Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

# New online reinsertion approaches for a dynamic Dial-a-Ride Problem

S. Vallee, A. Oulamara [*], W. Ramdane Cherif-Khettaf

*Université de Lorraine, CNRS, LORIA, Campus Scientifique – BP 239, 54506 Vandoeuvre-les-Nancy, France*

ARTICLE INFO

*Keywords:*
Dynamic DARP
Reinsertion heuristics
Ejection chain
Computational experiment

ABSTRACT

This study is inspired by a challenging dynamic Dial-a-Ride Problem (DARP) encountered in a mobility service operated by the company *Padam.*[1] . Customers ask for a transportation service either in advance or in real-time and get an immediate answer about whether their requests are accepted or rejected. The main goal is to maximize the number of accepted requests during the service while respecting maximum capacity of limited number of vehicles. In this study, we propose novel methods based on reinsertion techniques to improve *Padam*'s online system. The proposed reinsertion techniques aim to exploit the neighborhood of a solution and must be run in real-time whenever a request is rejected by an online system. Three main online reinsertion heuristics based on different neighborhoods are proposed. The first heuristic called HDR uses destruction and repair operators, the second heuristic called GH (graph heuristic) uses the ejection chain concept, and allow modeling the reinsertion problem as a constrained shortest path problem on directed graph built from a current solution, and the third heuristic called IGH (improved graph heuristic) based on GH with intensive exploration of the neighborhood. Our proposed approaches were extensively tested on real and hard instances provided by *Padam*. These instances contain up to 1011 requests and 14 vehicles and with very tight time windows. The obtained results revealed the performance of the proposed reinsertion methods compared to the online heuristic, where the number of served requests is increased and the number of vehicles is reduced while maintaining operating costs close to those of the online system. The results confirm the interest of using reinsertion techniques in the dynamic DARP service.

## 1. Introduction

The substantial growth of the transport sector in recent years has made it the prime player in energy consumption and greenhouse gas emissions, for example, transport accounts for 40%[2] of France's $CO_2$ emissions and 82% of passenger transport is by private car. In addition, most private trips are made with only the driver in the vehicle. The world situation is similar: according to [29] the cost of empty seats is estimated at 500 billion dollars per year. The transport sector has thus become a real economic and environmental challenge addressed by local authorities, public and private transport operators. Beyond a simple incentive for better occupancy of private vehicles, new challenges and initiatives are emerging, aimed at reducing the importance of private vehicles in the mobility chain and encouraging the efficient use of vehicles.

In this perspective of new mobility and thanks to the recent development of Information and communication technology (ICT) several car-sharing and car-pooling initiatives have been developed. These initiatives are a response to the economic and ecological challenges and initiate a fundamental change where the user is looking for a mobility service rather than a personal mode of transport. These new services offer to users a significant degree of flexibility at a competitive price, providing in some cases a solid alternative to conventional public transport systems.

On-demand public transport (ODT) attempts to innovate by offering a high level of service close to that of taxis in large cities, with significantly lower costs and environmental impact. There are many opportunities to develop efficient and easy-to-use ODT services. The companies offering this service will have to face a challenge of interconnection and inter-modality: they must be able to offer a unified service experience to local authorities (in the context of public transport), transport organization institutes and citizens. In this context, the start-up *Padam* is positioned as a Software as a Service (SaaS) provider, which aims to optimize on-demand public transport systems.

* Corresponding author.
*E-mail address:* oulamara@loria.fr (A. Oulamara).

[1] www.padam.io

[2] https://www.climate-transparency.org/countries/europe/france

https://doi.org/10.1016/j.jocs.2020.101199
Received 7 January 2020; Received in revised form 23 July 2020; Accepted 5 August 2020
Available online 25 August 2020

