Transfers in the on-demand transportation: the DARPT Dial-a-Ride Problem with transfers allowed

Samuel Deleplanque, Alain Quilliot

Transfers in the on-demand transportation: the DARPT

Dial-a-Ride Problem with transfers allowed

Samuel Deleplanque • Alain Quilliot

Abstract Today, the on-demand transportation is used for elderly and disabled people for short distances. Each user provides a specific demand: a particular ride from an origin to a destination with hard time constraints like time windows, maximum user ride time, maximum route duration limits and precedence. This paper deals with the resolution of these problems (Dial-a-Ride Problems – DARP), including the possibility of one transshipment from a transfer point by request. We propose an algorithm based on insertion techniques and constraints propagation.

1 Introduction

An important Operations Research model for the management of flexible reactive transportation system is the Dial-a-Ride Problems. In the DARP, people (or a combination of people and goods) can order a ride in which they define mobility demands, give a pick-up location, a delivery location, two time windows, an upper bound on the duration of the demand, and the load related to the demand. The majority of these problems uses are related to the elderly and disabled people, but the latest research in transportation (connected cars, autonomous cars, etc.) associated with technological advances (mobile communication, geo-localization…) could provide new services for the optimization problem. DARP are also related to a shared service transportation because several users could be in one vehicle at the same time. This type of problem is complex because it incorporates hard time constraints like time windows. The optimization consists of creating the route of a fleet of vehicles in order to satisfy all (or the most possible) the mobility demands emanating from people. To determine these routes, one has to find a balance between two opposite things: the quality of service, and minimization of the total cost. This work integrates the possibility to make transshipment between two vehicles in order to satisfy a request. This transshipment is done by a dynamic transfer point, meaning this point is computed at the same time that the demand is included in a vehicle planning.

Samuel Deleplanque
LIMOS, UMR CNRS 6158, Bat. ISIMA, BLAISE PASCAL University, France
E-mail: deleplan@isima.fr

Alain Quilliot
LIMOS, UMR CNRS 6158, Bat. ISIMA, BLAISE PASCAL University, France
E-mail: quilliot@isima.fr

DARP can be modeled in different ways. There exists a number of integer linear programmings [8], but the problem complexity is too high to use, most of which are NP-Hard because it also generalizes the Traveling Salesman Problem with Time Windows (TSPTW) [2] Therefore, the problem must be handled through heuristic techniques. [2] is one of the most important works on the subject and uses the Tabu search to solve it. Other techniques work well like dynamic programming (e.g. [14] and [1]) or variable neighborhood searches (VNS) (e.g. [13] and [6]). Moreover, a basic feature of DARP is that it usually derives from a dynamic context. So, algorithms for static DARP should be designed in order to take into account the fact that they will have to be adapted to dynamic and reactive context, which means synchronization mechanisms, interactions between the users and the vehicles, and uncertainty about fore coming demands. [15], and [9] later, developed the most used technique in dynamic context or in a real exploitation is heuristics based on insertion techniques. These techniques are a good solution when the people's requests have to take into account in a short time.

In this paper, we consider a generic DARP model with time windows and a mix QoS/Economical-Cost performance criterion, and propose algorithms for this model which are based upon randomized insertion techniques and constraint propagation, and so, which will easily adapt themselves to dynamic contexts, where demand package has to be inserted into (or eventually removed from) current vehicle schedules, in a very short time, while taking into account some probabilistic knowledge about fore coming demand packages. But, the main contribution of the paper is to allow transfer in the DARP. These transshipments are made dynamically and can be located everywhere. Little has been published on this subject, only [10]-[11] studied the problem. They express the problem by DARPT. The location of their transshipments points are known before the resolution. The authors use Tabu research, minimizing the total distance traveled by the fleet of vehicle. The closest problem is the Pickup and Delivery Problem with Transfers (PDPT).

There exist some exact methods to solve the PDPT, like [7], where the transfer points are those shaped by origin and destination nodes. [3] and [12] used a Branch-and-cut algorithm. As stated previously, the exact methods are not a good solution for solving the problem because it can't be used in a reactive context. The approximate methods are able to solve the problem in time. [4] created several rules for selecting the vehicle, assuming a demand given. Other rules help to trace the routes. The VNS is also used for this problem, [16] tested it on big instances (almost 200 requests). [17] worked on a heuristic where each pick-up and delivery nodes could be a transfer point. Their solution is based on insertion techniques and they allow the transshipments if the demand could not be inserted; [18] included the same algorithm in a dynamic context.

This paper is organized in the following manner: The next section will propose a model of the classic DARP. Then, we will explain how to handle the temporal constraints with a heuristic solution based on insertion techniques using propagation constraints. After, we will continue with the introduction of the DARPT model, framework and our updated solution based on insertion and propagation techniques. In the last part of the paper, the computational results will show the efficiency of our heuristics and we will compare the two solutions on the same instances.

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 $\mathbf{G} = (V, E)$, which contains at least some specific node *Depot*, and a *Demand* set $\mathcal{D} = (\mathcal{D}_i, i \in I)$, any demand $\mathcal{D}_i$ being defined as a 6-uple $\mathcal{D}_i = (o_i, d_i, \Delta_i, \mathcal{H}(o_i), \mathcal{H}(d_i), Q_i)$, where $\mathbf{o_i} \in V$ 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 bound (*transit bound*) on the duration of demand $\mathcal{D}_i$'s processing, $\mathcal{F}(o_i)$ is a time window related to the time $\mathcal{D}_i$ starts being processed, $\mathcal{F}(d_i)$ is a time window related to the time $\mathcal{D}_i$ ends being processed, and $\mathbf{Q_i}$ 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 $\mathcal{VH}$, and of the 6-uples $\mathcal{D}_i = (o_i, d_i, \Delta_i, \mathcal{F}(o_i), \mathcal{F}(d_i), Q_i)$, and while optimizing some performance criteria which 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 according to several ways, depending on the structure of the fleet $\mathcal{VH}$ and on the way this fleet is allowed to answer various demands of $\mathcal{D}$.

