

Incremental Sequence Variables: Application to the Patient Transportation Problem

No Author Given

No Institute Given

Abstract. The Patient Transportation Problem (PTP) consists in transporting as many patients as possible to medical appointments and bringing them back after their care by routing a fleet of heterogeneous vehicles. This problem is at the frontier between routing and scheduling. The optional selection of the visits makes it particularly difficult to model and solve with a standard CP successor model. The optional interval variables and sequence variables available in CP Optimizer combined with the powerful failure directed search appears to be the state-of-the-art approach for solving this problem. Unfortunately, this technology is not available in other solvers and the sequence variable implementation was not fully described in the literature. We introduce a possible representation for the domain of sequence variable called *Incremental Sequence Variable* (ISV) that naturally extends the standard subset bound domain for set variables with one additional append modifier that grows a sequence prefix at the end. We describe the few constraints and the filtering algorithms needed on those variables to model the PTP. Our experimental results on a large variety of benchmarks show that the proposed approach is competitive with the one using sequence variables in the well engineered CP Optimizer.

1 Introduction

The Patient Transportation Problem (PTP) consists in transporting as many patients as possible to medical appointments and bringing them back after their care by routing a fleet of heterogeneous vehicles. It is a door-to-door transportation service to people that do not have the ability to go to hospital by themselves. In many communities, the patient transportation services increase (due to the aging population) with the exigence of service quality.

The problem is more formally described on a directed complete graph $G = (V, A)$ with vertex set $V = S \cup D \cup R \cup D_p$ where S represents the start locations of patients (homes), D the destination locations of patients (hospitals), R the return locations of the patients and D_p the depots of the vehicles. A matrix $trans$ indicates for each arc $(i, j) \in A$ the non negative travel time ($trans_{i,j}$) from location i to j , assumed to satisfy triangle inequality. Days before executing the planning, each patient submits a request. Let \mathcal{R} be the set of all requests.

With each request $r \in \mathcal{R}$ is associated the service time required to embark/disembark the vehicle srv_r , the number of places required in the vehicle ld_r , the category of patient cat_r (with or without wheelchair, etc.), the time of rendez-vous rdv_r and the duration of the care $drdv_r$. A request r consists in either a two-way trip (forward: from a starting place $orig_r \in S$ to a destination place $dest_r \in D$ and backward: from $dest_r$ to a return place $ret_r \in R$) or a one-way trip (either forward or backward). Let \mathcal{V} be the set of heterogeneous vehicles. Each vehicle $v \in \mathcal{V}$ has a capacity k_v , a set C_v of request categories it can serve, a start depot $start_v \in D_p$, an end depot $end_v \in D_p$ and a time window $[savail_v, eavail_v]$ in which the route must be done. The maximum waiting time Wt indicates the duration of the time windows during which the patient has to be transported. The PTP consists in constructing a route for each vehicle such that the number of requests satisfied is maximized and:

1. for each accepted request r the forward trip is executed in the interval $[rdv_r - Wt, rdv_r]$ and the backward trip in $[rdv_r + drdv_r, rdv_r + drdv_r + Wt]$;
2. for each accepted request r , the service time for embarking and disembarking the vehicle is respected;
3. for each route, the transition times between locations are respected;
4. the category of the vehicle needed by the request is respected;
5. the capacity of the vehicle is not exceeding at any time.

Related Work The PTP is related to the Dial a Ride Problem [1, 2], a class of Vehicle Routing Problem with Time Windows [3, 4]. Two recent approaches for solving the PTP are [5] and [6]. The approach proposed in [5] consists in representing the problem with a scheduling model. Trips are represented by activities. Decision variables consist in accepting a request and for each activity, selecting a vehicle and choosing the start and end times of the activities.

The approach of [6] is based on IBM ILOG CP Optimizer solver (a commercial solver [7]). It also uses a scheduling approach but making use of the sequence variables from CP Optimizer [8, 9] to decide the order of visits in each vehicle.

Initial comparisons between the two approaches indicate that the one of [6] with sequence variables is more efficient. The reason is that the pure scheduling model from [5] does not exhibit sequencing variables. Therefore this model branches on the start time of the activities that could be deduced from the sequencing of the visits instead. Another shortcoming of the approach of [5] is that the optional aspect of activities is not explicitly dealt with in the model but managed through a custom search.

The high level functionality and constraints related to sequence variables of CP Optimizer have been briefly described in [9]. Unfortunately, no details are given on the implementation of such variables and the filtering algorithms of the constraints in the literature. Furthermore, the sequence variables used are oriented towards scheduling as those are composed of optional interval variables.

Our contributions We describe an implementation of sequence variables named *Incremental Sequence Variable* (ISV) based on the sparse-set data structure.

Our domain representation extends the subset bound domain [10] for set variables plus an additional growing prefix sequence on which elements can only be appended. We describe the constraints involving the sequence variables necessary to model the PTP. We experimentally test the new approach, show that it outperforms the scheduling approach [5] and is competitive with the model described in [6].

The rest of the paper is organized as follows. In Section 2 notions used in the paper are presented. Section 3 is devoted to the description of the Incremental Sequence Variable. Constraints on the sequence variable used for the PTP are presented in Section 4. Section 5, is devoted to the description of the PTP model based on the proposed tools. Section 6 reports experimental results on real and generated instances for PTP while Section 7 concludes the paper.

2 Preliminaries

By definition, a Constraint Satisfaction Problem (CSP) is a triplet (V, D, C) consisting of: a set of variables V , each associated to a domain in D and a set of constraints C restricting the domains of the variables.

Let $X = \{0, \dots, n\}$ be a finite set and $\mathcal{P}(X)$ the set of subsets (power set) of X . It is known that inclusion \subseteq is a partial order over $\mathcal{P}(X)$ and the structure $(\mathcal{P}(X), \subseteq)$ is a lattice generally used to represent 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 [10]).

