CASPT 2015

Constraint Propagation for the Dial-a-Ride Problem with Transfers

Samuel Deleplanque · Alain Quilliot

Abstract The Dial-a-Ride Problem (DARP) is an operation research problem which models a standard demand responsive transport system. This research considers one important feature: the transfers. We solve the Dial-a-Ride Problem with Transfers (DARPT) by a Monte Carlo procedure with a greedy insertion algorithm based on constraint propagation of the main time constraints (time windows, maximum ride times, maximum route times and time constraints related to the transfers).

Keywords: On-Demand Transportation · Transfers · Dial-a-Ride Problems

1 Introduction

The Dial-A-Ride Problem (DARP) is an operation research model for on-demand transportation. This system allows elderly and disabled people to move a short distance providing ride requests from an origin to a destination. These demands include hard time constraints (maximum ride times and time windows), while the system must fulfil the maximum route times and capacity constraints related to the fleet of vehicles.

The latest evolution in transportation could enable on-demand transportation systems to be more widely and commonly used. The technologies like geo-localization, mobile communication, connected cars and autonomous cars indeed provide new services for a more efficient system in terms of costs and quality of service. This evolution could explain why the DARP is experiencing a resurgence in the operation research literature since the last decade.

Laporte et al. (2007) gives different ways to model the DARP with ILP. However, the complexity of the problem (NP-Hard) and the responsiveness expected for this type of system (there are short time intervals between the transmission of the

Samuel Deleplanque
GOM, Computer Science Department,
Université Libre de Bruxelles, Belgium
Email: **sdelepla@ulb.ac.be**

Alain Quilliot
LIMOS, UMR CNRS 6158, Blaise Pascal University,
Clermont-Ferrand, France
Email: **quilliot@isima.fr**

demands and the rides) forces the problem to be handled through heuristic techniques. One of the main research papers on the DARP is Cordeau et al. (2003). They developed an efficient metaheuristic based on the Tabu Search to solve randomly generated instances which contain between 24 and 144 requests. Dynamic Programming (Psaraftis (1983)) and Variable Neighborhood Search (Healy et al. (1995), Parragh et al. (2010)]) are also good techniques to deal with the static problem.

Algorithms should be designed in order to be easily adapted to a dynamic context and a flow of uncertain demands. Madsen et al. (1995) and Psaraftis et al. (1986) proposed such an algorithm which is the most widely used for the dynamic DARP: an insertion heuristic.

The problem studied in this research allows the vehicles to satisfy a demand with transfers (or "transshipment"). This problem is named the Dial-A-Ride Problem with Transfers (DARPT). Masson et al. (2014) established its formulation for the first time. Here, the locations of the transfer nodes are static. Indeed, the transfer points could be either an origin node or a destination of any demand.

We may refer to the Pickup and Delivery Problem with Transfers (PDPT) for a larger state of the art. For some exacts methods, we may refer to the Cortes et al. (2010), Kerivin (2008) and Nakao et al. (2012). However, as stated previously, the exact methods are not well adapted to the DARP (and thus also to the DARPT), only approximate methods are able to solve the problem in time. In this context, Masson et al. (2013) develops a VNS and solves big instances with almost 200 requests. Shang et al. (1996) proposes an insertion heuristic where each pick-up and delivery nodes could be a transfer point. The transshipments are only allowed for the demands which couldn't be inserted. This algorithm is adapted to the dynamic case by Thangiah et al. (2007).

In this paper, we are solving the DARP and the DARPT by two randomized insertion heuristics into a Monte Carlo Process based on Deleplanque et al. (2013). We optimize an objective function with a mixed QoS/Economical Cost performance criterion. The validity of each insertion is mainly tested by the propagation of time constraints. The precedence constraints due to the transfers are included in this process. Their locations are located dynamically according to the shortest path joining the routes of the two vehicles involved. In the next sections, we present models of the DARP and the DARPT. Then, we describe and test our heuristics which propagate the time constraints for both problems.

2. The DARP: model and framework

2.1 The general Dial-a-Ride Problem

A *Dial a Ride* problem instance is essentially defined by a vehicle fleet $\mathcal{VH}$, a *transit* network **G** = (V, E). The graph contains a node *Depot*, and a *Demand* set $\mathcal{D}$ = ($\mathcal{D}_i$, i ∈ I). Any demand $\mathcal{D}_i$ is defined as a 6-uple $\mathcal{D}_i$ = (o_i, d_i, Δ_i, $\mathcal{F}(o_i)$, $\mathcal{F}(d_i)$, Q_i) such that:

- $\mathbf{o_i} \in V$ is the *origin* node of the demand $\mathcal{D}_i$,
- $\mathbf{d_i} \in V$ is the *destination* node of the demand $\mathcal{D}_i$,
- $\mathbf{\Delta_i} \geq 0$ is an upper (*transit*) bound on the time of demand $\mathcal{D}_i$'s processing,
- $\mathcal{F}(\mathbf{o_i})$ is a time window related to the time $\mathcal{D}_i$ starts being processed, $\mathcal{F}(\mathbf{d_i})$ is a time window related to the time $\mathcal{D}_i$ ends being processed,
- $\mathbf{Q_i}$ is a description of the load related to $\mathcal{D}_i$.

Dealing with such an instance means planning the handling demands of $\mathcal{D}$, by the fleet $\mathcal{VH}$, while taking into account the constraints which derive from the technical characteristics of the network G, of the vehicle fleet $\mathcal{VH}$, the demands of $\mathcal{D}$, and while optimizing some performance criteria. The objective is usually a mix of an economical cost (point of view of the fleet manager) and of QoS criteria (point of view of the users). This very general problem may be specialized. It depends on the structure of the fleet $\mathcal{VH}$ and on the way this fleet is allowed to answer various demands of $\mathcal{D}$. Temporal constraints related to the time windows $\mathcal{F}(o_i)$, $\mathcal{F}(d_i)$, i ∈ I, and to the transit bounds Δ_i, i ∈ I, may be more and less tight.

The problem may have to be handled according to a *dynamic* context, (demands are not known in advance and must be processed "online"). In such a case, one must take into account the way the system is supervised and the way its components communicate with the users. Conversely, it may be set in a static context where all data are known in advance. In this case, eventual divergences between the data which were used during the planning phases, and the actual situation the system has to face, put what is called *robustness* at stake.

Finally, one may have to tackle additional constraints, such that cumulative constraints involving human, technical or financial mutualized renewable or non-renewable resources. In such a case, one may thing into linking the reduced problem with the RCPSP (*Resource Constrained Project Scheduling Problem*) framework.

Throughout this work, we deal with homogeneous fleets and therefore limit ourselves to static points of view. Still, we do not intend to restrict ourselves to *Standard Dial a Ride*: so, we pay special attention to cases when temporal constraints are tight, and handle the case when the transfers are allowed.

2.2 The Standard Case Framework

The general notations of sequences and algorithms are described as follows. In any algorithmic description, we use the symbol ← in order to denote the value assignment operator: x ← α, means that the variable x receives the value α. Thus, we only use the symbol = as a comparator. For any sequence (or list) Γ whose elements belong to some set Z, we set First(Γ) the first element of Γ and Last(Γ) is the last element of Γ. For any z in Γ, Succ(Γ, z) gives the successor of z in Γ and Pred(Γ, z) gives the predecessor of z in the route Γ. For any z, z' in Γ, we use $z <<_\Gamma z'$ to denote that z is located before z' in Γ and $z <<_\Gamma^{=} z'$ if $z <<_\Gamma z'$ or z = z'.