The fleet $\mathcal{VH}$ may be heterogeneous, or, conversely, homogenous. In the first case, part of the problem consists in affecting the demands to the different classes of vehicles. The loads Q_i, $i \in I$, may be described as a set of objects, all endowed with their own characteristics (weight, volume, autonomous mobility…), or, conversely, summarized by a number (we say that they are *nominal*), which identifies any demand as a volume or as a weight of identical objects which are required to be transported from node o_i to node d_i. In the first case, part of the problem consists in identifying which combinations of objects may be simultaneously transported by a given vehicle. Also, in such a case, one may have to take into account the fact that loading and unloading processes are not neutral, and that their duration is likely to depend on the current load of the involved vehicles.

Load pre-emption (split loads) may be allowed, which means that a given load Q_i, $i \in I$, may be split into several blocks, every one being transported from node o_i to node d_i while using distinct vehicles, or while using twice a same vehicle. *Vehicle pre-emption* (**transfers**/transhipments) may be allowed too, which means that a given load Q_i, $i \in I$, may be transported in several steps, each step involving a specific vehicle: for instance, if we think into Q_i as into a single traveller, it may try to go from node o_i to node d_i while first using a shuttle and next a bus.

Temporal constraints related to the time windows $\mathcal{F}(o_i)$, $\mathcal{F}(d_i)$, $i \in I$, and to the transit bounds Δ_i, $i \in I$, may be more and less tight. The problem may have to handled according to a *dynamic* context, (demands are not known in advance in an accurate way and must be processed "on line"): in such a case, one must take into account the way the system is supervised and the way its various components communicate with the users. Conversely, it may be set in a static context: all data are known in advance; the planning is computed and is run by the system. In this case, eventual divergences between the data which were used during the planning phases, and the situation the system has to really face, put what is called *robustness* at stake.

Finally, additional constraints may have to be tackled, 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 with **nominal demands**, 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 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 note z $<<_\Gamma$ z' if z is located before z' in Γ; z $<<_\Gamma^{=}$ z' if z $<<_\Gamma$ z' or z = z'.
We consider here that **no pre-emption is allowed** (i.e. no **transfers** and no split loads), and that **no additional cumulative constraint** has to be taken into account. In such a case, it is known that we do not need to consider the whole transit network G = (V, E), and that 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 ∈ 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 ∈ I}, makes correspond the shortest path distance from x to y in the transit network G.
As a matter of fact, we also split the *Depot* node according to its arrival or departure status and to the various vehicles of the fleet $\mathcal{VH}$, and we 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} ∪ {o_i, d_i, i ∈ I} and the distance matrix DIST, whose meaning is that, 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 connect x to y in G: we suppose that DIST, satisfies the *triangle inequality*. Also, the following characteristics, which, to any node x in X, make correspond:

- its status $\mathcal{Status}$(x): *Origin*, *Destination*, *DepotA*, *Depot D*; we set *Depot* = *DepotD* ∪ *Depot A*;
- 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 ∈ X;
- its load $\mathcal{CH}$(x): if $\mathcal{Status}$(x) ∈ *Depot* then $\mathcal{CH}$(x) = 0; if $\mathcal{Status}$(x) = *Origin*, and if 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 ∈ 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 or d_i, then Δ(x) = Δ_i, and Δ(x) = Δ else, where Δ is an upper bound which is imposed on the duration of any vehicle tour.

According to this construction, we understand that the system works as follows: vehicle k ∈ {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)), after having taken in charge some subset $\mathcal{D}$(k) = {$\mathcal{D}_i$, i ∈ 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) ∈ $\mathcal{F}$(o_i), loaded the whole load Q_i, and kept it until it arrived in d_i at time t(d_i) ∈ $\mathcal{F}$(o_i) and unloaded Q_i, in such a way that t(d_i) - t(o_i) ≤ Δ_i. Clearly, solving the Standard Dial a Ride instance related to those data (X, DIST, K, CAP) will mean computing the subsets $\mathcal{D}$(k) = {$\mathcal{D}_i$, i ∈ I(k)}, the routes followed by the vehicles and the time values t(x), x ∈ X, in such a way that both economic performance and quality of service be the highest possible.

Remark on the Service Durations* and the *Waiting Times

Many authors include what they call *service durations* in their models. That means that they suppose that loading and unloading processes related to the various nodes of X require some time amount δ(x), (*service time*) and, so, that they distinguish, for any node x ∈ X, time values t(x) (beginning of the *service*) and t(x) + d(x) (end of the *service*). By the same way, some authors suppose that the vehicles are always running at their maximal speed, and make a difference between the time t*(x), x ∈ X, when some vehicle arrives in x, and the time t(x) when this vehicle starts servicing the related demand (loading or unloading process). We do not do it. Taking into account service times, which tends to augment the size of the variables of

the model and to make it more complex it, has really sense only if we suppose that the service times $\delta(x)$ depend on the current state (its current load) of the vehicle at the time the loading or unloading process has to be launched. Making explicitly appear waiting times $t(x) - t^*(x)$ is really useful if we make appear the speed profile as a component of the performance criteria. In case none of the situation holds, the knowledge of the routes of the vehicles and of the time value $t(x)$, $x \in X$, is enough to check the validity of a given solution and to evaluate its performance, and then it turns out that ensuring the compatibility of the model with data which involve service times and waiting times $t(x) - t^*(x)$, $x \in X$, is only a matter of adapting the times windows $\mathcal{F}(x)$, the transit bounds $\Delta(x)$, $x \in X$, and the distance matrix DIST.

Tours, Time-Valid Tours, Charge-Valid Tours, Valid Tour

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:

- $\mathcal{Status}(\text{First}(\Gamma)) = \mathit{DepotD}$; $= \mathcal{Status}(\text{End}(\Gamma)) = \mathit{DepotA}$;
- $\mathcal{VI}(\text{First}(\Gamma)) = \mathcal{VI}(\text{End}(\Gamma))$;
- For any node x in Γ, $x \neq \text{First}(\Gamma), \text{End}(\Gamma)$, $\mathcal{Status}(x) \notin \mathit{Depot}$;
- No node $x \in X$ appears twice in Γ;
- For any node $x = o_i$ (d_i) which appears in Γ, the node $\mathcal{Twin}(x)$ is also in Γ, and we have: $x <<_\Gamma \mathcal{Twin}(x)$ ($\mathcal{Twin}(x) <<_\Gamma x$).