This study is the continuation of joint research with the company *Padam* that proposed an offline module based on an adaptive large neighborhood search (ALNS) approach to optimize vehicle travel between user reservations [31]. In this paper, we have contributed to enhancing *Padam*'s optimization system by developing novel reinsertion algorithms of users requests based on preliminary results presented in [32]. The contributions of our paper are as follows: (1) solving of a real and complex problem encountered by *Padam* that can be modeled as a Dial-a-Ride Problem (DARP); (2) three new fast reinsertion algorithms that allow requests rejected by the online algorithm to be processed in real time: the first, called HDR, uses destruction and repair operators; the second, called GH (graph heuristics), uses the concept of an ejection chain; the third IGH (improved graph heuristic), allows more intensive exploration of the neighborhood. These reinsertion techniques are not used in the literature, only the study [16] proposed a reinsertion algorithm for dynamic DARP, in which the objective is to minimize the number of vehicles. Beyond the application to the *Padam* problem, our approach intends to be a generic approach to the dynamic DARP, the originality of our approaches consists in exploring intensively large neighborhood of a current solution compared to [16] that exploits a similarity between the new request and requests available in the solution to select a set of requests to be ejected. (3) Numerical experiments on real instances of the company show the significant contribution of reinsertion techniques in improving the quality of solutions.

The remainder of this paper is structured as follows: in Section 2, we present a brief review on the DARP. In Section 3, we present the industrial context and we provide a formal description of the problem. Our solution methods are presented in Section 4. Computational results are provided and analyzed in Section 5. Finally, some conclusions and research directions are outlined in Section 6.

## 2. Related work

The DARP is a variant of the vehicle routing problem with pickup and delivery [3,30]. The basic version of DARP consists of serving a set of users who specify their departure and arrival locations using a fleet of vehicles based at the depot. The user specifies the desired pickup time or the desired drop-off time or both. The objective is to design a set of vehicle routes with the minimum cost, and that are able to satisfy all demands, under a set of constraints. The most usual constraints relate to vehicle capacity, route duration and maximum ride time, i.e., the time spent by a user in the vehicle. There are two main versions of the problem: static and dynamic DARP. In the first case, all requests are known in advance before computing a solution while in the second case a part of the user requests arrives in real time and vehicle routes being adjusted in real-time to meet new demand. Dynamic DARP has received less consideration than the static case. For a recent survey on both static and dynamic cases see [13].

Dynamic DARP is frequently encountered when certain requests arrive in real time during the performance of the service and/or when an immediate response is expected when a customer is booking a trip. This requires the development of techniques that can manage incoming demand flows while providing high quality solutions. Thus, many studies on dynamic DARP have focused on fast heuristics to insert new requests and/or metaheuristic methods to provide more intense optimization to construct routes of vehicles [1,6,24]. Some studies aim to use stochastic information, such as the occurrence of future demands or traffic forecasts, for example [26,27,25,18]. This is referred to as dynamic and stochastic DARP. Other approaches focus on studying service parameters in a dynamic DARP [11,12]. The solving approaches for dynamic DARP proposed in the literature can be divided into three categories:

- Insertion heuristics: Insertion heuristics have been widely used to solve dynamic DARP problem because they are well suited for real-time responses. Several insertion strategies have been studied such as fuzzy logic in [17] and scheduling techniques [33]. For other techniques see [13].
- Insertion/reinsertion heuristics: In a real situation, it may happen that a customer does not receive a successful response to his request. This means that the configuration of the current rides does not allow serving the new request. In this case, rather than simply reject the customer's request, it is possible to move other requests already inserted and not yet served from one vehicle to another to find an arrangement that will serve the customer's request. The rearrangement must be done in a few seconds to keep the response time to the customer short. We call the corresponding heuristic *reinsertion algorithm*. We identified in the literature only the study [16] that use reinsertion heuristics to solve dynamic DARP. The authors have adapted the approach proposed for the static case [15]. This method uses parallel insertion heuristic followed by reinsertion heuristic. Whenever the insertion heuristic rejects a client, a reinsertion heuristic is used to attempt to insert the rejected client by moving other clients considered as similar. Two real-time strategies are proposed: "immediate insertion" in which a request is inserted as soon as it appears and "rolling-horizon insertion" in which the insertion of a request with a service time away from the present time is rescheduled to a later time.
- Event-based optimization: These heuristics run continuously between the appearance of two events allowing to benefit from the system's latency to improve the solutions. An event corresponds to the occurrence of a request, arrival of a vehicle at a stop, etc. Most of the proposed methods are based on metaheuristics such as parallel tabu search [1], variable neighborhood search (VNS) in [26], a greedy randomized adaptive search procedure (GRASP) in [24]. Some studies propose two-phase heuristics, a fast insertion heuristic followed by a post-optimization phase. This post-optimization phase is run each time an event occurs. It can use a local search as in [6] where the local search is continuously performed between consecutive vehicle stops. In [2] authors propose a post-optimization based on a tabu search and is performed between the appearance of requests. A destruction/repair heuristic is used as a post-optimization phase by [11]. Very few heuristics have been proposed, one can cite as an example the study of [14], where the authors proposed a heuristic called REGRET which calculates for each request a regret value, evaluating the profit obtained by moving this request from its current vehicle to another. The request with the greatest regret is chosen, and the procedure is iterated until no further improvement is possible.