We do not need to consider the whole transit network G = (V, E). We may restrict ourselves to the nodes which are either the origin or the destination of some

demand, while considering that any vehicle which visits two such nodes in a consecutive way does it according to a shortest path strategy. This leads us to consider the node set {*Depot*, o_i, d_i, $i \in I$} as made with pairwise distinct nodes, and provided with some distance function DIST, which to any pair x, y in {*Depot*, o_i, d_i, $i \in I$}, corresponds to the shortest path distance from x to y in the transit network G.
Additionally, we also split the *Depot* node according to its arrival or departure status and to the various vehicles of the fleet $\mathcal{VH}$. We also consider the input data of a *Standard Dial a Ride* instance as defined by the set {1..K = Card($\mathcal{VH}$)} of the vehicles of the homogenous fleet $\mathcal{VH}$, the common capacity $\mathcal{CAP}$ of a vehicle in $\mathcal{VH}$, the node set X = {*DepotD(k)*, *DepotA(k)*, k = 1..K} $\cup$ {o_i, d_i, $i \in I$} and the distance matrix DIST. For any x, y in X, DIST(x, y) is equal to the length, in the sense of the length function l, of a shortest path which connects x to y in G: we suppose that DIST, satisfies the *triangle inequality*.
For any node x in X, the following characteristics also apply:

- its status $\mathcal{Status}$(x): *Origin*, *Destination*, *DepotA*, *Depot D*. We set *Depot* = *DepotD* $\cup$ *DepotA*;
- its demand index: $\mathcal{Dem}$(x) = i if x = o_i or d_i, and $\mathcal{Dem}$(x) = 0 else;
- its vehicle index $\mathcal{VI}$: $\mathcal{VI}$(*DepotA*(k)) = $\mathcal{VI}$(*DepotD*(k)) = k and $\mathcal{VI}$(x) = *Undefined* for any other node x $\in$ X;
- its load $\mathcal{CH}$(x): if $\mathcal{Status}$(x) $\in$ *Depot* then $\mathcal{CH}$(x) = 0; if $\mathcal{Status}$(x) = *Origin*, then $\mathcal{CH}$(x) = Q_i; and if $\mathcal{Status}$(x) = *Destination*, then $\mathcal{CH}$(x) = $-Q_i$;
- its twin node $\mathcal{Twin}$(x): if x = *DepotA*(k) then $\mathcal{Twin}$(x) = *DepotD*(k) and conversely; if x = o_i then $\mathcal{Twin}$(x) = d_i and conversely;
- its time window $\mathcal{F}$(x): for any k = 1..K, $\mathcal{F}$(*DepotA*(k)) = [0, + ∞[= $\mathcal{F}$(*DepotD*(k)). Also, we suppose that any $\mathcal{F}$(x), x $\in$ X, is an interval, which may be written $\mathcal{F}$(x) = [$\mathcal{F}$.min(x), $\mathcal{F}$.max(x)];
- its transit bound Δ(x): if x = o_i (d_i), then Δ(x) = Δ_i, and Δ(x) = Δ else. Δ is an upper bound which is imposed on the duration of any vehicle tour.

The system works as follows: vehicle k $\in$ {1..K}, starts its journey from *DepotD*(k) at some time t(*DepotD*(k)) and ends it into *DepotA*(k) at some time t(*DepotA*(k)). It took in charge some subset $\mathcal{D}$(k) = {$\mathcal{D}_i$, i $\in$ I(k)} of $\mathcal{D}$. That means that for any i in I(k), vehicle k arrived in o_i at time t(o_i) $\in \mathcal{F}(o_i)$, loaded the whole load Q_i, and kept it until it arrived in d_i at time t(d_i) $\in \mathcal{F}(o_i)$ and unloaded Q_i, in such a way that t(d_i) - t(o_i) $\leq \Delta_i$. Clearly, solving the Standard Dial a Ride instance related to this data (X, DIST, K, CAP) will mean computing the subsets $\mathcal{D}$(k) = {$\mathcal{D}_i$, i $\in$ I(k)}, the routes followed by the vehicles and the time values t(x), x $\in$ X, in such a way that both economic performance and quality of service be the highest possible.

Tours, Valid Tour and performance criteria

In order to provide an accurate description of the output data of our standard Dial a Ride instance (X, DIST, K, $\mathcal{CAP}$), we need to talk about *tours* and related *time value sets*. A *tour* Γ is a sequence of nodes of X, which is such that:

- $\mathcal{Status}$(First(Γ)) = *DepotD*; $\mathcal{Status}$(End(Γ)) = *DepotA*;
- $\mathcal{VI}$(First(Γ)) = $\mathcal{VI}$(End(Γ));
- For any node x in Γ, x $\neq$ First(Γ), End(Γ), $\mathcal{Status}$(x) $\notin$ *Depot*;

- No node $x \in X$ appears twice in Γ;
- For any node $x = o_i$ (resp. d_i) which appears in Γ, the node $\mathcal{T}win(x)$ is also in Γ, and we have: $x <<_\Gamma \mathcal{T}win(x)$ (resp. $\mathcal{T}win(x) <<_\Gamma x$).

This tour Γ is *load-valid iff*: for any x in Γ, $x \neq First(\Gamma)$, we have $\Sigma_{y \backslash y <<_\Gamma x} \mathcal{CH}(y) \leq \mathcal{CAP}$. Moreover, this tour Γ is *time-valid iff* it is possible to associate, with any node x in Γ, some time value t(x), in such a way that: **(E1)**

- for any x in Γ, $x \neq Last(\Gamma)$, $\mathbf{t(Succ(\Gamma, x)) \geq t(x) + DIST(x, Succ(\Gamma, x))}$;
- for any x in Γ, $\mathbf{|t(\mathcal{T}win(x)) - t(x)| \leq \Delta(x)}$ and for any x in Γ, $\mathbf{t(x) \in \mathcal{F}(x)}$.

The tour Γ is said to be **valid** if it is both *time-valid* and *load-valid*. For any pair (Γ, t) defined by some *time-valid* tour Γ and by some valid related time value set t, we may set $\mathbf{\mathcal{G}lob(\Gamma, t) = t(End(\Gamma)) - t(First(\Gamma))}$: this quantity denotes the global duration of the tour Γ and $\mathbf{\mathcal{R}ide(\Gamma, t) = \Sigma_{x \backslash Status(x)=Origin} |\mathcal{T}win(x) - t(x)|}$: this other quantity may be viewed as a QoS criteria, and denotes the sum of the duration of the individual trips of the demanders which are taken in charge by tour Γ. If A and B are two multi-criteria coefficients, we may define the performance criteria $Cost_{A,B}(\Gamma, t)$ as follows: $\mathbf{Cost_{A,B}(\Gamma, t) = A.\mathcal{G}lob(\Gamma, t) + B.\mathcal{R}ide(\Gamma, t)}$.

So, let us suppose that we deduced from the data G = (V, E), $\mathcal{VH}$ = (K, $\mathcal{CAP}$), $\mathcal{D}$ = ($\mathcal{D}_i$ = (o_i, d_i, Δ_i, $\mathcal{F}(o_i)$, $\mathcal{F}(d_i)$, Q_i), $i \in I$), a 4-uple (X, DIST, K, $\mathcal{CAP}$), and that we are also provided with 2 multi-criteria coefficients A and $B \geq 0$. Then we see that solving the related *Standard Dial a Ride Problem* instance means computing: for any vehicle index k in 1..K, a valid tour T(k) and a time value set $t = \{t(x), x \in X\}$ in such a way that: the restriction of t to any T(k), k = 1..K, defines a valid time value set related to T(k), the tour set $T = \{T(k), k = 1..K\}$ induces a partition of X, and the quantity $\mathcal{P}erf_{A,B}(T, t) = \Sigma_{k = 1..K} Cost_{A,B}(T(k), t)$ is the smallest possible.

3 Handling temporal constraints

The algorithm described inside the next section will essentially be based upon the use of insertion techniques. Thus, we must be able to check in a fast way whether the insertion of some demand $\mathcal{D}_i$ inside a tour Γ will maintain the validity of Γ, and to get an evaluation of the quality of this insertion. We are first going to define a package of constraint handling tools.

3.1 Testing the load-validity and the Time-validity

Checking the load validity on Γ is straightforward. In order to be able to test the impact of the insertion of some demand into the tour Γ on the load validity of this tour, we associate, with any such a tour, the quantities $\mathcal{C}(\Gamma, x)$, $x \in \Gamma$, defined by: for any x in Γ, $\mathcal{C}(\Gamma, x) = \Sigma_{y \backslash y <<_x \text{ or } y = x} \mathcal{CH}(y)$. Subsequently, Γ is *load-valid iff* for any x in Γ, $\mathcal{C}(\Gamma, x) \leq \mathcal{CAP}$.

Checking the time validity of Γ, according to a current time window set $\mathcal{FS} = \{\mathcal{FS}(x) = [\mathcal{FS}.min(x), \mathcal{FS}.max(x)], x \in \Gamma\}$, may be performed through propagation of the following inference rules R_i, i = 1..5 performed by the *Propagate* procedure and we deduce the proposition 1:

- **Rule R_1**: y = Succ(Γ, x); $\mathcal{FS}$.min(x) + DIST(x, y) > $\mathcal{FS}$.min(y) ⊨ $\mathcal{FS}$.min(y) ← $\mathcal{FS}$.min(x) + DIST(x, y); NFact ← y;
- **Rule R_2**: y = Succ(Γ, x); $\mathcal{FS}$.max(y) - DIST(x, y) < $\mathcal{FS}$.max(x) ⊨ $\mathcal{FS}$.max(x) ← $\mathcal{FS}$.max(y) - DIST(x, y); NFact ← x;
- **Rule R_3**: y = Twin(x); x $<<_\Gamma$ y; $\mathcal{FS}$.min(x) < $\mathcal{FS}$.min(y) – Δ(x) ⊨ $\mathcal{FS}$.min(x) ← $\mathcal{FS}$.min(y) - Δ(x); NFact ← x;
- **Rule R_4**: y = Twin(x); x $<<_\Gamma$ y; $\mathcal{FS}$.max(y) > $\mathcal{FS}$.max(x) + Δ(x) ⊨ $\mathcal{FS}$.max(y) ← $\mathcal{FS}$.max(x) + Δ(x) ; NFact ← y;
- **Rule R_5**: x ∈ Γ; $\mathcal{FS}$.min(x) > $\mathcal{FS}$.max(x) ⊨ *Fail*.