This tour Γ is said to be ***load-valid*** *iff*: for any x in Γ, $x \neq \text{First}(\Gamma)$, we have $\mathbf{\Sigma}_{y \backslash y <<_\Gamma x} \mathcal{CH}(\mathbf{y}) \leq \mathcal{CAP}$. And, this tour Γ is said to be ***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, $x \neq \text{Last}(\Gamma)$, $\mathbf{t(Succ(\Gamma, x)) \geq t(x) + DIST(x, Succ(\Gamma, x))}$;
- for any x in Γ, $\mathbf{|t(\mathcal{Twin}(x)) - t(x)| \leq \Delta(x)}$ 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 $\mathcal{Glob}(\Gamma, t) = \mathbf{t(End(\Gamma)) - t(First(\Gamma))}$: this quantity denotes the global duration of the tour Γ and $\mathcal{Ride}(\Gamma, t) = \mathbf{\Sigma}_{x \backslash \mathcal{Status}(x) = \mathit{Origin}} \mathbf{|\mathcal{Twin}(x) - t(x)|}$: 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, B are two multi-criteria coefficients, we may define the performance criteria $\text{Cost}_{A, B}(\Gamma, t)$ as follows: $\mathbf{Cost_{A, B}(\Gamma, t) = A.\mathcal{Glob}(\Gamma, t) + B.\mathcal{Ride}(\Gamma, t)}$.

The Standard dial-a-ride model.

So, let us suppose that we deduced from the data G = (V, E), $\mathcal{VH}$ = (K, $\mathcal{CAP}$), $\mathcal{D} = (\mathcal{D}_i = (o_i, d_i, \Delta_i, \mathcal{F}(o_i), \mathcal{F}(d_i), Q_i), i \in I)$, 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 $\mathcal{Perf}_{A, B}(T, t) \ \Sigma_{k = 1..K} \text{Cost}_{A, B}(T(k), t)$ is the smallest possible.

3 Handling temporal constraints

Let Γ a tour. The algorithm which we are going to describe inside the next section 4 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 Γ will maintain the validity of Γ, and to get an evaluation of the quality of this insertion. We are first going to provide ourselves with a package of constraint handling tools.

3.1 Testing the load-validity and the Time-validity

Checking the load-validity on Γ is easy. In order to be able to test the impact of the insertion of some demand into the tour Γ on the charge-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, $\mathcal{C}(\Gamma, x) = \Sigma_{y \backslash y <<_\Gamma \text{ or } y = x} \mathcal{CH}(y)$. Then it comes that 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 ([5]):

- **Rule $\mathbf{R_1}$**: y = Succ(Γ, x); $\mathcal{FS}.\min(x) + \text{DIST}(x, y) > \mathcal{FS}.\min(y)$ |= $\mathcal{FS}.\min(y) <- \mathcal{FS}.\min(x) + \text{DIST}(x, y)$; NFact <- y;
- **Rule $\mathbf{R_2}$**: y = Succ(Γ, x); $\mathcal{FS}.\max(y) - \text{DIST}(x, y) < \mathcal{FS}.\max(x)$ |= $\mathcal{FS}.\max(x) <- \mathcal{FS}.\max(y) - \text{DIST}(x, y)$; NFact <- x;
- **Rule $\mathbf{R_3}$**: y = Twin(x); $x <<_\Gamma y$; $\mathcal{FS}.\min(x) < \mathcal{FS}.\min(y) - \Delta(x)$ |= $\mathcal{FS}.\min(x) <- \mathcal{FS}.\min(y) - \Delta(x)$; NFact <- x;
- **Rule $\mathbf{R_4}$**: y = Twin(x); $x <<_\Gamma y$; $\mathcal{FS}.\max(y) > \mathcal{FS}.\max(x) + \Delta(x)$ |= $\mathcal{FS}.\max(y) <- \mathcal{FS}.\max(x) + \Delta(x)$; NFact <- y;
- **Rule $\mathbf{R_5}$**: $x \in \Gamma$; $\mathcal{FS}.\min(x) > \mathcal{FS}.\max(x)$ |= $\mathcal{Fail}$.

Procedure Propagate
Input : (Γ: Tour, L: List of nodes, $\mathcal{FS}$: Time windows set related to the node set of Γ):
Output : ($\mathcal{Res}$: Boolean, $\mathcal{FR}$: Time windows set related to node set of Γ);
Continue <- true;
While L ≠Nil and Continue do
z <- First(L); L <- Tail(L);
For i = 1..5 do 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 $\mathcal{Fail}$ 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, $t(x) \in \mathcal{FS}(x)$.

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), $x \in X(\Gamma)$}which is compatible with Γ and $\mathcal{FS}$. ***End-Proof.***

We denote by $\mathcal{FP}(\Gamma)$ the time window set which result from a call *Propagate*(Γ, Γ, $\mathcal{F}$),). $\mathcal{FP}(\Gamma)$ 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 $\mathcal{FP}(\Gamma)$. We want to get some fast estimation of the best possible value $Cost_{A, B}(\Gamma, t) = A.\mathcal{Glob}(\Gamma, t) + B.\mathcal{Ride}(\Gamma, t)$, $t \in \mathcal{Valid}(\Gamma)$. We already noticed that it could be done through linear programming or through general shortest path and circuit cancelling techniques. Still, since we want to perform this evaluation process in a fast way, we design two ad hoc procedures EVAL1 and EVAL2. The EVAL1 procedure works in a greedy way, by first assigning to the node First(Γ) its largest possible time value, and by next performing a **Bellman process** in order to assign to 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), $x \in \Gamma$. This 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)] = $\mathcal{FP}(\Gamma)$;
δ(First(Γ)) <- b(First(Γ)); x <- First(Γ);
While $x \neq$ Last(Γ) do
    y < Succ(Γ, x); δ(y) <- Sup(a(y), δ(x) + DIST(x, y)) ;
    x <- y ; δ <- {δ(x), $x \in \Gamma$}; Val <- $Cost_{A, B, C}(\Gamma, \delta)$;
EVAL1 <- (Val, δ);

