using the state-of-the-art CP Optimizer. Note that the comparison with CP Optimizer is not straightforward as it is mostly black box and its interface does not offer much control over its behavior. However, despite the adaptive LNS search [16] and the advanced techniques (failure directed search [28], objective landscapes [15]) used by CP Optimizer, our approach is competitive in a LNS setting. The experiment in a DFS setting where the advanced search of CP Optimizer is not used suggests that the difference is mainly due to the modeling and propagation as even our generic search outperforms CP Optimizer in this setting.

Constraint parameters Several values were tested for the parameters of the `TransitionTimes` constraint by using the methodology proposed in [27]. It consists in storing the search tree obtained with the weakest filtering and replaying it with the constraints and parameters to test. The impact of the `Cumulative` constraint was also tested by comparing it to a simple checker.

Table 3 presents the results of this experiment on 3 medium sized PTP instances in a DFS setting. The instances are expressed in terms of the number of hospitals (h),

number of available vehicles (v) and number of patients (p). The values are displayed in terms of percentage compared to the base case (the parameter value for the first column). The first row corresponds to the percentage of size (in terms of the number of nodes) of the new search tree compared to the base case. The second row consists in the percentage of time taken to explore the new search tree. For example, on the Hard instance, for a depth d of 2, the search tree is 76.26% smaller which results in an exploration 63.67% faster. Each parameter was tested independently with the others set to their default values.

Table 3: Number of nodes explored (top) and time taken (bottom) with various parameter values

Instance				ρ				d					cache		Cumul.	
Set	h	v	p	0	10	20	∞	1	2	3	6	∞	$\times$	$\surd$	$\times$	$\surd$
Easy	24	9	96	100	100	100	100	100	100	100	100	100	100	100	100	100
				100	96.27	92.27	101	100	102.53	96.68	93.87	91.73	100	95.39	100	126.61
Medium	48	5	96	100	100	100	100	100	0.01	0.01	0.01	0.01	100	100	100	0.01
				100	80.79	55.67	53.07	100	0.05	0.06	0.05	0.06	100	77.24	100	0.01
Hard	96	5	96	100	100	100	55.35	100	76.26	70.31	64.48	59.86	100	100	100	0.01
				100	85.06	56.47	22.74	100	63.67	53.8	38.8	51.93	100	91.44	100	0.03

As can be observed, the constraints have an important impact on both the size of the search tree and the search time for the medium and difficult instances. The easy instance search tree was not affected by the constraints. Note that an increase in depth may result in a faster search despite having the same search tree size (such as for the Easy instance). This is most likely due to the cache that is filled faster in the first calls to Algorithm 1 and thus allows smaller searches in the subsequent calls which results in a gain of time over the whole propagation.

8 Conclusion

In this paper, we propose a new variable called *Insertion Sequence Variable* (ISV) to provide a flexible and efficient model for the DARP and its variant the PTP. The ISV domain extends the set domain variable with the possibility to insert an element after any sequenced element. Experimental results show that the proposed approach is competitive with existing sequence based approaches, outperforms dedicated approaches for the PTP and confirm the effectiveness of the new filtering algorithms proposed.

While used only in the context of the Dial-a-Ride problem in this paper, sequence variables could be used to model a large variety of Routing and Scheduling problems. As future work, it would be interesting to study the usage of the ISV on other problems as well as developing new global constraints and filtering algorithms.

Acknowledgments This research is financed by the Walloon Region (Belgium) as part of PRESupply Project. We thank Siddhartha Jain and Pascal Van Hentenryck for sharing with us their implementation of the LNS-FFPA algorithm.