The literature review reveals that reinsertion algorithms for dynamic DARP are only proposed in [16]. In this study, the number of vehicles is unlimited and the objective is to reduce the number of used vehicles, while in our study the number of vehicles is limited. In addition, the reinsertion methods developed in [16] does not allow to intensively explore the neighborhood of the solution since they are based on a simple similarity criterion. In our paper, we present three novel reinsertion algorithms for dynamic DARP. These algorithms allow us to explore intensively the neighborhood of the current solution and to test more reinsertion solutions than the approach proposed in [16], while remaining fast and simple. We study the performance of our reinsertion heuristics both in terms of the rides duration and of the number of accepted requests. Extensive tests on real life and hard instances provided by *Padam* are performed.

## 3. Industrial context and problem description

We address the challenging real-time DARP faced by a French company *Padam* that provides mobility service with dynamic bus lines depending on customer requests. In the *Padam* service, customers send a request of transportation via a mobile application or via a website either in advance, i.e., booking a few days before the service, or in real-time for an immediate service. Customers specify their pickup and drop-off locations, the number of passengers, and the desired start time. The transport service is operated with a fixed number of vehicles. Vehicles start and end at the depot and have limited capacity. All potential vehicle stops for pickup and drop-off are predefined and correspond to nodes where a vehicle can stop without impacting the traffic, such as train or subway stations, administrative buildings, etc. Thus, the customer's pickup and destination locations are associated with the nearest predefined locations and customers are served at these locations instead of the original pickup and destination locations. These potential stops are determined by a statistical study of customers travel patterns that combine several data sources, which is not presented in this article. The objective of *Padam* is to satisfy as many request as possible during the service. Each customer must receive a response to his/her request within 1 or 2 s. The request may be rejected, for instance, when vehicles are already full, otherwise a customer receives a proposal from the system, and he/she is free to accept it. When a request is inserted in the appropriate vehicle, pickup and drop-off times are communicated to the customer.

A formal description of the dynamic DARP is defined as follows. Let $G = (V, A)$ be a directed graph, where $V$ is the set of nodes and $A$ is the set of arcs. The set of nodes $V$ represents the set of pickup and drop-off locations and a depot. For each arc $(i, j) \in A$, we define a travel time $t_{ij}$. We denote $K$ as the number of vehicles available during the service and each vehicle $k$ has a capacity $Q_k$ and a maximum service duration $T_k$. Vehicles have their own time window of service (beginning and end of service). Each transportation request is characterized by a pair $(i,j)$ of nodes, where $i$ is a pickup node and $j$ a drop-off node, $d_i$ the requested time service, the number of passengers $q_i$ with $q_j = - q_i$ and a time window $[e_i, l_i]$ around $d_i$ is associated to each request $(i, j)$, where $e_i$ ($l_i$) is the earliest (latest) arrival time of the vehicle at $i$. Each request specifies if the demand is pickup oriented (PO) or drop-off oriented (DO). When the request is PO the customer is picked up around $d_i$ in the interval $[e_i, l_i]$ whereas if the request is DO the customer must be dropped off around $d_j$ in the interval $[e_j, l_j]$. The distinction between PO and DO requests is frequently used in DARP [5]. Furthermore, for each scheduled request (i.e., inserted request in a ride) a maximum ride time $M_{ij}$ between pickup $i$ and drop-off $j$ is imposed, where $M_{ij}$ is the maximum detour that a customer can accept and it is proportional to the shortest travel time $t_{ij}$ between $i$ and $j$, i.e., $M_{ij} \leq \gamma_k \times t_{ij}$, with $\gamma_k \geq 0$. The value of $\gamma_k$ is selected from a set $\Gamma$ of coefficients already predefined and depends on the value of $t_{ij}$. For instance, we set $\gamma_k$ to 1.3 when the travel time $t_{ij}$ is in the interval [10, 20] min. The set $\Gamma$ allows to differentiate between acceptable deviation of short and long travel time.