The domain of a set variable S on X is a tuple $\langle P, R, E \rangle$ where $P \subseteq X$ is the set of possible elements of S , $R \subseteq X$ is the set of required elements of S and $E \subseteq X$ is the set of excluded elements of S . The domain is defined as $\langle P, R, E \rangle \equiv \{S' \mid S' \subseteq X \wedge R \subseteq S' \subseteq R \cup P\}$. 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.

Notation	Operation	Complexity
<code>requires(S, e)</code>	move e to R , fails if e is in E	$\Theta(1)$
<code>excludes(S, e)</code>	move e to E , fails if e is in R	$\Theta(1)$
<code>isBound(S)</code>	return true iff S is bound	$\Theta(1)$
<code>isPossible(S, e)</code>	return true iff $e \in P$	$\Theta(1)$
<code>isRequired(S, e)</code>	return true iff $e \in R$	$\Theta(1)$
<code>isExcluded(S, e)</code>	return true iff $e \in E$	$\Theta(1)$
<code>allPossible(S)</code>	enumerate P	$\Theta(P)$
<code>allRequired(S)</code>	enumerate R	$\Theta(R)$
<code>allExcluded(S)</code>	enumerate E	$\Theta(E)$

Table 1: Operations supported by set variables

It is denoted by \vec{A} a *sequence* of X where $A \subseteq X$. The set of all sequences of X is denoted by $\vec{\mathcal{P}}(X)$. Let \vec{A} and \vec{B} be two disjoint sequences of X . A

concatenation of \vec{A} and \vec{B} is noted $\vec{A} + \vec{B}$. For example, if $\vec{A} = (1, 3, 5)$ and $\vec{B} = (2, 7)$, then $\vec{A} + \vec{B} = (1, 3, 5, 2, 7)$. Let \vec{A} be a sequence, for two elements $a, b \in A$, the relationship "a precedes b in \vec{A} " is expressed by the notation $a \prec_{\vec{A}} b$.

3 Incremental Sequence Variable

The relation " \vec{A} is the prefix of \vec{B} " between two sequences $\vec{A}, \vec{B} \in \vec{\mathcal{P}}(X)$ is denoted by the operator \sqsubseteq and defined as $\vec{A} \sqsubseteq \vec{B} \iff A \subseteq B \wedge \vec{B} = \vec{A} + B \setminus A$. For example, $\vec{A} = (1, 3, 7)$ is the prefix of $\vec{B} = (1, 3, 7, 9, 4)$: $\vec{A} \sqsubseteq \vec{B}$. The prefix relation is a partial order over $\vec{\mathcal{P}}(X)$ and the structure $(\vec{\mathcal{P}}(X), \sqsubseteq)$ is not a lattice.

Definition 1. An incremental sequence variable Sq on a set X is a variable whose domain is represented by a tuple $\langle \vec{A}, P, R, E \rangle$ where $\langle P, R, E \rangle$ is the domain of a set variable on X and $\vec{A} \in \vec{\mathcal{P}}(R)$. Sq is bound if P is empty and $|A| = |R|$. The domain of Sq is defined as

$$\langle \vec{A}, P, R, E \rangle \equiv \{ \vec{S} \in \vec{\mathcal{P}}(P \cup R) \mid R \subseteq S \wedge \vec{A} \sqsubseteq \vec{S} \}$$

For example, let us consider $X = \{0, 1, 2\}$, the variable Sq of domain $\langle (1), \{0\}, \{1, 2\}, \emptyset \rangle$ corresponds to the sequences $\{(1, 2), (1, 0, 2), (1, 2, 0)\}$. The sequences (1) and (1, 0) are not valid as they do not contain the required element 2.

The notation $e \in Sq$ where Sq is a sequence variable of domain $\langle \vec{A}, P, R, E \rangle$, indicates that the element e is required in the sequence ($e \in R$). The incremental sequence variable inherits all the operations defined on the set variable and supports the additional operations summarized in Table 2.

Notation	Operation	Complexity
<code>isBound(Sq)</code>	return true iff P is empty and $ A = R $	$\Theta(1)$
<code>appends(Sq, e)</code>	append e to the end of \vec{A} , moves e to R if needed, fails if e is in E	$\Theta(1)$
<code>lastAppended(Sq)</code>	return the last element of \vec{A}	$\Theta(1)$
<code>isMember(Sq, e)</code>	return true iff e is present in \vec{A}	$\Theta(1)$
<code>allMember(Sq)</code>	enumerate \vec{A}	$\Theta(A)$
<code>isAppendable(Sq, e)</code>	return true iff $e \in (R \setminus A) \cup P$	$\Theta(1)$
<code>allAppendable(Sq)</code>	enumerate $(R \setminus A) \cup P$	$\Theta(R \setminus A + P)$

Table 2: Operations supported by incremental sequence variables

3.1 Implementation

The implementation proposed for the Incremental Sequence Variable uses array-based sparse sets as in [11]. The internal set variable $\langle P, R, E \rangle$ is implemented with 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 . Similarly, \vec{A} is represented by an array of length $|X|$ called `members` and a reversible integer m . The a first elements in `members` correspond to the sequence \vec{A} . Two additional arrays called `elemPos` and `memberPos` map each element of X with its positions in `elems` and `members`, allowing access in $\Theta(1)$.

Access and identity operations are implemented by comparing the position of an element with the values of r , p and m . Modification operations (`append`, `requires` and `excludes`) are implemented by swapping the element whose status is modified with the element after m or r or before p and incrementing the corresponding reversible integer. Backtracks are done by resetting the reversible integers to their previous value. Enumeration operations are done by iterating over the subpart of the array corresponding to the desired set or sequence. Additional methods allow specifying under which case a propagator should be notified of changes in the domain. Possible events triggering such notifications are: appending, requiring or excluding an element and when the sequence variable is bound. An illustration of the sparse set representation for the variable Sq with a domain of $\langle (f, b), \{c\}, \{b, e, f\}, \{a, d\} \rangle$ is given in Figure 1.