Procedure Propagate
Input: (Γ: Tour, L: List of nodes, $\mathcal{FS}$: Time windows related to the node set of Γ);
Output: ($\mathcal{Res}$: Boolean, $\mathcal{FR}$: Time windows related to node set of Γ);
Continue ← true;
While L ≠ Nil and Continue do
    z ← First(L); L ← Tail(L);
    For i = 1..5, compute all the pairs (x, y) which make possible an application of the rule R_i and which are such that x = z or y = z;
    For any such pair (x, y) do
        Apply the rule R_i;
        If NFact is not in L then insert NFact in L;
        If *Fail* then Continue ← false;
Propagate ← (Continue, $\mathcal{FS}$);

Proposition 1.
The tour Γ is time-valid according to the input time window set $\mathcal{FS}$ if and only if the $\mathcal{Res}$ component of the result of a call Propagate($\mathcal{FS}$, Γ) is equal to 1. In such a case, any valid time value set t related to Γ and $\mathcal{FS}$ is such that: for any x in Γ, t(x) ∈ $\mathcal{FS}$(x).

Proof. The part (only if) of the above equivalence is trivial, as well as the second part of the statement. As for the part (if), we only need to check that if we set, for any x in Γ: $\mathcal{FS}$(x) = [$\mathcal{FS}$.min(x), $\mathcal{FS}$.max(x)] and t(x) = $\mathcal{FS}$.min(x); then we get a time value set t ={t(x), x ∈ X(Γ)}which is compatible with Γ and $\mathcal{FS}$. ***End-Proof*.**

We denote by $\mathcal{FP}$(Γ) the time window set which result from a call *Propagate*(Γ, L, $\mathcal{F}$). $\mathcal{FP}$(Γ) may be considered as the largest (in the inclusion sense) time window set which is included into $\mathcal{F}$ and which is stable under the rules R_i, i = 1..5, and is called the *window reduction* of $\mathcal{F}$ through Γ.

3.2 Evaluating a tour

Let us consider now the tour Γ, provided with the window reduction set $\mathcal{FP}$(Γ). We want to get some fast estimation of the best possible value $Cost_{A,B}$(Γ, t) = A.$\mathcal{Glob}$(Γ,

t) + B.$\mathcal{R}ide$(Γ, t), t ∈ $\mathcal{V}alid$(Γ). We design two ad hoc procedures EVAL1 and EVAL2. The EVAL1 procedure works in a greedy way, first by assigning the node First(Γ) its largest possible time value, and by next performing a Bellman process in order to assign every node x in Γ its smallest possible time value. The EVAL2 procedure starts from a solution produced by EVAL1, and improves it by performing a sequence of local moves, each move involving a single value t(x), x ∈ Γ. These procedures and the Proposition 2 are given below.

Procedure EVAL1. **Input:**(Γ: Tour); **Output:** (Val: Number, δ: value set);
For any x in Γ, let us set set: [a(x), b(x)] = $\mathcal{FP}$(Γ);
δ(First(Γ)) ← b(First(Γ)); x ← First(Γ);
While x ≠ Last(Γ) do
y < Succ(Γ, x); δ(y) ← Sup(a(y), δ(x) + DIST(x, y));
x ← y; δ ← {δ(x), x ∈Γ}; Val ← $Cost_{A,B}$(Γ, δ);
EVAL1← (Val, δ);

Procedure EVAL2. **Input:**(Γ: Tour); **Output:** (Val: Number, δ: value set);
For any x in Γ, let us set: [a(x), b(x)] = $\mathcal{FP}$(Γ);
For any x in Γ do δ(x) ←EVAL1(Γ, $\mathcal{FS}$).δ; Continue ← true;
While Continue do
Search for x in Γ such that one of the two statements **(E2)** or **(E3)** below is true:

- **(E2)**: (λ_x< 0) ∧ ($\mathcal{S}tatus$(x) ∈ {*Origin*, *DepotD*}) ∧ (δ(x) ≠ Inf(b(x), δ(Succ(Γ, x) – DIST(x, Succ(Γ, x)));
- **(E3)**: (λ_x> 0) ∧ ($\mathcal{S}tatus$(x) ∈ {*Destination*, *DepotA*}) ∧ (δ(x) ≠ Sup(a(x), δ(Pred(Γ, x) + DIST(Succ(Γ, x)), x));

If $\mathcal{F}ail$(Search) then Continue ← false;
Else
If **(E2)** then δ(x) ←Inf(b(x), δ(Succ(Γ, x) – DIST(x, Succ(Γ, x)));
If **(E3)** then (δ(x) ←Sup(a(x), δ(Pred(Γ, x) + DIST(Succ(Γ, x)), x));
EVAL2← (Val = $Cost_{A,B}$(Γ, δ)), δ);

Proposition 2.
Both EVAL1 and EVAL2 yield a time value set δ which is compatible with Γ and $\mathcal{F}$ (with Γ and $\mathcal{FP}$(Γ)). Besides, if B = 0, then EVAL1 yields an optimal value Val, that means yields the smallest possible value $Cost_{A,B}$(Γ, δ), δ ∈ $\mathcal{V}alid$(Γ, $\mathcal{F}$).

Proof. As in the description of both procedures EVAL1 and EVAL2, we suppose that for any x in Γ, the time window $\mathcal{FP}$(Γ) may also be written $\mathcal{FP}$(Γ) = [a(x), b(x)]. The first part of the above statement is trivial. In case B = 0, minimizing $Cost_{A,B}$(Γ, δ) means minimizing δ(Last(Γ)) – δ(First(Γ)). We must deal with two cases:

- **1**. There exists x ∈ First(Γ) such that: δ(x) = a(x), x $<<_\Gamma^=$ y $<<_\Gamma$ Last(Γ). For all y, we have: δ(Succ(Γ, y)) – δ(y) = DIST(y, Succ(Γ, y));
  Then the stability of $\mathcal{FP}$(Γ)(x) under the inference rule $\mathbf{R_3}$ allows us to deduce δ(Last(Γ)) = a(Last(Γ)), and the result since δ(First(Γ)) = b(First(Γ)).

- **2**. For any x in X(Γ), x ≠ Last(Γ), we have δ(Succ(Γ, x)) – δ(x) = DIST(x, Succ(Γ, x)).

Then the result comes in an immediate way. ***End-Proof.***

Γ being some valid tour, we denote by VAL1(Γ) and VAL2(Γ) the values respectively produced by the application of EVAL1 and EVAL2 to Γ.

4 An insertion algorithm for tightly constrained instances

4.1 The insertion mechanism

The algorithm works in a very natural way. Let Γ be some valid tour, let $\mathcal{D}_i$ = (o_i, d_i, Δ_i, $\mathcal{F}(o_i)$, $\mathcal{F}(d_i)$, Q_i) be some demand whose origin and destination nodes are not in Γ, and let x, y be two nodes in Γ, such that x $<<_\Gamma^=$ y. Then we denote by INSERT(Γ, x, y, i) the tour which is obtained by locating o_i between x and Succ(Γ, x) and locating d_i between y and Succ(Γ, y). We say that the tour INSERT(Γ, x, y, i) results from the insertion of demand D_i into the tour Γ according to the insertion nodes x and y. The tour INSERT(Γ, x, y, i) may not be valid. So, before anything else, we must detail the way the validity of this tour is likely to be tested.

Testing the Load-Admissibility of INSERT(Γ, x, y, i).
We only need to check with a procedure **Test-Load**, that for any z in Segment(Γ, x, y) = {z such that x $<<_\Gamma^=$ z $<<_\Gamma^=$ y} we have, $\mathcal{C}(\Gamma, z) + Q_i \leq \mathcal{CAP}$.