Procedure EVAL2. Input:(Γ: Tour); **Output:** (Val: Number, δ: value set)
For any x in Γ, let us set set: [a(x), b(x)] = $\mathcal{FP}(\Gamma)$;
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)**: $(\lambda_x < 0) \wedge (\mathcal{Status}(x) \in \{Origin, DepotD\}) \wedge (\delta(x) \neq Inf(b(x), \delta(Succ(\Gamma, x) – DIST(x, Succ(\Gamma, x)))$;
  - **(E3)**: $(\lambda_x > 0) \wedge (\mathcal{Status}(x) \in \{Destination, DepotA\}) \wedge (\delta(x) \neq Sup(a(x), \delta(Pred(\Gamma, x) + DIST(Succ(\Gamma, x)), x))$;

  If $\mathcal{Fail}$(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 = $Cost_{A, B}(\Gamma, \delta)$), δ);

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

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

- **First Case**: there exists $x \in$ First(Γ) such that: δ(x) = a(x) and for any y such that $x <<_\Gamma^= y <<_\Gamma$ Last(Γ), we have: δ(Succ(Γ, y)) – δ(y) = DIST(y, Succ(Γ, y));
  Then the stability of $\mathcal{FP}(\Gamma)(x)$ 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)).
- **Second Case**: for any x in X(Γ), $x \neq$ 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 of the standard DARP

4.1 The insertion mechanism

It works in a very natural way. Let Γ be some valid tour, let $\mathcal{D}_i = (o_i, d_i, \Delta_i, \mathcal{F}(o_i), \mathcal{F}(d_i), Q_i)$ 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}, $\mathcal{FP}$(Γ)), 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)] = $\mathcal{FP}$(Γ)(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')) ∧ ($\mathcal{C}$(Γ, z) + Q ≤ $\mathcal{CAP}$);

The second test *Test-Node1* (based on *Test-Node*) aims at checking the feasibility of the insertion of an origin/destination node u, v, related to some load Q, between two consecutive node 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 ≠ 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-Charge*(Γ, x, y, i);
If Test = 1 then (Test, $\mathcal{F1}$) <- *Propagate*(Γ, {o_i, d_i}, $\mathcal{FP}$(Γ);
If Test = 1 then Val <- EVAL1(INSERT(Γ, x, y, i), $\mathcal{F1}$).Val else Val <- Undefined;
Test-Insert <- (Test, Val – Val1(Γ));

4.2 The insertion process

So, this process takes as input the demand set $\mathcal{D} = (\mathcal{D}_i = (o_i, d_i, \Delta_i, \mathcal{F}(o_i), \mathcal{F}(d_i), Q_i), i \in I)$, the 4-uple (X, DIST, K, $\mathcal{CAP}$) which we defined in section 2, and 2 multi-criteria coefficients A and $B \geq 0$, and it works in a greedy way through successive insertions of the various demands $\mathcal{D}_i = (o_i, d_i, \Delta_i, \mathcal{F}(o_i), \mathcal{F}(d_i), Q_i)$ 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, we make in such a way that, at 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 into some tour T(k), k = 1..K;
- current tours T(k), k = 1..K: for any such a tour T(k), we know the related time windows $\mathcal{FP}(T(k))(x)$, $x \in T(k)$, as well as the load values $\mathcal{C}(T(k), x)$, $x \in T(k)$, 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 ∈ $\mathcal{Q}$, 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) provides us with the number of vehicles which are still able to deal with demand $\mathcal{D}_i$.

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, that means which are such that N-FREE(i_0) and Card(Free(i_0)) are small. **(E4)**

2 – Then, in a second step, it picks up (k_0, x_0, y_0, v_0) in FREE(i_0) which simultaneously corresponds to one of the smallest values v, and to one of the smallest values EVAL2(INSERT(T(k), x, y, i_0)).Val – VAL2(T(k)): more specifically it first builds the list $\mathcal{L}$-*Candidate* of the N_1 (up to five) 4-uples (k, x, y, v) in FREE(i_0) with best (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 $\mathcal{L}$-*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 $\mathcal{L}$-*Candidate*. N_1 and N_2 become two parameters of the INSERTION procedure. **(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 $i \in J$, the set $\Lambda(i)$ 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 and N_2: Integer, *the demand set* $\mathcal{D} = (\mathcal{D}_i = (o_i, d_i, \Delta_i, \mathcal{F}(o_i), \mathcal{F}(d_i), Q_i), i \in I)$, *the 4-uple (X, DIST, K,* $\mathcal{CAP}$*) defined above, and 2 multi-criteria coefficients A and* $B \geq 0$);

Output: (T: tour set, t: time value set, $\mathcal{P}erf$: induced $\mathcal{P}erf_{A,B}(T, t)$ value, $\mathcal{R}eject$: 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 ; $\mathcal{R}eject$ <- 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 $\mathcal{R}eject$ <- $\mathcal{R}eject \cup \{i_0\}$
Else
Derive from FREE(i_0) the $\mathcal{L}$-*Candidate* list and Pick up (k_0, x_0, y_0, v_0) in $\mathcal{L}$-*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 $i \in J$ do
$\Lambda(i)$ <- {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 $\Lambda(i)$ 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;
$\mathcal{P}erf$ <- $\mathcal{P}erf_{A, B}$(T, t);
INSERTION <- (T, t, $\mathcal{P}erf$, $\mathcal{R}eject$);

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 |$\mathcal{R}eject$| is the smallest possible, and which is such that, among those pairs which minimize |$\mathcal{R}eject$|, it yields the best $\mathcal{P}erf_{A, B}$(T, t) value).

5 Dial a ride problem with transfers

We are going to deal now with the case when *transfers* is allowed, that means 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), may be handled in several successive steps, each step involving some vehicle $k \in K$, which make the charge Q_i go from some origin or relay node x to some relay or destination node y. Transfers means here that, at any time during the transportation, while the load Q_i is always handled as a whole, the route it follows may be split into several sub-route, all those sub-routes being taken in charge by distinct vehicles. As a matter of fact, we are going to restrict here ourselves to the case when no more than 2 vehicles are allowed to perform such a transportation task: though this restriction is not going to induce any true restriction on concepts and methods, it will help us in describing them; also, **practical applications are such that they will hardly allow a same demand to be handled by more than 2 or 3 different vehicles**.