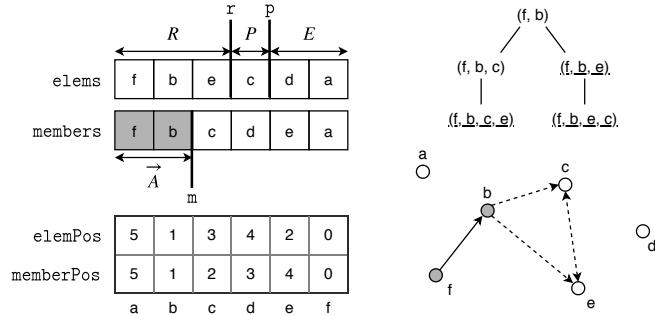


Fig. 1: The incremental sequence variable domain $\langle (f, b), \{c\}, \{b, e, f\}, \{a, d\} \rangle$ represented using sparse sets (left) and the corresponding tree and paths (right). Valid sequences are underlined in the tree.

4 Constraints on Incremental Sequence Variables

We introduce the constraints needed to solve the PTP with sequence variables.

4.1 First and Last constraints

The constraint $\text{First}(Sq, f)$, holds if the element f is the first in the sequence.

$$\text{First}(Sq, f) = \left\{ \vec{S} \mid \vec{S} \in \langle \vec{A}, P, R, E \rangle \wedge \vec{S} = (f) + \vec{S}' \right\}$$

The constraint $\text{Last}(Sq, l)$ holds if the element l is the last in the sequence:

$$\text{Last}(Sq, l) = \left\{ \vec{S} \mid \vec{S} \in \langle \vec{A}, P, R, E \rangle \wedge \vec{S} = \vec{S}' + (l) \right\}$$

Propagation The propagation for **First** is called only once when the constraint is posted. It appends f as the first element if \vec{A} is empty. Otherwise, it checks the first element of \vec{A} and fails if it is different from f . When **Last** is posted, l is made required. The propagation is called each time an element is appended or required. It ensures that l cannot be appended while at least one other element is present in R . When l is appended, all the remaining elements in P are excluded.

4.2 Dependency constraint

The constraint $\text{Dependency}(Sq, U)$ ensures that all the elements of U are in the sequence or none of them:

$$\text{Dependency}(Sq, U) = \left\{ \vec{S} \mid \vec{S} \in \langle \vec{A}, P, R, E \rangle \wedge (U \subseteq S \vee U \cap S = \emptyset) \right\}.$$

Propagation The propagation is called when an element from U is appended, required or excluded. As soon as one of the elements of U is appended or required, all the others are required. Conversely, if an element of U is excluded, all the others are excluded from the sequence.

4.3 Precedence constraint

This constraint enforces precedences between the elements of a sequence \vec{O} in the sequence variable:

$$\text{Precedence}(Sq, \vec{O}) \equiv \forall a, b \in S \cap O, a \underset{\vec{O}}{\prec} b \implies a \underset{\vec{S}}{\prec} b \text{ where } \vec{S} \in \langle \vec{A}, P, R, E \rangle$$

Propagation The propagation is triggered whenever an element e from O is appended in the variable Sq . All the elements before e in \vec{O} are excluded in the variable Sq .

Example 1. Let us consider the sequence variable Sq with domain $\langle (0, 2), \{1, 3, 4\}, \{0, 2\}, \emptyset \rangle$. The constraint $\text{Precedence}(Sq, \vec{O})$ where $\vec{O} = (0, 1, 2, 3, 4)$ reduces this domain to $\langle (0, 2), \{3, 4\}, \{0, 2\}, \{1\} \rangle$.

4.4 Sequence Allocation constraint

Given a list of m sequences $(Sq_v)_{v=0, \dots, m-1}$ of domain $\langle \vec{A}_v, P_v, R_v, E_v \rangle$ over a set of elements X . Each element i of X is associated with an integer domain variable $x_i \in D_i$ that indicates the sequence where it appears. Note that D_i may contain values not corresponding to any sequence. The constraint $\text{SequenceAllocation}((Sq_v), (x_i), X)$ ensures that each element i of X is a member of the sequence indicated by its associated variable x_i .

$$\text{SequenceAllocation}((Sq_v), (x_i), X) \equiv \forall i \in X, \forall v \in \{0..m-1\}, i \in Sq_v \iff x_i = v$$

Propagation The propagation is called when an element is appended, required or excluded in one of the sequence variables Sq or when the domain of a variable x_i changes. If an element i is required or appended in one of the sequences, its x_i variable is fixed to the value of the sequence and the element is excluded from the other sequences. If an element i is excluded from a sequence, the corresponding value is removed from its x_i variable. If a value corresponding to a sequence variable is removed from the domain of a x_i variable, the element i is excluded from the sequence variable. If the variable x_i is fixed to a singleton corresponding to a sequence variable, the element is required in the corresponding sequence.

4.5 Transition Times constraint

In a scheduling context such as in the PTP, the elements to sequence correspond to visits made by a vehicle. Therefore each element of a sequence is also associated with time window variables. The **TransitionTimes** 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 $trans_{i,j}$ specifies transition times associated to each couple of activities (i, j) . The **TransitionTimes** constraint is then defined as

$$\text{TransitionTimes}(Sq, start_i, dur_i, trans_{i,j}) \equiv \\ \forall a, b \in S, a \underset{\vec{S}}{\preceq} b \implies start_b \geq start_a + dur_a + trans_{a,b} \text{ where } \vec{S} \in \langle \vec{A}, P, R, E \rangle.$$

Propagation The propagation is called whenever an element is appended or required or if one of the bounds of a time window changes. The filtering algorithm is split into three parts: *time windows update*, *feasible path checking and filtering* and *append deduction*.