Testing the Time-Admissibility of INSERT(Γ, x, y, i).
It should be sufficient perform a call *Propagate*(Γ, {o_i, d_i}, $\mathcal{FP}(\Gamma)$), while using the list {o_i, d_i} as a starting list. Still, such a call is likely to be time consuming. So, in order to make the testing process go faster, we introduce several intermediary tests which aim at interrupting the testing process in case non-feasibility. The first test *Test-Node* aims at checking the feasibility of the insertion of a node u, related to some load Q, between two consecutive node z and z' of a given tour Γ. It only provides us with a necessary condition for the feasibility of this insertion:

Procedure Test-Node
Input: (Γ, z, z': nodes in Γ, u: node out Γ, Q: load); **Output:** Boolean
Let us set, for any x in Γ, [a(x), b(x)] = $\mathcal{FP}(\Gamma)(x)$; Let us set: [α, β] = $\mathcal{F}(u)$;
Test node ← (a(z) + DIST(z, u) ≤ β) ∧ (α + DIST(u, z') ≤ b(z')) ∧ (a(z) + DIST(z, u) + DIST(u, z') ≤ b(z')) ∧ ($\mathcal{C}(\Gamma, z)$ + Q ≤ $\mathcal{CAP}$);

The second test *Test-Node1* (based on *Test-Node*) aims at checking the feasibility of the insertion of an origin u and a destination v nodes related to some load Q, between two consecutive nodes z and z' of a given tour Γ. So, testing the admissibility of a tour INSERT(Γ, x, y, i) may be performed through the following procedure:

Procedure Test-Insert
Input: (Γ, x, y, i); **Output:** (Test: Boolean, Val: Number);
If $x \neq y$ then Test ← *Test-Node*(Γ, x, Succ(Γ, x), o_i, Q_i) ∧ *Test-Node*(Γ, y, Succ(Γ, y), d_i, Q_i);
Else Test ← *Test-Node1*(Γ, x, Succ(Γ, x), o_i, d_i, Q_i);
If Test = 1 then Test ← *Test-Load*(Γ, x, y, i);
If Test = 1 then (Test, $\mathcal{F}1$) ← *Propagate*(Γ, {o_i, d_i}, $\mathcal{FP}$(Γ);
If Test = 1 then Val ← EVAL1(INSERT(Γ, x, y, i), $\mathcal{F}1$).Val else Val ← Undefined;
Test-Insert← (Test, Val – Val1(Γ));

4.2 The insertion process

So, this process takes as input the demand set $\mathcal{D}$ = ($\mathcal{D}_i$ = (o_i, d_i, Δ_i, $\mathcal{F}(o_i)$, $\mathcal{F}(d_i)$, Q_i), i ∈ I), the 4-uple (X, DIST, K, $\mathcal{CAP}$), and two multi-criteria coefficients A and B ≥ 0. The algorithm works in a greedy way through successive insertions of the various demands $\mathcal{D}_i$ of the demand set $\mathcal{D}$. The basic point is that, since we are concerned with tightly constrained time windows and transit bounds, we use, while designing the INSERTION algorithm, several constraint propagations tricks. Namely, any time we enter the main loop of this algorithm, we are provided with:

- the set $I_1 \subset I$ of the demands which have already been inserted;
- current tours T(k), k = 1..K: for any such a tour T(k), we know the related time windows $\mathcal{FP}$(T(k))(x), x ∈ T(k), as well as the load values $\mathcal{C}$(T(k), x), x ∈ T(k), and the values VAL1(T(k)) and VAL2(T(k));
- the knowledge, for any i in J = (I - I_1) of the set FREE(i) of all the 4-uple (k, x, y, v), k = 1..K, x, y ∈ T(k), v ∈ **Q**, such that a call **Test-Insert**(T(k), x, y, i) yields a result (1, v). We denote by N-FREE(i) the cardinality of the set V-FREE(i) = {k = 1..K, such that there exists a 4-uple (k, x, y, v) in FREE(i)}. N-FREE(i) is the number of vehicles available for $\mathcal{D}_i$.

Then, the INSERTION algorithm works according to the following scheme (1-4):

1. The process selects a demand i_0 in J, among those demands which are the most constrained: i_0 is such that N-FREE(i_0) and Card(Free(i_0)) are small. **(E4)**
2. Then, it picks up (k_0, x_0, y_0, v_0) in FREE(i_0) which corresponds to one of the smallest values EVAL2(INSERT(T(k), x, y, i_0)).Val – VAL2(T(k)). More specifically, it builds the list *L-Candidate* of the N_1 4-uples (k, x, y, v) in FREE(i_0) with the smallest value v. For any such a 4-uple, it computes the value w = EVAL2(INSERT(T(k), x, y, i_0)).Val – VAL2(T(k)), and it orders *L-Candidate* according to increasing values w. Then it randomly chooses (k_0, x_0, y_0, v_0) among those $N_2 \leq N_1$ first 4-uples in *L-Candidate.* **(E5)**
3. It inserts the demand $\mathcal{D}_{i0}$ into T(k_0) according to the insertion nodes x_0, y_0, which means that it replaces T(k_0) by INSERT(T(k_0), x_0, y_0, i_0). Then, it defines, for any i ∈ J, the set Λ(i) as being the set of all pairs (x, y) such that there exists some 4-uple (k_0, x', y', v) in FREE(i), which satisfies: **(E6)**

- (x' = x) or ((x' = x_0) and x' = Pred(T(k_0), x)) or ((x' = x_0= y_0) and (x' = Pred(Pred(T(k_0), x))));
- (y' = y) or ((y' = y_0) and y' = Pred(T(k_0), y)) or ((y' = x_0 = y_0) and (y' = Pred(Pred(T(k_0), y))))

4. Finally, it performs, for any pair (x, y) in $\Lambda(i)$, a call Test-Insert($T(k_0)$, x, y, i), and it updates FREE(i) and N-FREE(i) consequently.

Procedure INSERTION
Input: (N_1, N_2, $\mathcal{D}$, (*X, DIST, K, CAP*), *A and B*);
Output: (T, t, *Perf*: induced $Perf_{A,B}(T, t)$ value, *Reject*: rejected demand set);
For any k = 1..K do
    T(k) ← {*DepotD(k), DepotA(k)*}; t(*DepotD*(k)) = t(*DepotA*(k)) ← 0;
I_1←Nil; J ← I; *Reject*← Nil;
For any i ∈ J do
    FREE(i) ← all the possible 4-uple (k, x, y, v), k = 1..K, x, y ∈{*DepotD(k), DepotA(k)*}, $x <<_{T(k)}^{=} y$, v = EVAL2({*DepotD(k),* o_i, d_i, *DepotA(k)*}).Val; N-FREE(i) ← K;
While J ≠ Nil do
    Pick up some demand i_0 in J as in **(E4)**; Remove i_0 from J;
    If FREE(i_0) = Nil then *Reject* ←*Reject* ∪ {i_0}
    Else
        Derive from FREE(i_0) the *L-Candidate* list and Pick up (k_0, x_0, y_0, v_0) in *L-Candidate* as in **(E5)**;
        $T(k_0)$ ← INSERT($T(k_0)$, x_0, y_0, i_0); δ← EVAL2($T(k_0)$).δ; Insert $i_0$into I_1;
        For any x in $T(k_0)$ do t(x) ←δ(x);
        For any i ∈ J do
            Λ(i) ← {all pairs (x, y) such that there exists some 4-uple (k_0, x', y', v) in FREE(i), which satisfies **(E6)**
            For any pair (x, y) in Λ(i) do
                (Test, Val) ←**Test-Insert**($T(k_0)$, x, y, i);
                Remove (k_0, x, y, v) from FREE(i) in case such a 4-uple exists and update N-FREE(i) consequently;
                If Test = 1 then insert (k_0, x, y, Val) into FREE(i) and update N-FREE(i) consequently;
Perf ←$Perf_{A,B}$(T, t);
INSERTION← (T, t, *Perf, Reject*);

Since the above instruction may be written in a non-deterministic way, the whole INSERTION algorithm becomes non-deterministic inside some MONTE-CARLO framework. This process keeps the best result (the pair (T, t)) such that |*Reject*| is the smallest possible, and which is such that, among those pairs which minimize |*Reject*|, it yields the best $Perf_{A,B}$(T, t) value.

5 Dial a ride problem with transfers

In this section, we deal with the DARPT case when *transfers* are allowed. meaning then the load Q_i related to some demand $\mathcal{D}_i$ = (o_i, d_i, Δ_i, $\mathcal{F}(o_i)$, $\mathcal{F}(d_i)$, Q_i). $\mathcal{D}_i$ may be handled in several successive steps, each step involving some vehicle k ∈ K, which makes the load Q_i go from some origin or relay node x to some relay or destination node y. Transfers means here that, while the load Q_i is always handled as a whole,

the route it follows may be split into several sub-routes. All these sub-routes being taken in charge by distinct vehicles. We are going to restrict ourselves here to a case when no more than two vehicles are allowed to perform such a transportation task.