5.1 The insertion mechanism

The basic point is here that, in case a given load Q_i has to be successively handled by two vehicles k and k', the set X, which arises from a construction process involving only origin/destination nodes, 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, which may be nodes inside a large scale transit network, are not known in advance: so, we feel that we should try to handle those exchange nodes in an implicit way and to deal with them in a dynamic way.

Extending the node set X into an implicit node set Z.
In order to put this in a formal way, we first need to extend X in order to make appear the relay nodes. Since we want to handle those relay node in a dynamic way, 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, Dist(x, x') = DIST(x, x'). Depending on the way Z is defined, this can be done in several ways: for instance, if nodes of X are related to some point in the 2-dimensional affine plane R^2, and if DIST represents the Euclidian distance, then Z may be the whole 2-dimensional plane R^2, and $\mathcal{Dist}$ may denote the Euclidian distance, which we are able to compute in an effective way once we know the 2D-coordinates of both points z and z'.

It comes that the input of the *Dial a Ride problem with transfers (**DARPT**)* is going to be defined, as in the standard version, by a transit network G = (V, E), a vehicle fleet $\mathcal{VH}$ = (K, $\mathcal{CAP}$), a demand set $\mathcal{D} = (\mathcal{D}_i = (o_i, d_i, \Delta_i, \mathcal{F}(o_i), \mathcal{F}(d_i), Q_i), i \in I)$, by a 4-uple (X, DIST, K, $\mathcal{CAP}$), and with 2 multi-criteria coefficients A and $B \geq 0$, augmented with some node set Z, which will derive from the transit network G and which will be such that $X \subset Z$, and by a function $\mathcal{Dist}$, with domain Z.Z, which will be such that: for any x, y in X, DIST(x, y) = $\mathcal{Dist}(x, y)$. Also, 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 $\lambda < 1$.

Building Relay Nodes: the Implicit Set Z*.

Additional nodes in Z – X are going to be used as relay nodes. Any such an active relay node will appear in 2 tours, once as an emitter node and once as a receiver node. Since we would like to keep on with the kind of model which we have been using for the standard version of the *Dial a Ride* problem, we also would like to make in such a way that all nodes which are going to appear in a tour family T be distinct. In order to do it, we define the implicit node set Z* as follows:

- $Z^* = X \cup \{(z, i, -1), (z, i, 1), i \in I, z \in Z\}$: the meaning of node (z, i, -1) is that if such a node becomes active, then it 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', $k \neq k'$. 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 the $\mathcal{Status}$, $\mathcal{Twin}$, $\mathcal{Dem}$, $\mathcal{CH}$, Δ, and $\mathcal{F}$ functions by setting, for every $z \in Z$, $i \in I$:

- $\mathcal{Status}(z, i, -1)$ = *Out-Reload*, $\mathcal{Status}(z, i, +1)$ = *In-Reload*;
- $\mathcal{Twin}(z, i, +1) = \{(z, i, -1)$, $\mathcal{Twin}(z, i, -1) = \{(z, i, +1)$;
- $\mathcal{Dem}(z, i, +1) = \mathcal{Dem}(z, i, -1) = i$;
- $\mathcal{CH}(z, i, -1) = -Q_i$, $\mathcal{Node}(z, i, +1) = Q_i$;
- $\Delta(z, i, -1) = \Delta(z, i, +1) = +\infty$;
- $\mathcal{F}(z, i, -1) = \mathcal{F}(z, i, +1) = [0, +\infty[$.

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

Tours, Valid Tours, Tour Family, Covering Tour Family, **Active Nodes**.

A tour, in the sense of the *DARPT*, becomes a sequence Γ of nodes of Z*, which is such that: $\mathcal{Status}(\text{Start}(\Gamma))$ = *DepotD*, $\mathcal{Status}(\text{End}(\Gamma))$ = *DepotA*, $\mathcal{VI}(\text{Start}(\Gamma)) = \mathcal{VI}(\text{End}(\Gamma))$, for any node x in Γ: $x \neq \text{Start}(\Gamma)$, $x \neq \text{End}(\Gamma)$, $\mathcal{Status}(x) \notin$ *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 classic DARP section, we say that: such a tour Γ is ***load-valid*** iff : for any x in Γ, x ≠ Start(Γ), we have $\Sigma_{y \backslash y <<_{\Gamma} x} \mathcal{H}(y) \leq \mathcal{CAP}$ and such a tour Γ is ***time-valid*** iff it is possible to associate, with any node x in Γ, some time value δ(x) ≥ 0, in such a way that:

- for any x in Γ, x ≠ Last(Γ), $\boldsymbol{\delta(Succ(\Gamma, x)) \geq \delta(x) + \mathcal{D}ist(x, Succ(\Gamma, x))}$;
- for any x in Γ \ Status(x) ∉ {*Out-Reload, In-Reload*}, $\boldsymbol{|\delta(\mathcal{T}win(x)) - \delta(x)| \leq \Delta(z)}$;
- for any x in Γ, δ(x) ∈ $\mathcal{F}$(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 charge-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 ∈ J} (they contain no node x such that $\mathcal{D}em$(x) ∈ 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 ∈ Z, i ∈ J, ε ∈ {-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 ∈ $\cup_{k = 1..K}$ 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)) ≥ 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.

The Dial a Ride Problem with Transfers model.
Input: *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), by a 4-uple (X, DIST, K, $\mathcal{CAP}$), and by 2 multi-criteria coefficients A and B ≥ 0, augmented with the node set Z, such that X ⊂ Z, and with the function $\mathcal{D}ist$, with domain Z.Z, such that: for any x, y in X, DIST(x, y) = $\mathcal{D}ist$(x, y).*
Output: *A valid covering collection T = {T(k), k = 1..K} for the Demand Index set I, a related valid time value set = {t(x), x ∈ ACT(T)}, such that the quantity $\mathcal{P}erf_{A, B}$(T, t) is the smallest possible.*

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 ∈ 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, $i \in I$, that means while making in such a way the riding times of the demanders get improved;
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.