Time window update This filtering algorithm is used to fix the start and duration of the activities already appended in the sequence based on their order in \vec{A} . It is also used to update the time windows of appendable activities (i.e. in the set $(R \setminus A) \cup P$) based on the last appended activity of the sequence. This update is done in linear time.

Feasible Path checking and filtering This algorithm is used to check that there exists at least one feasible extension of the current sequence composed of the required activities not yet appended (i.e. in the set $R \setminus A$) that satisfies the transition time constraints. The function $\text{feasiblePath}(\ell = \text{lastAppended}(Sq), \Omega = R \setminus A, t = start_\ell + dur_\ell, d)$ checks that there exists at least one feasible sequencing of the activities in Ω satisfying the transition time constraints after having completed the activity ℓ at a time t . This problem is NP-Complete [12]. An exact algorithm consists in performing a recursive depth first backtracking search (DFS) enumerating all possible sequences until a feasible one is eventually found.

Algorithm 1 is based on this approach. A pruning is done in the loop at line 6 if one realizes that no activity can be appended. This loop checks that all the activities can possibly be appended as the one following the activity ℓ . By the triangle inequality assumption of the transition times, if at least one activity of Ω cannot be appended directly after ℓ , then it can surely not be appended later in time if some activities were inserted in between. Therefore **false** is returned in such case which corresponds to the infeasibility pruning. Otherwise every possible extension is considered recursively at line 13 and if one possible extension leads to a positive outcome, **true** is returned. Since this algorithm is exponential in $|\Omega|$ the depth of the tree is limited to d . If this limit is reached, the path is assumed to be feasible by returning **true** at line 11.

Algorithm 1: feasiblePath(ℓ, Ω, t, d)

Input: ℓ : last activity visited, Ω : set of activities to reach, t : departure time from ℓ , d : depth and **cache**: memoization map

```

1 if  $\Omega = \emptyset$  then
2   | return true;
3   |  $(t_f, t_i) \leftarrow \text{cache.getOrElse}((\ell, \Omega), (-\infty, +\infty))$  ;
4   if  $t \leq t_f$  then return true;
5   if  $t \geq t_i$  then return false;
6   for  $a \in \Omega$  do
7     | if  $\max(\text{start}_a) < \max(t + \text{trans}_{\ell,a}, \min(\text{start}_a))$  then
8       |   |  $\text{cache.update}((\ell, \Omega), (t_f, \min(t_i, t)))$  ;
9       |   | return false ; // pruning since infeasible sequence
10  if  $d \leq 0$  then
11    | return true ; // pruning since maximum depth reached
12  else
13    | for  $a \in \Omega$  sorted in increasing ( $\min(\text{start}_a) + \min(\text{dur}_a)$ ) do
14      |   | if feasiblePath( $a, \Omega \setminus \{a\}, \max(t + \text{trans}_{\ell,a}, \min(\text{start}_a)), d - 1$ ) then
15        |   |   |  $\text{cache.update}((\ell, \Omega), (\max(t_f, t), t_i))$  ;
16        |   | return true;
17  return false;
```

The time complexity of Algorithm 1 is $\mathcal{O}(|\Omega|^d)$ in worse case as it corresponds to a depth-first search of depth d with a branching factor $|\Omega|$. In order to reduce the time complexity of the successive calls to **feasiblePath**, memoization is used to avoid exploring several times a partial extension that can be proven infeasible or feasible based on previous executions. This optimization is implemented with gray lines. A global map called **cache** is assumed to contain keys composed of the arguments of the function, that is a pair with (Ω, ℓ) . 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 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 3 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 8 and 15 depending on the result found.

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.

Algorithm 2: TransitionTimesFiltering($Sq : \langle \vec{A}, P, R, E \rangle, d$)

```

1  $\ell \leftarrow \text{lastAppended}(Sq) ;$ 
2  $cache \leftarrow \text{map}() ; // \text{ initializing the memoization map}$ 
3 if  $\text{!feasiblePath}(\ell, R \setminus A, \text{start}_\ell + \text{dur}_\ell, d)$  then
4   return failure ;
5 forall  $a \in P$  do
6   if  $\text{!feasiblePath}(\ell, (R \setminus A) \cup \{a\}, \min(\text{start}_\ell) + \min(\text{dur}_\ell), d)$  then
7     excludes( $Sq, a$ );

```

This TransitionTimesFiltering algorithm executes in $\mathcal{O}(|P| \cdot |R \setminus A|^d)$. Notice that the cache is shared and reused along the calls in order to avoid many subtree explorations.

Append deduction The third propagation algorithm detects if a required activity must be appended next and adds it to the sequence. For each required activity i , the algorithm checks that there exists at least one appendable activity e (i.e. $e \in (R \setminus A) \cup P$) that can be visited before while keeping the required activity feasible afterwards. If it is not the case then the required activity is added to the sequence. The pseudocode is given in Algorithm 3. Its time complexity is $\mathcal{O}(|R \setminus A| \cdot |(R \setminus A) \cup P|)$.