The objective of dynamic DARP is to serve a maximum requests under the described constraints while minimizing the service time duration of the vehicles. Given the dynamic nature of the problem, where customer's requests arrive online, it is clear that maximizing only the number of served requests results in a local optimum. In our approach, we will rather use the total duration of the rides as an objective to be optimized while including the maximum number of requests in the rides and limiting the detour of the vehicles. The idea is to create trips with fewer unnecessary detours and with "straight" routes that satisfy a maximum number of requests at the end of the service.

## 4. Heuristic methods

In the *Padam* service, an *online insertion heuristic* is implemented. This heuristic rejects many requests due to its difficulty in finding feasible insertions close to the start time of the requests of customers.

In this section, we propose a *reinsertion* heuristics whose objective is to find feasible insertion of requests when the online heuristic fails to provide a successful solution. Unlike online heuristic, reinsertion heuristics are allowed to move already planned unprocessed requests from one vehicle to another in order to insert the new request. Three online reinsertion heuristics based on different neighborhoods are proposed. The first heuristic called *destroy-repair heuristic* (HDR) uses destruction and repair operators, the second called *graph heuristic* (GH) is based on the concept of ejection chain in a directed graph generated from a current solution, and the third one called *improved graph heuristic* (IGH) is a recursive version of GH allowing more intensive neighborhood exploration. Before detailing the reinsertion heuristics, we present a brief description of the online algorithm.

### 4.1. Online insertion heuristic

When a new request appears, the insertion heuristic is run to get a quick proposal list for the customer, generally in less than 1 s time response. This *insertion heuristic* tests each possible insertion position of a pickup and drop-off in each ride. If feasible insertions are found, the heuristic returns several proposals that are submitted to the customer, each one at a different time like timetables in public transportation system. The customer is free to choose one of the proposals or to reject them. The proposals are characterized by their pickup and drop-off hours. For instance, if $h$ is the requested hour of the service, the customer may be interested in proposals in the time interval TW $= [h - W, h + W]$, where $W$ is around 20 min [31]. When a customer chooses a proposal with $h_p$ and $h_d$ as pickup and drop-off time, respectively, insertion algorithm impose a time-window around the pickup ($TW_p$) and drop-off ($TW_d$) hours as follows:

$$\mathrm{TW}_p = \left[h_p - \mathrm{PWB}, h_p + \mathrm{PWA}\right]$$

$$\mathrm{TW}_d = \left[h_d - \mathrm{DWB}, h_d + \mathrm{DWA}\right]$$

where PWB, PWA, DWB, DWA are parameters fixed by the company. These time-windows ensure that future requests of customers can be inserted in the same ride while maintaining a high-quality service for planned requests.

### 4.2. Heuristic destroy and reinsert – HDR

The heuristic destroy and reinsert – HDR is based on destroy/repair neighborhoods search [28]. The idea is destroying a part of the solution by removing planned requests and then reinserting them together with the new request. Thus, each iteration of HDR consists in a destroy/repair steps, and these iterations are performed until a computing time limit *MT* is reached. Algorithm 1 describes our HDR method which is different from a pure large neighborhood search (LNS) method [28], since once a solution is found, it is no longer improved, but rather returns to the original solution and trying to find another solution. This is needed for practical reason, since at this stage, it is not certain that the customer will validate the proposal. Even when the customer validates it, he/she may decide to do it a few minutes later. In the meantime, other requests may be accepted, which may prevent the proposed solution.

**Algorithm 1**. Heuristic destroy and repair – HDR.

```
Input: Current solution $S$, new request $r$, list of remove operators $LDO$, list of repair
       operators $LRO$, time limit $MT$
Output: Best feasible solution $S_b$
$L \leftarrow EmptyList()$;
while $run\_time \leq MT$ do
    $op \leftarrow SelectRemoveOperators(LDO)$
    $k \leftarrow ChooseNumberOfRequests()$
    $LR \leftarrow RemoveRequests(op, k)$
    $LR \leftarrow LR \cup \{r\}$
    $s_{new} \leftarrow ReinsertRequests(LR, LRO, S)$
    if $feasible(s_{new})$ then
        $L \leftarrow L \cup s_{new}$
    end
end
if $L$ not empty then
    return $GetBestSolution(L)$
end
```

The first step of Algorithm 1 consists in choosing the destruction operators (line 3) that will be applied to remove requests. A random number of requests is then selected (line 4) and removed from the current solution (line 5). The removed requests are grouped together with the new request $r$ (line 6) and then reinserted in the routes (line 7). A feasible solution is obtained when all requests are reinserted. These destruction and repair phases are repeated until the maximum $MT$ time is reached. The following sections detail the components of Algorithm 1.