5.1 The insertion mechanism

In case a given load Q_i has to be successively handled by two vehicles k and k', the set X is likely not to be sufficient to describe the route of the vehicles. The two related routes T(k), T(k') will have to intersect and exchange load Q_i in a relay node z, which will be neither an origin node o_j nor a destination node d_j. Those relay nodes are not known in advance: so, we try to handle those exchange nodes in an implicit way and to deal with them in a dynamic way.

INPUT for the DARPT - Extending the node set X into an implicit node set Z.
We first need to extend X in order to create the relay nodes. Since we want to handle these relay nodes dynamically, we suppose that X may be embedded into some (eventually infinite) implicit node set Z such that $X \subset Z$: Z may be a large scale, eventually infinite, set. For any pair of nodes z, z' in Z, we suppose that we are able to compute some distance $\mathcal{Dist}(z, z')$, in such a way that for any x, x' in X, $\mathcal{Dist}(x, x') = DIST(x, x')$.
The input of the *Dial a Ride problem with transfers* (DARPT) is going to be defined by a transit network G = (V, E), a vehicle fleet $\mathcal{VH}$ = (K, $\mathcal{CAP}$), a demand set $\mathcal{D}$ = ($\mathcal{D}_i$ = (o_i, d_i, Δ_i, $\mathcal{F}(o_i)$, $\mathcal{F}(d_i)$, Q_i), i ∈ I), a 4-uple (X, DIST, K, $\mathcal{CAP}$), and with 2 multi-criteria coefficients A and B ≥ 0, augmented with some node set Z. In addition, since we are going to handle the node set Z a dynamic way, we should be able to create new active nodes from existing ones. So, we suppose that we are provided with a function $\mathcal{Midst}$, which, from any pair of nodes z, z' in Z, compute a new node z" = $\mathcal{Midst}(z, z')$ in Z, in such a way that Dist(z, z") and Dist(z", z) are no larger than some fraction λ.Dist(z, z'), with λ< 1.

Building Relay Nodes: the Implicit Set Z*.
Additional nodes in Z – X are going to be used as relay nodes. Any such active relay node will appear in two tours, once as an emitter node and once as a receiver node. Since we would like to continue with the structure of the model that we have been using for the standard version of the *Dial a Ride* problem, we also would like distinct nodes. In order to do it, we define the implicit node set Z* as follows:

- Z* = X ∪ {(z, i, -1), (z, i, 1), i ∈ I, z ∈ Z}: the node (z, i, -1) will appear as emitter node for load Q_i inside some tour T(k), which means that load Q_i is first going transported from o_i to z by vehicle k, and next from z to d_i by some other vehicle k', k ≠ k'. It comes that node (z, i, 1) will appear in a symmetric way in tour T(k').
- for any node z in Z, we set: $\mathcal{Node}(z, i, -1) = \mathcal{Node}(z, i, +1) = z$;
- for any node x in X we set: $\mathcal{Node}(x) = x$.

We extend $\mathcal{Status}$, $\mathcal{Twin}$, $\mathcal{Dem}$, $\mathcal{CH}$, Δ, and $\mathcal{F}$ by setting, for every z ∈Z, i ∈ I:

- $\mathcal{Status}(z, i, -1)$ = *Out-Reload*, $\mathcal{Status}(z, i, +1)$ = *In-Reload*;
- $\mathcal{Twin}(z, i, +1) = \{(z, i, -1)$, $\mathcal{Twin}(z, i, -1) = \{(z, i, +1)$;

- $\mathcal{D}em(z, i, +1) = \mathcal{D}em(z, i, -1) = i$;
- $\mathcal{CH}(z, i, -1) = -Q_i$, $\mathcal{N}ode(z, i, +1) = Q_i$;
- $\Delta(z, i, -1) = \Delta(z, i, +1) = +\infty$; $\mathcal{F}(z, i, -1) = \mathcal{F}(z, i, +1) = [0, +\infty[$.

Clearly, the function $\mathcal{D}ist$ may be also extended in a canonical way to Z*.Z*.

Tours, Valid Tours, Tour Family and Covering Tour Family.

A tour, in the sense of the *DARPT*, becomes a sequence Γ of nodes of Z*, which is such that: $\mathcal{S}tatus(\text{Start}(\Gamma)) = DepotD$, $\mathcal{S}tatus(\text{End}(\Gamma)) = DepotA$, $\mathcal{VI}(\text{Start}(\Gamma)) = \mathcal{VI}(\text{End}(\Gamma))$, for any node x in Γ: $x \neq \text{Start}(\Gamma)$, $x \neq \text{End}(\Gamma)$, $\mathcal{S}tatus(x) \notin Depot$, no node $x \in Z^*$ appears twice in Γ, and for any demand $i \in I$, one of the configuration below occurs, which excludes the others:

- o_i and d_i appear in Γ, $o_i <<_\Gamma d_i$, and no node (z, i, ε), $\varepsilon \in \{-1, 1\}$ is in Γ;
- none among o_i, d_i, (z, i, ε), $\varepsilon \in \{-1, 1\}$, appears in Γ;
- o_i and $(z, i, -1)$ are in Γ, such that $o_i <<_\Gamma (z, i, -1)$, and none among d_i, $(z, i, + 1)$, is in Γ;
- d_i and $(z, i, +1)$ are in Γ, such that $(z, i, +1) <<_\Gamma d_i$, and none among o_i, $(z, i, - 1)$, is in Γ;

As stated in the DARP sections, we say that a tour Γ is *load-valid iff*: for any x in Γ, $x \neq \text{Start}(\Gamma)$, we have $\Sigma_{y \backslash y <<_\Gamma x} \mathcal{CH}(y) \leq \mathcal{CAP}$ and such a tour Γ is *time-valid iff* it is possible to associate, with any node x in Γ, a time value $\delta(x) \geq 0$, in such a way that:

- for any x in Γ, $x \neq \text{Last}(\Gamma)$, $\mathbf{\delta(Succ(\Gamma, x)) \geq \delta(x) + \mathcal{D}ist(x, Succ(\Gamma, x))}$;
- for any x in $\Gamma \setminus \text{Status}(x) \notin \{Out\ (In)\ \text{-}Reload\}$, $\mathbf{|\delta(\mathcal{T}win(x)) - \delta(x)| \leq \Delta(z)}$;
- for any x in Γ, $\delta(x) \in \mathcal{F}(x)$.

In case δ exists, it is called a *valid time value set* related to Γ. In case the tour Γ is both *time-valid* and *load-valid*, we say that it is ***valid***. Clearly, a feasible solution of our *Dial a Ride with transfers* problem cannot be defined as a family $T = \{T(k), k = 1..K\}$ of pairwise disjoint valid tours. We must link tours which involve *Out-Reload* and *In-reload* nodes related to a same demand. In order to do it, we consider some subset J of the *Demand Index* set I, some tour collection $T = \{T(1)..T(K)\}$, and we say that T defines a ***covering collection*** for J if the tours T(1)..T(K) are pair-wise disjoint valid tours, their union contains the whole set $\{o_i, d_i, i \in J\}$ (they contain no node x such that $\mathcal{D}em(x) \in I - J$), and every tour T(k), k = 1..K, starts with the node *DepotD*(k) and ends with the node *DepotA*(k). The nodes (z, i, ε), $z \in Z$, $i \in J$, $\varepsilon \in \{-1, 1\}$ which appear in $\cup_{k = 1..K} T(k)$ define the *active relay node set* of the tour collection T. We denote it by ACT(T). The tour collection $T = \{T(k), k = 1..K\}$ is a *valid covering collection* for J *iff* :

- for every k = 1..K, the tour T(k) is *load-valid*;
- the collection $T = \{T(k), k = 1..K\}$ is a *covering collection* for J;
- there exists some time value set $t = \{t(x), x \in \cup_{k = 1..K} T(k)\}$ such that: for any k = 1..K, the restriction of t to the nodes of T(k) defines a valid time value set related to T(k); and, for every active *Out-Reload* node x in ACT(T), we have $t(\mathcal{T}win(x)) \geq t(x)$;

In case such a time value set t exists, we say that T is *time-valid* and t is called a *valid time value set* related to T.

5.2 Handling the relay nodes: an insertion mechanism

Once again, we want to deal with the above model by successively inserting the demands $\mathcal{D}_i$, $i \in I$, into a tour collection T, until this tour collection defines a *valid covering collection* for I. In order to do it, we first need to explain which kind of insertion mechanisms we intend to use. We are going to use two insertion operators: INSERT and INSERT2.

The **INSERT** operator works as in section 4.1: k being some vehicle index, x, y being two nodes in T(k) such that $x <<^{=}_{T(k)} y$, i being some demand index which is such that neither o_i nor d_i is in T(k), INSERT(T(k), x, y, i) denotes the tour which is obtained through insertion of o_i between x and Succ(T(k), x) in T(k) and by insertion of d_i between y and Succ(T(k), y) in T(k).