That means that the whole process may be summarized as follows:

Dial-a-Ride with Transfers Insertion Algorithmic Scheme:
Continue <- true;
While Continue do
  Compute updated 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 <- *Reject*; *Reject* <- 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* <- *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* <- *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);
        ($T(k_1)$, $T(k_2)$) <- INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z);
        Update t; Update the time windows $\mathcal{H}(x)$, $x \in \cup\ k = {}_{1..K}\ T(k)$; **(E10)**
        Update the sets FREE2(i), $i \in J$; Insert i_0 into I_1;
  Update Continue;

Keep the best result (T, t, *Reject*, $Perf_{A, B, C}$(T, t)) which was obtained during this process.

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

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

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

For any i ∈ 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 *FP*(T)(z), z' gives Succ(T(k), z), and [α, β] denotes the time window *F*(o_i);
- (a(z) + DIST(z, o_i) ≤ β) ∧ (α + DIST(o_i, z') ≤ b(z')) ∧ (a(z) + DIST(z, o_i) + DIST(o_i, z') ≤ b(z')) ∧ (*C*(Γ, z) + Q_i ≤ *CAP*).

In a same way, for any i ∈ J, the set FREE2-d(i) will be made with the pair (k, z) such that:

- (a(z) + DIST(z, d_i) ≤ β) ∧ (α + DIST(d_i, z') ≤ b(z')) ∧ (a(z) + DIST(z, d_i) + DIST(d_i, z') ≤ b(z')) ∧ (*C*(Γ, z) + Q_i ≤ *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 we are provided with a function *Midst*, which, from any pair of nodes z, z' in the node set Z, compute 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.

Exchange function. **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)));

The result which is produced by ***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, ordered according to some auxiliary performance criteria.

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

What remains to be done is explaining the way we 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 trying 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), $\mathcal{C}(\Gamma, z) + Q_{i0} \leq \mathcal{CAP}$} $\wedge$ {For any z in Segment($T(k_2)$, x_2, y_2), $\mathcal{C}(\Gamma, z) + Q_{i0} \leq \mathcal{CAP}$}

As for the time validity, things become more difficult, since we need to take into account all the existing linking constraints, which means that the whole collection T is involved in the test. So, let us suppose that we just computed a copy Π of the tour collection which would result from the application of INSERT2(k_1, k_2, x_1, y_1, x_2, y_2, z), and that we just extend the DIST matrix in order to take into account the new active nodes (z, i_0, -1) and (z, i_0, 1). We may use the notation Succ and Pred for the whole collection Π as well as for a specific tour, since a given node in Z* is not going to appear more than once in Π. We denote by $\mathcal{FS}(x)$, $x \in ACT(\Pi) = \cup_{k = 1..K} \Pi(k)$ the current time windows related to the nodes x which appear in Π. What we do is to propagate the same 5 rules as in section 3.2 augmented with the following rule R_6 and R_7:

- Rule R_6: y = $\mathcal{Twin}(x)$; $\mathcal{Status}(x)$ = *Out-Reload*; $\mathcal{FS}$.min(x) > $\mathcal{FS}$.min(y) |= $\mathcal{FS}$.min(y) <- $\mathcal{FS}$.min(x); NFact <- y;
- Rule R_7: y = $\mathcal{Twin}(x)$; $\mathcal{Status}(x)$ = *Out-Reload*; $\mathcal{FS}$.max(x) > $\mathcal{FS}$.max(y) |= $\mathcal{FS}$.max(x) <- $\mathcal{FS}$.max(y); NFact <- x;

So, checking the time validity of Π, according to a current time window set $\mathcal{FS} = \{\mathcal{FS}(x) = [\mathcal{FS}.\min(x), \mathcal{FS}.\max(x)], x \in ACT(\Pi) = \cup_{k = 1..K} \Pi(k)\}$ 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 $\mathcal{FR} = \{\mathcal{FR}(x), x \in ACT(\Pi)\}$, is completely determined by Π and by the original time window set $\mathcal{F}$. So, we denote it by $\mathcal{FP}(\Pi)$, 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, and we apply a *Bellman* process: we start by providing every *DepotD* node x with a maximal $\mathcal{FP}(\Pi)(x)$.max time value, and next, we do in such a way we assign any other node x with a time value $\delta(x)$ which is the smallest possible, taking into account the initial assignments δ(*DepotD*(k), k = 1..K, the current time windows $\mathcal{FP}(\Pi)(x)$, $x \in ACT(\Pi)$, the linking constraint and the distance constraints. Clearly, procedures *Propagate2* and *Evaluate2*, which will be at the core of the construction of the $\mathcal{L\text{-}Candidate}$ list in **(E8)**, are also being involved in **(E10)**, when it will come to updating the time value set t and the current time windows $\mathcal{F}$. Once the "($T(k_1)$, $T(k_2)$)) <- INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2 , z);" instruction have been performed, time windows $\mathcal{FP}(T)$ will be computed, as well as t = Evaluate2(T).δ.

5.6 Attempting an insertion with transfer and building the 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 to (i_0, k_1, k_2, x_1, y_1, x_2, y_2, z). So, we perform the process **Try-Insert2**(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z), while proceeding step by step:

First step: we perform a call *Test-Load2*(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z);

Second step: we check that $DIST(o_{i0}, Succ(T(k_1), x_1) + Length(T(k_1), Succ(T(k), y_1) + \mathcal{Dist}(y_1, z) + \mathcal{Dist}(z, Succ(T(k_2), x_2) + Length(T(k_2), Succ(T(k_2), x_2), y_2) + DIST(y_2, d_{i0}) \leq \Delta_{i0}$;

Third step: we create temporarily two new active 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 resepctive neighbours in $T(k_1)$ and $T(k_2)$.