Algorithm 3: AppendDeduction($Sq : \langle \vec{A}, P, R, E \rangle$)

```

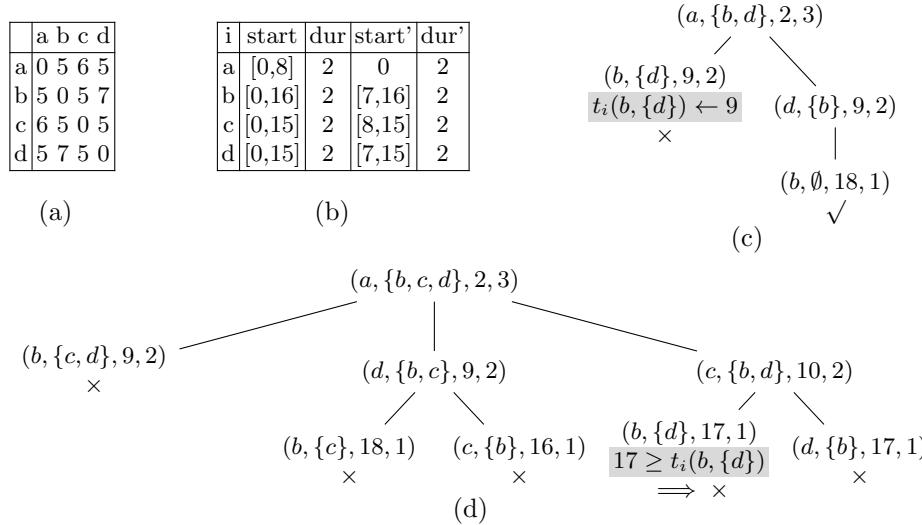
1  $\ell \leftarrow \text{lastAppended}(Sq) ;$ 
2  $t \leftarrow \min(\text{start}_\ell) + \min(\text{dur}_\ell) ;$ 
3 forall  $i \in R \setminus A$  do
4    $\text{feasible} \leftarrow \text{false} ;$ 
5   forall  $e \in \text{allAppendable}(Sq) \setminus \{i\}$  do
6      $\text{depe} \leftarrow \max(t + \text{trans}_{\ell,e}, \min(\text{start}_e)) + \min(\text{dur}_e) ;$ 
7      $\text{feasible} \leftarrow \max(\text{start}_r) \geq \text{depe} + \text{trans}_{e,i} ;$ 
8     if  $\text{feasible}$  then break;
9   if  $\text{!feasible}$  then
10    append( $Sq, i$ ) ;
11    break ;

```

Example 2. Consider the following example where $X = \{a, b, c, d\}$ is the set of activities. The transition times between activities are given in the table of Fig 2 (a) and the initial time windows (start, duration) in the table of Fig 2 (b). We consider the sequence variable Sq of domain $\langle (a), \{c\}, \{a, b, d\}, \emptyset \rangle$. 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 new time bounds are reported in columns 'start' and 'dur' of Fig 2 (b).

2. *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 \checkmark . The initial value of the parameter d is 3. The usage of memoization between both searches is highlighted. The domain is updated to $\langle(a), \emptyset, \{a, b, d\}, \{c\}\rangle$ as the filter detects that c must be excluded.
3. *Append deduction* is applied. The domain after propagation is $\langle(a, d), \emptyset, \{a, b, d\}, \{c\}\rangle$ as sequencing b before d is not feasible.



- Each request $r \in \mathcal{R}$ is associated with a boolean variable Rs_r corresponding to 1 if the request r is selected (realized in the solution) and 0 otherwise.
- Each stop $i \in X$ is associated with an activity defined by a triplet of variables (s_i, d_i, e_i) where s_i is the starting time of the activity, d_i is the duration of the activity e_i is the ending time of the activity. The relation $e_i = s_i + d_i$ must hold. The initial domain of these variables is defined by the corresponding request $r \in \mathcal{R}$ and whether the stop is part of a forward or a backward trip: for a forward trip, $s_i \in [rdv_r - Wt, rdv_r - srv_r]$ and $e_i \in [rdv_r - Wt + srv_r, rdv_r]$; for a backward trip, $s_i \in [rdv_r + drdv_r, rdv_r + drdv_v + Wt - srv_r]$ and $e_i \in [rdv_r + drdv_r + srv_r, rdv_r + Wt]$. in both cases, $d_i = srv_r$. A variable x_i indicates which vehicle visits the stop. Its initial domain contains all the compatible vehicle indices ($cat_r \in C_v$) along with an additional value \perp which indicates that the stop is not visited by any vehicle. Variables for the depot stops of a vehicle v are initialized with a vehicle variable $x_i = v$ and time windows corresponding to the availability of the vehicle: $s_i \in [savail_v, eavail_v]$, $e_i \in [savail_v, eavail_v]$, $d_i = e_i - s_i$.
- Alternative stop activities are duplicated for each vehicle and defined by the variables $(salt_{i,v}, dalt_{i,v}, ealt_{i,v})_{i \in X, v \in \mathcal{V}}$. They are linked with the original stop activities by an **element** constraint [13]. These auxiliary variables are used for the **TransitionTimes** constraint over the sequence variable corresponding to their vehicle.
- For each travel $t = (p, d) \in \mathcal{T}$, an activity is defined by the variables (st_t, dt_t, et_t) with $st_t = s_p$, $et_t = e_d$ and $dt_t = et_t - st_t$. The vehicle variable x_t of the travel t is the vehicle variable of its pickup stop $x_t = x_p$. The variable $lt_t = ld_r$ indicates the load of the associated request r . These variables are used for the cumulative global constraint.

5.2 Constraints

$$\max\left(\sum_{r \in \mathcal{R}} Rs_r\right) \quad (1)$$

s.t.

$$Rs_r \equiv (x_i \neq \perp) \quad \forall r \in \mathcal{R}, i \in X \mid i \in r \quad (2)$$

$$\text{element}(x_i, (salt_{i,v}), s_i) \quad \forall v \in \mathcal{V} \quad (3)$$

$$\text{element}(x_i, (dalt_{i,v}), d_i) \quad \forall v \in \mathcal{V} \quad (4)$$

$$\text{element}(x_i, (ealt_{i,v}), e_i) \quad \forall v \in \mathcal{V} \quad (5)$$

$$\text{First}(Sq_v, sd_v) \quad \forall v \in \mathcal{V} \quad (6)$$

$$\text{Last}(Sq_v, ed_v) \quad \forall v \in \mathcal{V} \quad (7)$$

$$\text{Precedence}(Sq_v, (p, d)) \quad \forall v \in \mathcal{V}, t = (p, d) \in \mathcal{T} \quad (8)$$

$$\text{Dependency}(Sq_v, (p, d)) \quad \forall v \in \mathcal{V}, t = (p, d) \in \mathcal{T} \quad (9)$$

$$\text{SequenceAllocation}((Sq_v), (x_i), X) \quad \forall v \in \mathcal{V} \quad (10)$$

$$\text{TransitionTimes}(Sq_v, (salt_{i,v}), (dalt_{i,v}), trans) \quad \forall v \in \mathcal{V} \quad (11)$$

$$s_d \geq e_p + trans_{p,d} \quad \forall t = (p, d) \in \mathcal{T}$$

The objective function (1) maximizes the number of served requests. Constraint (2) links the request selection variables with the vehicle variables of the request travel(s). The group of `element` constraints (3) links each stop activity with the alternative activity corresponding to the vehicle performing the activity.