The **INSERT2** operator works by inserting demand $\mathcal{D}_i$ into two distinct tours: k, k' being two distinct vehicle indices, x, y being two nodes in T(k) such that $x <<^{=}_{T(k)} y$, x', y' being two nodes in T(k') such that $x' <<^{=}_{T(k')} y'$, z being some relay node in Z, INSERT2(i, k, k', x, y, x', y', z) denotes a pair of tours(INSERT2(i, k, k', x, y, x', y', z).First, INSERT2(i, k, k', x, y, x', y', z).Second) in such a way that:

- INSERT2(i, k, k', x, y, x', y', z).First is the tour which is obtained through insertion of o_i between x and Succ(T(k), x) in T(k) and by insertion of (z, i, -1) between y and Succ(T(k), y) in T(k);
- INSERT2(i, k, k', x, y, x', y', z).Second is the tour which is obtained through insertion of (z, i, 1) between x' and Succ(T(k'), x') in T(k') and by insertion of d_i between y' and Succ(T(k), y') in T(k').

5.3 A general insertion scheme

We notice that the two operators which we described above have quite different impacts on the way a global insertion schema is going to work. While testing the feasibility of an application of the INSERT operator is a local task, which only involve dealing with the T(k) tour, testing the feasibility of the INSERT2 operator is likely to involve more than the T(k), T(k') tours. By the same way, handling the FREE(i) is going to become more complicated once we start introducing relay nodes and linking constraints. For this reason, we decompose the resolution process into two steps (1-2):

1. We only use the INSERT operator, while proceeding as in the INSERTION procedure of section 4.2. Since we would like to use transfers and the related INSERT2 operator in order to make the whole tour system more efficient, we perform this first step while using stronger transit bounds Δ_i, $i \in I$, meaning that the riding times of the demanders improves;
2. The first step is likely to yield rejected demands, because of the stronger transit. So, the second step deals with those rejected demands while only using the INSERT2 operator.

The whole process may be summarized as follows:

Dial-a-Ride with Transfers Insertion Algorithmic Scheme:
Continue ← true;
While Continue do

Compute transit bounds Δ^*_i, $i \in I$, such that for any $i \in I$, $\Delta^*_i \leq \Delta_i$; **(E7)**
1 - Apply the INSERTION(N_1, N_2) procedure of section 4.2, while replacing, for any $i \in I$, Δ_i by Δ^*_i;
Let *Reject* the resulting rejected demand index set, T the resulting valid covering collection associated with $I_1 = I - Reject$, and t the related valid time value set;
2 -: we keep on replacing, for any $i \in I$, Δ_i by Δ^*_i;
$J \leftarrow Reject$; $Reject \leftarrow$ Nil; Initialize the sets FREE2(i), $i \in J$;
While $J \neq$ Nil do
Picks up some demand i_0 in J; Remove i_0 from J;
If FREE2(i_0) = Nil then $Reject \leftarrow Reject \cup \{i_0\}$;
Else
Derive from FREE2(i_0) a *Weak-L-Candidate* list of 7-uples; **(E8)**
Derive from *Weak-L-Candidate* a *L-Candidate* list, made with those 7-uples which are such that the replacement of ($T(k_1)$, $T(k_2)$) by INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2 , z) is going to maintain the validity of T; **(E9)**
If *L-Candidate* = Nil then $Reject \leftarrow Reject \cup \{i_0\}$
Else
Picks up (k_1, k_2, x_1, y_1, x_2, y_2, z) in *L-Candidate*;
Create two new active nodes related to (z, i_0, -1) and (z, i_0, 1);
($T(k_1)$, $T(k_2)$) $\leftarrow$INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2 , z); Update t;
Update the time windows $\mathcal{F}(x)$, $x \in \cup k = {}_{1..K}T(k)$; **(E10)**
Update the sets FREE2(i), $i \in J$; Insert i_0 into I_1;
Update Continue;
Keep the best result (T, t, *Reject*, $Perf_{A,B}(T, t)$) obtained during this process.

Update-Δ imposes stronger transit bounds, creating a need for transfers in order to obtained better $Perf_{A,B}(T, t)$ values, we specify **(E7)** with this procedure as follows:

Function Update-Δ. **Input**: (λ : Number in Q, $\lambda > 1$); **Output:** (Δ^*_i, $i \in I$);
For i in I do: If $\Delta_I > \lambda$. DIST(o_i, d_i) then $\Delta^*_I \leftarrow \lambda$ DIST(o_i, d_i) else $\Delta^*_i \leftarrow \Delta^*_i$;

5.4 The sets FREE2 and the construction of the *weak-L-Candidate* list

For any $i \in J$, the set FREE2-o(i) will be made with the pair (k, z) such that:
- the active node z is in T(k), different from Last(T(z)), [a(z), b(z)] denotes the time window $\mathcal{FP}(T)(z)$, z' gives Succ(T(k), z), and [α, β] denotes the time window $\mathcal{F}(o_i)$;
- $(a(z) + DIST(z, o_i) \leq \beta) \wedge (\alpha + DIST(o_i, z') \leq b(z')) \wedge (a(z) + DIST(z, o_i) + DIST(o_i, z') \leq b(z')) \wedge (\mathcal{C}(\Gamma, z) + Q_i \leq \mathcal{CAP})$.

In the same way, for any $i \in J$, the set FREE2-d(i) will be made with such that:
- $(a(z) + DIST(z, d_i) \leq \beta) \wedge (\alpha + DIST(d_i, z') \leq b(z')) \wedge (a(z) + DIST(z, d_i) + DIST(d_i, z') \leq b(z')) \wedge (\mathcal{C}(\Gamma, z) + Q_i \leq \mathcal{CAP})$.

So, for any (x_1, k_1) in FREE1-o(i_0), and for any (y_2, k_2) in FREE2-d(i_0), $k_1 \neq k_2$, we compute a relay node z through the following process summarized in 2 steps.

1 – First, let us recall that the function *Midst*, which, from any pair of nodes z, z' in the node set Z, computes a new node z" = *Midst*(z, z') in Z, in such a way that *Dist*(z, z") and *Dist*(z", z) are no larger than some fraction λ.*Dist*(z, z'), with λ< 1;
2 - For any node y in $T(k_1)$, we denote by *Close*(y, k_2)the first (in the sense of the relation $<<_{T(k2)}$) element x in $T(k_1)$ such that t(x) ≥ t(y). Then, we apply the following *Exchange* function below.

Function Exchange.
Input: (x_1, k_1, y_2, k_2); **Output:** (y in $T(k_1)$, x in $T(k_2)$, z : relay node);
Compute y in $T(k_1)$, such that:

- $x_1 <<_{T(k)}$ z, *Close*(y, k_2) $<<^{=}_{T(k2)}$ y_2, and DIST(y, *Close*(y, k_2) is the smallest possible ;

If y is undefined then ***Exchange*** ← *Undefined*
Else ***Exchange***← (y, Pred(*Close*(y, k_2)), *Midst*(z, U(z, l)));

Exchange(x_1, k_1, y_2, k_2) provides us with the parameters y_1, x_2, z, which would eventually allow us to perform the ($T(k_1)$, $T(k_2)$) ← INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2 , z) instruction. So, the *Weak-L-Candidate* list **(E8)** will be defined by those 7-uple (k_1, k_2, x_1, y_1, x_2, y_2, z) which we obtain this way, and the *L-Candidate* list will defined by those among those 7-uple which are such the validity of the tour collection T will be preserved through application of the INSERT2 operator.