Fourth step: we perform a call INSERT2(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z) on Π; Time windows $\mathcal{FS}(x)$, in $ACT(\Pi)$ are the current time windows $\mathcal{FR}(T)(x)$, if x is in ACT(T), and they are: $\mathcal{F}(o_{i0})$ if $x = o_{i0}$, $\mathcal{F}(d_{i0})$ if $x = d_{i0}$ and $[0, +\infty[$ if x = z-in or z-out;

Fifth step: we run the *Propagate2* procedure. In case of success, we run the *Evaluate2* procedure and get a resulting value *Val*;
Sixth step: we restore the DIST matrix, the Π collection, and the ACT(Π) set.
At the end of this process, **Try-Insert2**(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z) provides us with a 6-uple (*Res*: Boolean, *Val*: Number). The Boolean *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 *Val* provides us, in case *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. Any time it enters this main "while" loop, it is provided with a subset J of the *Reject1* set, which contains the demands which remain to be inserted and a current *Reject* set, a pair (T, t), related time windows *FP*(T)(x), x ∈ ACT(T), and a related current active set ACT(T), and finally the sets FREE2-o(i) and FREE2-d(i), i ∈ J, defined as in VIII.4; 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 such that N-FREE-o(i) = 1 or N-FREE-d(i) = 1; else it is chosen in a random way among the demands in J which minimize the quantity N-FREE-o(i) = 1 + N-FREE-d(i); **(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*(i_0, k_1, k_2, x_1, y_1, x_2, y_2, z) process, and, in case, the *Res* component of the result is true, it inserts (k_1, k_2, x_1, y_1, x_2, y_2, z) into a *L-Candidate* list, ordered according to increasing 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 creates two new active nodes related to (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, together with the time windows *FP*(x), x ∈ ACT(T), through application of the *Propagate2* and *Evaluate2* procedures and the process 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(T1: partial tour collection, t1: related time value set, *Rej*: Rejected Demand set): (T: tour set, t: time value set, *Perf*: induced *Perf*(T, t) value, *Reject*: rejected demand set, N_3: integers);
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 *PH*(T) through application of the *Propagate2* and *Evaluate2* procedures;
For any i ∈ J, update FREE2-o(i) and FREE2-d(i);
Perf <- *Perf*(T, t);
INSERTION2 <- (T, t, Perf, Reject);

DARPT-INSERTION
Input: (N_1, N_2 **,** N_3, P: Integers, *A transit network G = (V, E), a vehicle fleet VH = (K, CAP), a demand set D = (D_i = (o_i, d_i, Δ_i, F(o_i), F(d_i), Q_i), i ∈ I), by a 4-uple (X, DIST, K, CAP), and by 2 multi-criteria coefficients A and B ≥ 0, augmented with the node set Z, such that X ⊂ Z, and with the function Dist, with domain Z.Z, such that: for any x, y in X, DIST(x, y) = Dist(x, y).*
Output: (T: Tour collection, t: time valued set, *Perf*: Performance Value, *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, *Perf1*, *Reject1*) <- INSERTION1(N_1, N_2);
If *Reject1* = Nil then *DARPT-INSERTION* <- (T, t, *Perf*, *Reject*)
Else (T, t, *Perf*, *Reject*) <- INSERTION2(T1, t1, *Reject1*, N_3);
Δ <- Δ-Aux ; Update λ: λ <- λ – 1/P.(Λ – 1);
Keep the best result (T, t, *Reject*, $Perf_{A, B}$(T, t)) which was obtained during this process.

6 Computational experiments

All the insertion techniques based on the constraint propagation were implemented in C++ and each replication was run on the same thread of an Intel Q8300 (2.5 GHz).

6.1 Experiments on the DARP

Our first experiments deal with the randomly generated instances of [2]. To analyse the behaviour of our solution, we used the same objective function used in [19] and adapted in [20]. The instances have between 24 and 144 requests which have to be supported by a fleet of 3 to 13 vehicles. The maximum route duration is 480 for each vehicle and for each instance. The capacity is equal to 6 and the maximum ride time is 90. [19] used the objective function given in equation (4), the terms penalizing the violations have been removed. Thus, we minimize travel distance (c(s)), excess ride time (r(s), cf. (1)), passenger waiting (l(s), cf. (2)),

the total duration $\mathcal{Glob}$ (g(s)) and early arrival (e(s), cf. (3)). We set the weight like in [19] and [20] to w_1=8, w_2=3, w_3=1, w_4=1, w_5= $|\mathcal{D}|$.

$$r(s) = \sum_{k=1}^{K} \sum_{i \in k} (Ride(i) - DIST(o_i, d_i)) \quad (1)$$

$$l(s) = \sum_{k=1}^{K} \sum_{x=succ(First(\Gamma_k))}^{pred(last(\Gamma_k))} Wait_{succ(x)}(\mathrm{C}(\Gamma_k, succ(x)) - q_x) \quad (2)$$

$$e(s) = \sum_{k=1}^{K} \sum_{x=First(\Gamma_k)}^{pred(pred(last(\Gamma_k)))} F.Min(succ(x)) - (\delta(\mathrm{x}) + \mathrm{DIST}(\mathrm{x}, \mathrm{succ}(\mathrm{s}))) \quad (3)$$

$$Cost = w_1 c(s) + w_2 r(s) + w_3 l(s) + w_4 g(s) + w_5 e(s) \quad (4)$$

The table 1 gives the values of the $\mathcal{COST}$ obtained with the proposed insertion techniques using constraint propagation. Gap is computed with the results of our insertion technique and the VNS such as *Gap<-100.(VNS-TI)/TI*. We take best results over 25.10^4 replications with a variation in the values of N_1 and N_2 (each lower than 4). We noted only the objective function of the two works. So we compare our *Insertion Techniques* (IT) with the *Variable Neighbourhood Search* (VNS) and the *Genetic Algorithm* (GA). Refer to [19] and [20] for the other values. As with the VNS technique, we obtained results always better than the GA. Moreover, we often obtained better results than the variable neighbourhood search. So we found a large difference between [19] and the others works, but solutions obtained by us and [20] are close even though in R10a we obtain a large gap. In fact, time constraints of this instance are very tight and we use a simple learning algorithm without computing a precise order for introducing the demands already rejected.

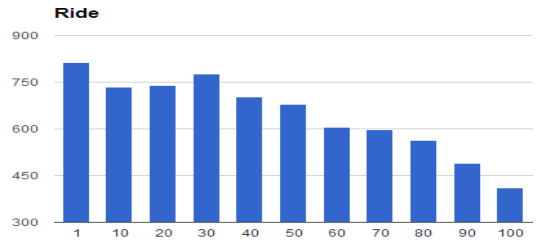
Early arrivals have the largest weight in the objective function and our solution gives us numbers close to 0 (except for R10a) for the majority of the results. In this case, no vehicle arrives at a node before the beginning of a node's time window. Each large gap obtained (negative or positive) is related to the early arrivals. This is also why the solution based on a genetic algorithm [19] resulted in a very large cost. Our CPU times are close (or lower) to the VNS' runs with the same number of iterations (e.g. for the 25.10^4 replications we required less than one minute for the smallest instance (R1a) and 38 minutes for the biggest and hardest instance (R10a).