Constraint (4) ensures that each vehicle leaves from its starting depot while Constraint (5) ensures that each vehicle will end at its end depot. Constraint (6) ensures that each pickup stop is visited before its corresponding drop stop. Constraint (7) ensures that each couple of pickup and drop stops is present in a same vehicle. Constraint (8) enforces that the vehicle variable of each stop corresponds to the sequence containing the stop or the value \perp if the stop is not visited. Constraint (9) ensures the consistency of the time windows regarding the transition times between stops.

Constraint (10) ensures that the number of places occupied by patients in a same vehicle v cannot exceed its capacity k_v . This constraint is referred in the literature as the cumulative resource global constraint [14]. We use the filtering algorithm of Gay et al. [15]. Constraint (11) is an additional constraint that improves the propagation by ensuring that the transition time is taken into account in the time window of the stops of a same travel.

5.3 Search

An Adaptive Large Neighborhood Search as described in [16, 17] is used. One relaxation method (*random request*) and two search heuristics (*sequence selection* and *visit selection*) are considered along with different parameter values for the size of the relaxation and the backtrack limit of the search.

Random request relaxation Inspired by the partial order schedule relaxation [18], a random number of requests are relaxed from the current best solution. Those can be reinserted anywhere in any sequence while the others are forced to be reassigned in the same sequence in the same linear order. The relaxation procedure is given in Algorithm 4.

Algorithm 4: Relaxation($sol, toRelax$)

Input: sol : current solution, $toRelax$: set of requests to relax

- 1 $stopsToFix \leftarrow \{i \mid i \in X \wedge i \text{ is part of a request not in } toRelax\}$;
- 2 **forall** $i \in stopsToFix$ **do**
- 3 | $x_i \leftarrow sol.x_i$;
- 4 **forall** $v \in \mathcal{V}$ **do**
- 5 | $seqStops \leftarrow \{i \mid i \in stopsToFix \text{ and } sol.x_i = v\}$;
- 6 | $seqStopsSorted \leftarrow seqStops$ sorted by $sol.s_i$;
- 7 | $\text{post}(\text{Precedence}(Sq_v, seqStopsSorted))$;

Sequence selection search This heuristic given in Algorithm 5 selects the sequence that finishes the earliest (according to the earliest end time of its last appended activity) and consider all its possible extensions. Branches are explored from left

to right according to the latest end time of appendable stops. The travel distance between the last appended stop and the new stop is used as a tie breaker.

Algorithm 5: SequenceSelection

```

1 extendableSeqs  $\leftarrow \{v \mid v \in \mathcal{V} \wedge \text{isBound}(Sq_v)\}$  ;
2 if extendableSeqs =  $\emptyset$  then
3   | Solution found ;
4 else
5   |  $v_{\min} \leftarrow \operatorname{argmin}_{v \mid v \in \text{extendableSeqs}} \min(e_{\text{lastAppended}(Sq_v)})$ ;
6   |  $\ell \leftarrow \text{lastAppended}(Sq_{v_{\min}})$  ;
7   | forall  $i \in \text{allAppendable}(Sq_{v_{\min}})$  by order of  $(\max(e_i), \text{trans}_{\ell, i})$  do
8   |   | branch(append( $Sq_{v_{\min}}$ ,  $i$ )) ;

```

Stop selection search This heuristic given in Algorithm 6 selects the stop which has the earliest time window end and generates a new branch for each possible sequence where it can be appended. If possible, an additional branch is generated where the stop is excluded from each sequence by setting its vehicle variable to the value \perp . Branches are explored by order of travel distance between the last appended stop of the sequence considered and the new stop. Note that this search heuristic is not complete since the alternatives where the selected stop is postponed by at least one position in each sequence are not considered.

Algorithm 6: StopSelection

```

1 appendableStops  $\leftarrow \{i \mid i \in X \wedge i \text{ is appendable in at least one sequence}\}$  ;
2 if appendableStops =  $\emptyset$  then
3   | Solution found ;
4 else
5   |  $i_{\min} \leftarrow \operatorname{argmin}_{i \mid i \in \text{appendableStops}} \max(e_i)$  ;
6   | compatibleSeqs  $\leftarrow \{v \mid v \in \mathcal{V} \wedge \text{isAppendable}(Sq_v, i_{\min})\}$  ;
7   | forall  $v \in \text{compatibleSeqs}$  by order of  $\text{trans}_{\text{lastAppended}(Sq_v), i_{\min}}$  do
8   |   | branch(append( $Sq_v$ ,  $i_{\min}$ )) ;
9   |   | if ( $\perp \in x_{i_{\min}}$ ) then
10  |   |   | branch( $x_{i_{\min}} \leftarrow \perp$ ) ;

```

6 Experimental results

This section reports the comparison of the model presented in section 5 with state-of-the-art CP approaches for the PTP. The model based on incremental sequences variables described in Section 5 is referred as the *Sequence Based (SEQ+ALNS)* approach. The maximum depth of the constraint *TransitionTimes* was fixed at $d = 3$. Relaxation sizes of 10, 20, 40 and failure limits of 200, 500, 750 were used as parameters for the relaxation and the search in the ALNS. The model proposed in [5] is referred as *Scheduling with Maximum Selection Search (SCHED+MSS)* and the model proposed in [6] is referred as *Liu CP Optimizer model (LIU_CPO)*. A greedy approach referred as (GREEDY) was used to compute the initial solutions given to the compared models. Each approach was run 5

times on each instance, starting from the same solution with a time limit of 300 sec. The best solution out of the 5 runs is reported. The system used for the experiments is a PowerEdge R630 server (128GB, 2 proc. Intel E5264 6c/12t) running on linux. The **SCHED+MSS** and **SEQ+ALNS** models were implemented on OscaR [19] running on Scala 2.12.4. The **LIU_CPO** model was implemented using the Java API of CPLEX Optimization Studio V12.8 [7].