References

1. Aggoun, A., Beldiceanu, N.: Extending chip in order to solve complex scheduling and placement problems. Mathematical and computer modelling **17**(7), 57–73 (1993)
2. Braekers, K.: Dial-a-Ride Problems Instances. `http://alpha.uhasselt.be/kris.braekers/` (2019), [Online; accessed 2-December-2019]
3. Cappart, Q., Thomas, C., Schaus, P., Rousseau, L.M.: A constraint programming approach for solving patient transportation problems. In: International Conference on Principles and Practice of Constraint Programming. pp. 490–506. Springer (2018)
4. Cordeau, J.F.: A branch-and-cut algorithm for the dial-a-ride problem. Operations Research **54**(3), 573–586 (2006)
5. Cordeau, J., Laporte, G.: The dial-a-ride problem (DARP): variants, modeling issues and algorithms. 4OR **1**(2), 89–101 (2003). https://doi.org/10.1007/s10288-002-0009-8, `https://doi.org/10.1007/s10288-002-0009-8`
6. Cordeau, J.F., Laporte, G.: A tabu search heuristic for the static multi-vehicle dial-a-ride problem. Transportation Research Part B: Methodological **37**(6), 579–594 (2003)
7. Cordeau, J.F., Laporte, G.: The dial-a-ride problem: models and algorithms. Annals of Operations Research **153**(1), 29–46 (2007)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
9. Gervet, C.: Interval propagation to reason about sets: Definition and implementation of a practical language. Constraints **1**(3), 191–244 (1997)
10. Hartert, R., Schaus, P., Vissicchio, S., Bonaventure, O.: Solving segment routing problems with hybrid constraint programming techniques. In: International Conference on Principles and Practice of Constraint Programming. pp. 592–608. Springer (2015)
11. Ho, S.C., Szeto, W., Kuo, Y.H., Leung, J.M., Petering, M., Tou, T.W.: A survey of dial-a-ride problems: Literature review and recent developments. Transportation Research Part B: Methodological **111**, 395–421 (2018)
12. IBM Knowledge Center: Interval variable sequencing in CP Optimizer. `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.9.0/ilog.odms.ide.help/refcppopl/html/interval_sequence.html` (2019), [Online; accessed 22-November-2019]
13. IBM Knowledge Center: Search API for scheduling in CP Optimizer. `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.9.0/ilog.odms.cpo.help/refcppcpoptimizer/html/sched_search_api.html?view=kc#85` (2019), [Online; accessed 22-November-2019]
14. Jain, S., Van Hentenryck, P.: Large neighborhood search for dial-a-ride problems. In: International Conference on Principles and Practice of Constraint Programming. pp. 400–413. Springer (2011)
15. Laborie, P.: Objective landscapes for constraint programming. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 387–402. Springer (2018)
16. Laborie, P., Godard, D.: Self-adapting large neighborhood search: Application to single-mode scheduling problems. Proceedings MISTA-07, Paris **8** (2007)
17. Laborie, P., Rogerie, J.: Reasoning with conditional time-intervals. In: FLAIRS conference. pp. 555–560 (2008)
18. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Reasoning with conditional time-intervals. part ii: An algebraical model for resources. In: Twenty-Second International FLAIRS Conference (2009)

19. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Ibm ilog cp optimizer for scheduling. Constraints **23**(2), 210–250 (2018)
20. Liu, C., Aleman, D.M., Beck, J.C.: Modelling and solving the senior transportation problem. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 412–428. Springer (2018)
21. OscaR Team: OscaR: Scala in OR (2012), available from `https://bitbucket.org/oscarlib/oscar`
22. Perron, L., Furnon, V.: Or-tools (2019), `https://developers.google.com/optimization/`
23. Perron, L., Furnon, V.: OR-Tools Sequence Var. `https://developers.google.com/optimization/reference/constraint_solver/constraint_solver/SequenceVar` (2019), [Online; accessed 22-November-2019]
24. de Saint-Marcq, V.l.C., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS). pp. 1–10 (2013)
25. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: International Conference on Principles and Practice of Constraint Programming. pp. 417–431. Springer (1998)
26. Thomas, C., Cappart, Q., Schaus, P., Rousseau, L.M.: CSPLib problem 082: Patient transportation problem. `http://www.csplib.org/Problems/prob082` (2018)
27. Van Cauwelaert, S., Lombardi, M., Schaus, P.: How efficient is a global constraint in practice? Constraints **23**(1), 87–122 (2018)
28. Vilím, P., Laborie, P., Shaw, P.: Failure-directed search for constraint-based scheduling. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 437–453. Springer (2015)