Table 1 - Insertion techniques (IT) compared to GA ([19]) and VNS ([20])

Inst.	|D|	GA [19]	VNS [20]	IT	Gap
R1a	24	4696	3234.6	3371.4	-4.1
R2a	48	19426	14640.2	9025.7	62.2
R3a	72	65306	15969.1	10780.8	48.1
R5a	120	213420	23852.0	14054.2	69.7
R9a	108	333283	13806.4	14175.7	-2.6
R10a	144	740890	25016.5	35359.5	-29.3
R1b	24	4762	2825.5	2927.6	-3.5
R2b	48	13580	5003.1	5066.5	-1.3
R5b	120	98111	12360.5	12528.9	-1.3
R6b	144	185169	16499.4	16339.4	1.0
R7b	36	9169	4601.7	4523.1	1.7
R9b	108	167709	13412.8	13564.9	-1.1
R10b	144	474758	16420.0	17546.5	-6.4
Average		179252.2	12895.5	12251.1	5.3

6.1 Experiments on the DARPT

Figure 1 - Indirect minimization of the Ride times



For the first short experimentation on the DARP with transfers, the **DARPT-INSERTION** algorithm was applied to the first instance of [2], the R1a, in order to analyse the evolution of the *Ride* time caused by the variation of the maximum ride time Δ. We report in Figure 1 eleven *Ride* times, each with 10 replications, where all the demands have been included in *valid* routes. These times are shorter and shorter (up to half of the first time) during all the executions, the *Glob* times had a small raise but not comparable to the *Ride* time. In a real context, so in a reactive context, depending on the time the system needs to accept or not the demand (virtual insertion and synchronization mechanisms included), this QoS criterion could be managed by the DARPT-INSERTION's value p.

For the second experimentation on the DARPT, we applied our solution to solve the DARPT on a set of randomly generated instances. Each instance is different by the size of the windows, the number of demand, and the number of cars. Like in [2], we randomly generated the coordinates of pick-up and drop-off nodes in the square of side 20. We split the square in 4 parts and the fleet *VH* in 4 sub-fleets *VH1, VH2, VH3*, and *VH4* related to the sub-squares *EP1, EP2, EP3*, and *EP4*. *D* is classified in two sets: the transverse demands which are 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 equal to the product of 20 and the distance between the origin node and the destination node. The capacity CAP equals 6 for each vehicle. Each demand has a large time windows (all the day, from 0 to 1440 minutes) and another tight (15 or 30 minutes), their *Status* is grant randomly. We performed 100 replications of 12 sets of 5 instances generated by the parameters written above. Table 3 gives the results. 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 computed Gap such as $Gap <- (R1_t - R1_c)/(R1_c/100)$. All the results are average of the 100 replications. When comparing average rates obtained by each resolution, about 11.9% of demand can be inserted if the transfers are allowed in addition compared with no transfer. $R1_t$ and $R1_c$ are obviously better when the fleet has more cars (K=5), but if the gap is more important it means there are more possibility to do a transshipment. The first fourth instances have 3 times less demands inserted than the second set (with the same fleet). The R1t and R1c for the first set are a little less than 3 times less than the second set. That is explain by the fact the second set are a bigger choice to included his demands.

Table 2. 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

Acknowledgements We wish to thank you the *Conseil Regional d'Auvergne* and the FEDER of the European Union.

References

1. Chevrier, R., Canalda, P., Chatonnay, P., Josselin, D. (2006). Comparison of three algorithms for solving the convergent demand responsive transportation problem, ITSC'2006, 9th Int. IEEE Conf. on ITS, Toronto, Canada, 1096–1101.
2. 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.
3. Cortes, C. E., Matamala, M., Contardo, C. (2010). The pickup and delivery problem with transfers, European Journal of Operational Research 200 p711-724 .
4. Cortes, C. E., Jayakrishnan, R. (2002). Design and operational concepts of high-coverage point-to-point transit system, Transportation Research Record 1783 p178-187.
5. Deleplanque, S., Quilliot, A. (2012). Insertion techniques and constraint propagation for the DARP. FedCSIS. 393–400. IEEE conference publications.
6. 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.
7. Kerivin, H., Lacroix, M., Mahjoub, A. R., Quilliot, A. (2008). The splittable pickup and delivery problem with reloads, European Journal of Industrial Engineering 2 - p112-133.
8. Laporte, G., Cordeau, J.F. (2007). The dial-a-ride problem: models and algorithms. Annals of Operations Research.
9. 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.
10. Masson, R., Lehuédé, F., Péton, O. (2011). A tabu search algorithm for the Dial-a-Ride Problem with Transfers, Proceedings of the ICIESM.
11. Masson, R., Lehuédé, F., Péton, O. (2012). Simple Temporal Problems in Route Scheduling for the Dial–a–Ride Problem with Transfers, 2012, Lecture Notes in Computer Science, Volume 7298/2012, 275-291, DOI: 10.1007/978-3-642-29828-8_18.
12. Nakao, Y., Nagamochi, H. (2012). Worst case analysis for pickup and delivery problems with transfer, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E91-A (9) p2328-2334 .
13. 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.
14. 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.
15. 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.
16. Masson, R., Lehuede, F., Peton, O. (2012). An adaptive large neighborhood search for the pickup and delivery problem with transfers, Transportation Science in press.
17. Shang, J. S., Cu, C. K. (1996). Multicriteria pickup and delivery problem with transfer opportunity. Computers & Industrial Engineering.
18. Thangiah, S., Fergany, A., Awam, S. (2007). Real-time split-delivery PDPTW with transfers, Central European Journal of Operations Research - 15 329-349.
19. R.M. Jorgensen, J. Larsen, and K.B. Bergvinsdottir Solving the DARP using genetic algorithms. Journal of the Operational Research Society, 58(10):1321-1331, 2007.
20. S.N. Parragh, K.F. Doerner, R.F. Hartl. Variable neighborhood search for the dial-a-ride problem. Computers & Operations Research, 37 p. 1129–1138, 2010.