5.5 Evaluate and testing the validity of an application of the INSERT2 operator

The process needs to check the validity of the tour collection T, in case we apply it some INSERT2 process. So, let us consider that we are provided with $I_1 \subset$ I, with a *valid covering collection* associated with I_1, with $i_0 \subset$ I - I_1, with a 7-uple (k_1, k_2, x_1, y_1,x_2, y_2, z) as above, and that we intend to perform INSERT2(k_1, k_2, x_1, y_1,x_2, y_2, z). Clearly, checking the load validity of the two resulting tours $T(k_1)$ and $T(k_2)$ can be easily done through application of the following procedure:

Procedure Test-Load2. **Input:** (i_0, k_1, k_2, x_1, y_1, x_2, y_2, z); **Output**: Boolean;
Test-Load2 ← {For any z in Segment($T(k_1)$, x_1, y_1), $C(\Gamma, z) + Q_{i0} \leq CAP$} ∧ {For any z in Segment($T(k_2)$, x_2, y_2), $C(\Gamma, z) + Q_{i0} \leq CAP$}

For testing the time validity, we need to take into account the whole collection T. Let us suppose that we just computed a copy Π of the tour collection which would result from the application of INSERT2. We also extend the DIST matrix in order to take into account the new nodes (z, i_0, -1) and (z, i_0, 1). We denote by *FS*(x), x ∈ ACT(Π) = $\cup_{k = 1..K}$ Π(k) the current time windows which appear in Π. We propagate the same 5 rules as in section 3.2 augmented with the following rule R_6 and R_7:

- Rule R_6: y = *Twin*(x); *Status*(x) = *Out-Reload*; *FS*.min(x) >*FS*.min(y) ⊨ *FS*.min(y) ←*FS*.min(x); NFact ← y;
- Rule R_7: y = *Twin*(x); *Status*(x) = *Out-Reload*; *FS*.max(x) >*FS*.max(y) ⊨ *FS*.max(x) ←*FS*.max(y); NFact ← x;

So, checking the time validity of Π, according to a current time window set $\mathcal{FS}$ = {$\mathcal{FS}$(x) = [$\mathcal{FS}$.min(x), $\mathcal{FS}$.max(x)], x ∈ ACT(Π) = $\cup_{k = 1..K}$Π(k)} is performed by application of a procedure *Propagate2* based on *Propagate*. We may consider, in case $\mathcal{Res}$ = 1, that the resulting time window set $\mathcal{FR}$ = {$\mathcal{FR}$(x), x ∈ ACT(Π)}, is completely determined by Π and by the original time window set $\mathcal{F}$. So, we denote it by $\mathcal{FP}$(Π), and we consider it as attached to any *time-valid* tour collection Π. Like in section 3, the tour collection Π is *time valid iff* the *Propogate2* function yields a positive $\mathcal{Res}$ signal.
As in Section 3, we need to evaluate the collection Π in case it is *time-valid* and compute some well-fitted related time value set δ. In order to do it, we only focus on the $\mathcal{Glob}$ component of the $\mathcal{Perf}$ criteria. We apply a *Bellman* process: we start by providing every *DepotD* node x with a maximal $\mathcal{FP}$(Π)(x).max time value, and next, we assign any other node x with a time value δ(x) which is the smallest possible, taking into account the initial assignments δ(*DepotD*(k)), k = 1..K, the current time windows $\mathcal{FP}$(Π)(x), x ∈ ACT(Π), the linking constraint and the distance constraints.

5.6 Attempting an insertion with transfer and building the $\mathcal{L}$-*Candidate* list.

So, let us suppose that we are provided with a non-inserted demand index i_0 and with some 7-uple (k_1, k_2, x_1, y_1, x_2, y_2, z). We want to try and, in case of success, to evaluate, an application of the process INSERT2. We perform thus the process Try-Insert2(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z), while proceeding step by step:
1. We perform a call *Test-Load2*(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z);
2. We check that DIST(o_{i0}, Succ(T(k_1), x_1) + Length(T(k_1), Succ(T(k), y_1) + $\mathcal{Dist}$(y_1, z) + $\mathcal{Dist}$(z, Succ(T(k_2), x_2) + Length(T(k_2), Succ(T(k_2), x_2), y_2) + DIST(y_2, d_{i0}) ≤ Δ_{i0};
3. We create two new nodes z-out = (z, i_0, -1) and z-in = (z, i_0, -1) in Z*, and we augment the DIST matrix in such a way it will provide with the respective distances between z -out and z-in and their respective neighbors in T(k_1) and T(k_2).
4. We perform a call INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z) on Π. The considered time windows are such that: $\mathcal{F}$(o_{i0}) if x = o_{i0}, $\mathcal{F}$(d_{i0}) if x = d_{i0} and [0, + ∞[if x = z-in or z-out;
5. We run the *Propagate2* procedure. In case of success, we run the *Evaluate2* procedure and get a resulting value $\mathcal{Val}$;
6. We restore the DIST matrix, the Π collection, and the ACT(Π) set.
At the end of this process, *Try-Insert2* provides us with $\mathcal{Res}$ and $\mathcal{Val}$. The Boolean $\mathcal{Res}$ expresses the feasibility of an application of an INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z) call and the number $\mathcal{Val}$ provides us, in case $\mathcal{Res}$ = *True*, with an evaluation of such a call. We are now able to summarize the whole resolution process of the *DARPT*.

5.7 The whole process

The global process consists in a "for" loop, during which the parameter λ progressively decreases from an initial value Λ until 1: the length P of this loop is a parameter of the main procedure. Any iteration inside this loop works as described in section 5.3. Δ values are updated as in **(E7)**. A *first step* involves a call

INSERTION(N_1, N_2) and yields some *Reject1* rejected demand index set, together with some pair (T, t), where T is a *valid covering collection* for I – *Reject1*, and t is a related time value set. In case *Reject1* is not empty, a second step is performed, which involves a call to a procedure INSERTION2, which works while trying to insert the rejected demands through applications of the INSERT2 operator. The INSERTION2 procedure takes as input the 3-uple (T, t, *Reject1*) which was computed through the *first step*, and proceeds, according to a "while" loop, in order to insert demands of *Reject1* through the INSERT2 operator. We denote by N-FREE-o(i) (N-FREE-d(i)) the number of vehicle which appear in FREE2-o(i) (FREE2-d(i)). The process is composed of four main steps:

1. It picks up some demand i_0 in J: if there exists i such that N-FREE-o(i) = 1 or N-FREE-d(i) = 1, then i_0 is chosen in a random way among those demands in J which is such that N-FREE-o(i) = 1 or N-FREE-d(i) = 1. Otherwise, it chooses i_0 randomly among the demands which minimize the sum of the two quantities. **(E11)**
2. It builds the *Weak-L-Candidate* list as in section 8.4, with those 7-uple (k_1, k_2, x_1, y_1, x_2, y_2, z) which are such that (k_1, x_1) ∈ FREE2-o(i_0), (k_2, x_2) ∈ FREE2-d(i_0), and (y_1, x_2, z) = *Exchange*(x_1, k_1, y_2, k_2); **(E12)**
3. For any 7-uple (k_1, k_2, x_1, y_1,x_2, y_2, z) in *Weak-L-Candidate*, it tries the *Try-Insert2* process, and, in case the *Res* component of the result is true, it inserts the 7-uple into a *L-Candidate* list, ordered according to the related *Val* component values; **(E13)**
4. In case *L-Candidate* is empty, then i_0 is inserted into *Reject*, else:
   - The process picks up (k_1, k_2, x_1, y_1, x_2, y_2, z) in *L-Candidate*: it proceeds through a random choice among the up to N_3 elements of *L-Candidate*. N_3 becomes a parameter of INSERTION2; **(E14)**
   - It activates the nodes (z, i_0, -1) and (z, i_0,1) and it effectively performs the insertion: (T(k_1), T(k_2)) ← INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z);
   - It updates t, applies the *Propagate2* and *Evaluate2* procedures and updates the sets FREE2(i), i ∈ J;

Finally, the process keeps the best result (T, t, *Reject*, $Perf_{A,B}$(T, t)) which was ever obtained during this process. So the INSERTION2 and the DARPT-INSERTION procedure come may be described as follows:

Procedure INSERTION2
Input: T1: partial tour collection, t1: time value set, *Rej*: Rejected Demand set);
Output: (T, t, *Perf*, *Reject*: rejected demand set, N_3);
J ←*Rej*; *Reject*← Nil; T ← T1; t ← t1;
While J ≠ Nil do
    Pick up some demand i_0 in J as in **(E11)**; Remove i_0 from J;
    If FREE2-o(i_0) = Nil or FREE2-d(i_0) = Nil then *Reject* ← *Reject* ∪ {i_0}
    Else
        Compute the *Weak-L-Candidate* list according to **(E12)**;
        Build the *L-Candidate* list according to **(E13)**;
        If *L-Candidate* = Nil then Reject ← Reject ∪ {i_0}
        Else
            Picks up (k_1, k_2, x_1, y_1, x_2, y_2, z) in *L-Candidate* according to **(E14)**;

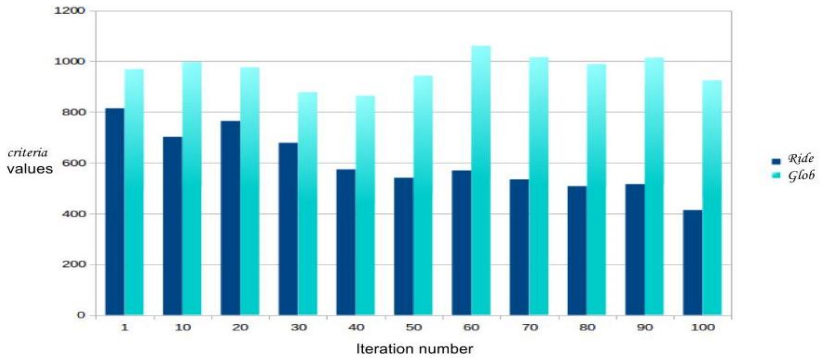
Create two new active nodes related to (z, i_0, -1) and (z, i_0, 1);
($T(k_1)$, $T(k_2)$) ← INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z);
Update t and $\mathcal{PF}$(T) through application of *Propagate2* and *Evaluate2*;
For any i ∈ J, update FREE2-o(i) and FREE2-d(i);
$\mathcal{Perf}$ ← $\mathcal{Perf}$(T, t);
INSERTION2← (T, t, $\mathcal{Perf}$, Reject);