Tests were performed on the benchmark of PTP instances used in [5].

We compare the improvement ratio ρ_m of the three methods (m) relative to the initial solution provided by **GREEDY**: $\rho_m = \frac{x_m - x_{\text{GREEDY}}}{x_{\text{GREEDY}}}$ where x_m is the number of requests served with model m .

Results are displayed in Table 3. Highlighted cells show dominant solutions for the instance. **LIU_CPO** and **SEQ+ALNS** outperform **SCHED+MSS**. The average improvement ratio is 14.6 for **SCHED+MSS**, 21.8 for **SEQ+ALNS** and 22.6 for **LIU_CPO**. **SCHED+MSS** is clearly dominated by the two other approaches but **SEQ+ALNS** and **LIU_CPO** cannot really be compared. This indicates that the Incremental Sequence approach is worth considering for solving PTP like problems at the frontier between vehicle routing and scheduling.

7 Conclusion

The Patient Transportation Problem consists in allocating and sequencing a fleet of heterogeneous vehicles to attend their medical appointments. The particularity of this problem is the optionality of elements in the sequence. To provide a flexible and efficient model for this problem, we proposed a variable called *Incremental Sequence Variable* with a domain representation that extends the classical one for set variables allowing the additional `append` domain modification. This variable is independent from scheduling and could be used in other contexts such as traffic engineering problems for computer networks [21]. We described the constraints and filtering algorithms on the sequence variable required to solve the PTP. Experimental results on a large variety of PTP benchmarks demonstrated that the proposed approach is competitive with the one using sequence variables in the well-engineered CP Optimizer.

In our future work, we plan to extend the approach to deal with the dynamic aspect of the problem (new requests, or modification/cancellation of old ones can occur). Explaining the transition times constraint [22] would also be an interesting research direction. Finally, we plan to add support for sequence variables in modelisation languages such as XCSP [23] or MiniZinc [24, 25].

Table 3: Experimental results ($|R|$, $|V|$ and $|H|$ are the number of requests, vehicles and hospitals ; ρ is the improvement ratio in percent, * indicates that the solution has been proven optimal).

Difficulty	Name	Instances			GREEDY		SCHED+MSS		LIU_CPO		SEQ+ALNS	
		$ H $	$ V $	$ R $	BestSol	Sol	Sol	ρ	Sol	ρ	Sol	ρ
Easy	RAND-E-1	4	2	16	*15	14	15	7.1	*15	7.1	15	7.1
	RAND-E-2	8	4	32	*32	*32	*32	0.0	*32	0.0	*32	0.0
	RAND-E-3	12	5	48	*28	26	28	7.7	*28	7.7	28	7.7
	RAND-E-4	16	6	64	*64	60	62	3.3	*64	6.7	*64	6.7
	RAND-E-5	20	8	80	*79	73	74	1.4	*79	8.2	79	8.2
	RAND-E-6	24	9	96	*96	91	93	2.2	*96	5.5	96	5.5
	RAND-E-7	28	10	112	*112	102	106	3.9	*112	9.8	109	6.9
	RAND-E-8	32	12	128	*128	*128	*128	0.0	*128	0.0	*128	0.0
	RAND-E-9	36	14	144	*144	142	142	0.0	*144	1.4	*144	1.4
	RAND-E-10	40	16	160	*160	157	158	0.6	159	1.3	*160	1.9
Medium	RAND-M-1	8	2	16	*12	8	11	37.5	*12	50.0	12	50.0
	RAND-M-2	16	3	32	*20	18	20	11.1	*20	11.1	20	11.1
	RAND-M-3	24	4	48	*35	26	32	23.1	*35	34.6	34	30.8
	RAND-M-4	32	4	64	42	25	35	40.0	41	64.0	42	68.0
	RAND-M-5	40	5	80	68	45	58	28.9	68	51.1	63	40.0
	RAND-M-6	48	5	96	59	40	50	25.0	59	47.5	58	45.0
	RAND-M-7	56	6	112	74	46	60	30.4	74	60.9	69	50.0
	RAND-M-8	64	8	128	94	72	82	13.9	94	30.6	94	30.6
	RAND-M-9	72	8	144	92	65	82	26.2	92	41.5	87	33.8
	RAND-M-10	80	9	160	111	75	97	29.3	111	48.0	103	37.3
Hard	RAND-H-1	16	2	16	*8	7	8	14.3	*8	14.3	8	14.3
	RAND-H-2	32	3	32	*19	15	18	20.0	*19	26.7	19	26.7
	RAND-H-3	48	4	48	*34	19	31	63.2	*34	78.9	34	78.9
	RAND-H-4	64	4	64	*25	11	22	100.0	*25	127.3	25	127.3
	RAND-H-5	80	5	80	*48	30	42	40.0	*48	60.0	45	50.0
	RAND-H-6	96	5	96	*47	22	38	72.7	*47	113.6	46	109.1
	RAND-H-7	112	6	112	44	25	38	52.0	40	60.0	44	76.0
	RAND-H-8	128	8	128	85	58	76	31.0	85	46.6	85	46.6
	RAND-H-9	144	8	144	87	54	74	37.0	82	51.9	87	61.1
	RAND-H-10	160	8	160	83	48	69	43.8	79	64.6	80	66.7
Real	REAL-1	1	9	2	*2	*2	*2	0.0	*2	0.0	*2	0.0
	REAL-2	1	9	2	*2	*2	*2	0.0	*2	0.0	*2	0.0
	REAL-3	3	9	3	*1	1	1	0.0	*1	0.0	1	0.0
	REAL-4	2	9	4	*4	*4	*4	0.0	*4	0.0	*4	0.0
	REAL-5	5	9	21	*21	*21	*21	0.0	*21	0.0	*21	0.0
	REAL-6	5	9	22	*22	*22	*22	0.0	*22	0.0	*22	0.0
	REAL-7	5	9	23	*23	*23	*23	0.0	*23	0.0	*23	0.0
	REAL-8	7	9	24	*24	*24	*24	0.0	*24	0.0	*24	0.0
	REAL-9	15	9	45	*44	44	44	0.0	*44	0.0	44	0.0
	REAL-10	26	9	99	*98	98	98	0.0	*98	0.0	98	0.0
	REAL-11	22	9	100	*92	89	90	1.1	*92	3.4	92	3.4
	REAL-12	32	9	101	*100	98	98	0.0	*100	2.0	100	2.0
	REAL-13	37	9	110	*107	98	103	5.1	*107	9.2	107	9.2
	REAL-14	28	9	111	*102	100	101	1.0	*102	2.0	102	2.0
	REAL-15	35	9	122	116	97	108	11.3	116	19.6	115	18.6
	REAL-16	36	9	123	*117	107	109	1.9	*117	9.3	112	4.7
	REAL-17	42	9	128	121	103	112	8.7	121	17.5	119	15.5
	REAL-18	31	9	130	*126	112	120	7.1	*126	12.5	124	10.7
	REAL-19	34	9	131	120	107	115	7.5	120	12.1	120	12.1
	REAL-20	34	9	134	124	107	113	5.6	119	11.2	122	14.0
	REAL-21	39	9	136	*126	112	117	4.5	*126	12.5	123	9.8
	REAL-22	31	9	138	*131	115	119	3.5	*131	13.9	127	10.4
	REAL-23	31	9	139	133	113	119	5.3	132	16.8	133	17.7
	REAL-24	37	9	139	122	103	109	5.8	115	11.7	119	15.5
	REAL-25	39	9	139	130	119	125	5.0	128	7.6	130	9.2
	REAL-26	38	9	140	*131	110	116	5.5	*131	19.1	119	8.2
	REAL-27	35	9	147	137	121	129	6.6	134	10.7	135	11.6
	REAL-28	34	9	151	143	117	128	9.4	143	22.2	135	15.4
	REAL-29	39	9	155	134	120	125	4.2	128	6.7	126	5.0
	REAL-30	41	9	159	132	116	129	11.2	124	6.9	132	13.8