DARPT-INSERTION
Input: (N_1, N_2, N_3, P, G, $\mathcal{VH}$, $\mathcal{D}$, (X, DIST, K, $\mathcal{CAP}$), A, B, Z, $\mathcal{Dist}$);
Output: (T, t, $\mathcal{Perf}$: performance value, $\mathcal{Reject}$: Rejected Demand Index set);
Δ-Aux ←Δ; Initialize λ with a large value Λ;
For p = 1..P do
Δ←*Update-Δ* (λ, Δ); (T1, t1, $\mathcal{Perf1}$, *Reject1*) ←INSERTION1(N_1, N_2);
If $\mathcal{Reject1}$ = Nil then *DARPT-INSERTION* ← (T, t, $\mathcal{Perf}$, $\mathcal{Reject}$);
Else (T, t, $\mathcal{Perf}$, $\mathcal{Reject}$) ← INSERTION2(T1, t1, $\mathcal{Reject1}$, N_3);
Δ ← Δ-Aux; λ←λ – 1/P.(Λ – 1);
Keep the best result (T, t, $\mathcal{Reject}$, $\mathcal{Perf}_{A,B}$(T, t)) obtained during this process.

6 Computational experiments

All the insertion techniques presented below were implemented in C++ and each replication was run successively on the same core of an Intel Q8300 at 2.5 GHz. For the first short experimentation on the DARP with transfers, the **DARPT-INSERTION** algorithm is applied to the first instance of Cordeau et al. (2003). We solve the R1a instance in order to analyse the evolution of the $\mathcal{Ride}$ time caused by the variation of the maximum ride time Δ. Figure 1 reports the $\mathcal{Ride}$ and the $\mathcal{Glob}$ times (in minutes) where all the demands have been included in valid routes. These times are shorter and shorter (up to half of the first time) during the program execution. The $\mathcal{Glob}$ times increases a little but not it is comparable to the $\mathcal{Ride}$ time drop. In a real context, i.e. a reactive context, depending on the time the system needs to accept (or not accept) the demand, this QoS criterion could be managed by the DARPT-INSERTION's value p.

Figure 1 – Ride and Global times during 100 iterations (P = 100)



For the second set of experimentations on the DARPT, we applied our solution to solve the DARPT on randomly generated instances. Each instance is different according the size of the time windows, the number of demands, and the number of cars.

Like in Cordeau et al. (2003), we randomly generated the coordinates of pick-up and drop-off nodes in the square of side 20. We split the square into 4 parts and the fleet $\mathcal{VH}$ into 4 sub-fleets $\mathcal{VH1}$, $\mathcal{VH2}$, $\mathcal{VH3}$, and $\mathcal{VH4}$ related to 4 sub-squares $\mathcal{EP1}$, $\mathcal{EP2}$, $\mathcal{EP3}$, and $\mathcal{EP4}$. $\mathcal{D}$ is classified in two sets: the transverse demands, which have its origin node in a different sub-square than the destination, and the local demands.

For each instance studied here, 50% of the demands are local and uniformly set to the 4 sub-squares. We generated a different maximum user ride time which equals the product of 20 and the distance between the origin node and the destination node. The capacity $\mathcal{CAP}$ equals 6 for each vehicle. Each demand has a large time window (all the day, from 0 to 1440 minutes) and a tight time window (15 or 30 minutes), their $\mathcal{Status}$ is granted randomly.

We solve 12 sets of 5 instances generated by the parameters written above. Table 1 gives the results obtained after 100 replications of our algorithm. We provided $R1_c$ which is the rate of the demand inserted in the routes when the transfers are forbidden. $R1_t$ is the same rate when transfers are allowed. We compute the value *Gap* such as $Gap \leftarrow (R1_t - R1_c)/(R1_c/100)$.

When comparing average rates obtained by each algorithm, about 11.9% of demand can be inserted when the transfers are allowed. Without surprise, $R1_t$ and $R1_c$ are better when the fleet has more cars (K=5). The gap is more important when there is a higher number of ways to do a transshipment. The instances {5, 6, 7, 8} clearly have a better insertion rate than the first and third sets (resp. with |D|=32 and |D|=96), with the same fleet.

When |D| = 96, the *Gap* values are smaller than for the other instances. According to the criterion $\mathcal{Glob}$, it is clear that the algorithm will begin to insert the local demands. Indeed, the vehicle will first choose the demands which imply a short rise of the $\mathcal{Glob}$ value. More local demands a vehicle must integrate results in a lower the capacity to accept transverse demands and, by extension, fewer transshipments.

Table 1. DARP classic Vs DARP with transfers

Inst.	|D|	K	Win. Size	RI_c	RI_t	*Gap* (%)	**Inst.**	|D|	K	Win. Size	RI_c	RI_t	*Gap* (%)
1	32	4	15	54.59	62.76	14.95	**7**	64	5	15	39.06	46.12	18.08
2	32	4	30	61.84	68.32	10.47	**8**	64	5	30	45.17	48.20	6.72
3	32	5	15	70.28	86.02	22.40	**9**	96	4	15	22.43	24.16	7.74
4	32	5	30	77.00	89.07	15.67	**10**	96	4	30	25.92	27.02	4.24
5	64	4	15	29.37	34.34	16.94	**11**	96	5	15	28.64	32.62	13.93
6	64	4	30	35.29	37.05	4.97	**12**	96	5	30	33.26	35.35	6.30

References

M. Carey and A. Kwieciński, 1995. Properties of expected costs and performance measures in stochastic models of scheduled transport. European Journal of Operational Research, 83, 182–199.

Cordeau, J.-F., Laporte, G. (2003). A Tabu Search heuristic algorithm for the static multi-vehicle dial-a-ride problem, Transportation Research B, 37, 579–594.

Cortes, C. E., Matamala, M., Contardo, C. (2010). The pickup and delivery problem with transfers, European Journal of Operational Research, 200, 711-724.

Deleplanque, S., Quilliot, A. (2013). Constraint propagation for the Dial-a-Ride Problem with Split Loads. Studies in Computational Intelligence, 470, Springer, 31-50.

Healy, P., Moll, R. (1995). A new extension of local search applied to the dial-a-ride problem, European Journal of Operational Research, 83, 83–104.

Kerivin, H., Lacroix, M., Mahjoub, A. R., Quilliot, A. (2008). The splittable pickup and delivery problem with reloads, European Journal of Industrial Engineering, 2, 112-133.

Laporte, G., Cordeau, J.F. (2007). The dial-a-ride problem: models and algorithms. Annals of Operations Research, 153, 29-46.

Madsen, O., Ravn, H., Rygaard, J. (1995). A heuristic algorithm for the DARP with time windows, multiple capacities, and multiple objectives, Annals of OR 60, 193–208.

Masson, R., Lehuédé, F, Péton, O. (2014), The Dial-A-Ride Problem with Transfers, Computers & Operations Research, 41, 12-23.

Masson, R., Lehuédé, F., Péton, O. (2013). An adaptive large neighborhood search for the pickup and delivery problem with transfers. Transportation Science, 47(3), 344-355.

Nakao, Y., Nagamochi, H. (2012). Worst case analysis for pickup and delivery problems with transfer, Communications and Computer Sciences, E91-A (9), 2328-2334.

Parragh, S.N., Doerner, K.F., Hartl, R.F. (2010). Variable neighborhood search for the dial-a-ride problem, Computers & Operations Research, 37, 1129–1138.

Psaraftis, H. (1983). An exact algorithm for the single vehicle many-to-many dial-a-ride problem with time windows. Transportation Science 17, 351–357.

Psaraftis, H., Wilson, N., Jaw, J., Odoni, A. (1986). A heuristic algorithm for the multi-vehicle many-to-many advance request DARP. Transportation Research B, 20B, 243-257.

Shang, J. S., Cu, C. K. (1996). Multicriteria pickup and delivery problem with transfer opportunity. Computers & Industrial Engineering, 30(4), 631-645.

Thangiah, S., Fergany, A., Awam, S. (2007). Real-time split-delivery PDPTW with transfers, Central European Journal of Operations Research, 15, 329-349.