References

1. Cordeau, J.F., Laporte, G.: The dial-a-ride problem: models and algorithms. *Annals of Operations Research* **153** (2007) 29–46
2. Jain, S., Van Hentenryck, P.: Large neighborhood search for dial-a-ride problems. In: International Conference on Principles and Practice of Constraint Programming, Springer (2011) 400–413
3. Kilby, P., Shaw, P.: Vehicle routing. In: Foundations of Artificial Intelligence. Volume 2. Elsevier (2006) 801–836
4. Mouthuy, S., Massen, F., Deville, Y., Van Hentenryck, P.: A multistage very large-scale neighborhood search for the vehicle routing problem with soft time windows. *Transportation Science* **49** (2015) 223–238
5. 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, Springer (2018) 490–506
6. 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, Springer (2018) 412–428
7. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Ibm ilog cp optimizer for scheduling. *Constraints* **23** (2018) 210–250
8. Laborie, P., Rogerie, J.: Reasoning with conditional time-intervals. In: FLAIRS conference. (2008) 555–560
9. 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)
10. Gervet, C.: Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* **1** (1997) 191–244
11. de Saint-Marcq, V.I.C., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS). (2013) 1–10
12. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
13. Van Hentenryck, P., Carillon, J.P.: Generality versus specificity: An experience with ai and or techniques. In: AAAI. (1988) 660–664
14. Aggoun, A., Beldiceanu, N.: Extending chip in order to solve complex scheduling and placement problems. *Mathematical and computer modelling* **17** (1993) 57–73
15. Gay, S., Hartert, R., Schaus, P.: Simple and scalable time-table filtering for the cumulative constraint. In: International conference on principles and practice of constraint programming, Springer (2015) 149–157
16. Thomas, C., Schaus, P.: Revisiting the self-adaptive large neighborhood search. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Springer (2018) 557–566
17. Laborie, P., Godard, D.: Self-adapting large neighborhood search: Application to single-mode scheduling problems. *Proceedings MISTA-07, Paris* **8** (2007)
18. Godard, D., Laborie, P., Nuijten, W.: Randomized large neighborhood search for cumulative scheduling. In: ICAPS. Volume 5. (2005) 81–89
19. OscaR Team: OscaR: Scala in OR (2012) Available from <https://bitbucket.org/oscarlib/oscar>.
20. Thomas, C., Cappart, Q., Schaus, P., Rousseau, L.M.: CSPLib problem 082: Patient transportation problem. <http://www.csplib.org/Problems/prob082> (2018)

21. 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, Springer (2015) 592–608
22. Francis, K.G., Stuckey, P.J.: Explaining circuit propagation. *Constraints* **19** (2014) 1–29
23. Boussemart, F., Lecoutre, C., Audemard, G., Piette, C.: Xcsp3: An integrated format for benchmarking combinatorial constrained problems. arXiv preprint arXiv:1611.03398 (2016)
24. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: International Conference on Principles and Practice of Constraint Programming, Springer (2007) 529–543
25. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: Minizinc with strings. In: International Symposium on Logic-Based Program Synthesis and Transformation, Springer (2016) 59–75