### 4.2.1. Destroy step

The destroy step uses three different destroy operators. Each operator selects requests to be removed among a list $LI$ of planned requests. Note that $LI$ does not necessarily contains all planned requests. Indeed, it is not relevant to remove requests that are distant (in time) from the requested hour $h$ of the new request $r$. Thus, we define a time window $[h - W - T, h + W + T]$ with $T$ a parameter that restrict $LI$ to requests whose pickup hour or drop-off hour is in the time window $[h - W - T, h + W + T]$. If $T$ is too small, the set of requests will be small and will not provide enough opportunity to insert the new request. However, if $T$ is too large, the search will spend more time in deleting requests that are not relevant for the insertion of $r$. Section 5.2.1.1 studies the impact of several values of $T$. In addition to the list of requests to remove, destruction operators also receive the number $k$ of requests to be selected. At each iteration, $k$ is randomly selected, in the interval $[k_{min}, k_{max}]$. Finally, at each iteration, the destruction operator used is randomly selected in a list LDO of destruction operators. In the following we describe three destroy operators that are used in Algorithm 1.

Random operator: Select $k$ requests randomly.
Worst operator: Select $k$ requests with the largest savings, i.e., the gain is defined as the difference between the cost when the request is in the solution and the cost when it is removed. This operator is widely used in the literature [23]. In order to increase diversification, this operator is randomized as follow: all requests of $LI$ are sorted in decreasing order of the saving cost in a list $L$. A random number $y$ is sampled between 0 and 1 and the request at position $\lfloor y^{p_r}|L|\rfloor$ is chosen where $|L|$ is the size of $L$ and $p_r$ is a parameter. This process is repeated until $k$ requests are selected.
Relatedness operator: Select a seed request randomly and select $k - 1$ related requests to the seed request [21]. The relatedness measure between requests $i$ and $j$ is defined as follow:

$$\frac{1}{2}\left(t_{p_i,p_j} + t_{d_i,d_j}\right) + \frac{1}{2}\left(\left|u_{p_i} - u_{p_j}\right| + \left|u_{d_i} - u_{d_j}\right|\right)$$

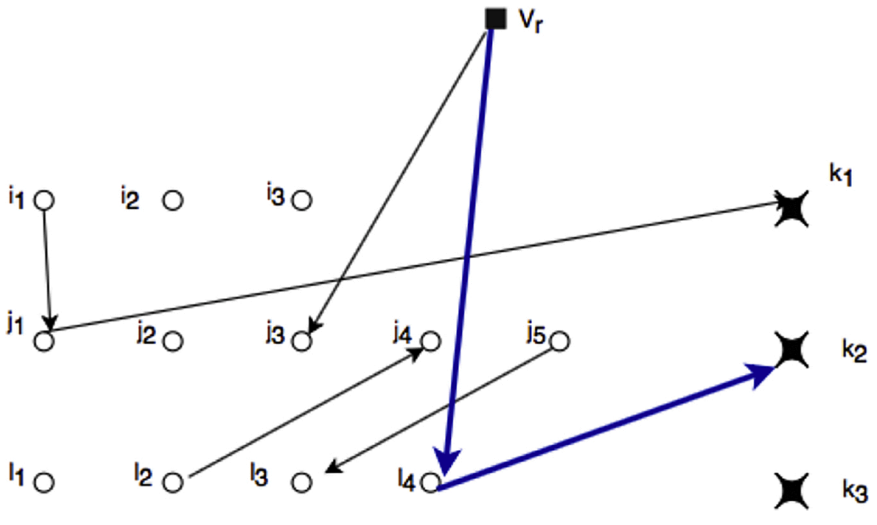


**Fig. 1.** A graph with 3 routes. $V_r$ represents the new request.

where $p_i$ and $d_i$ are respectively the pickup and drop-off nodes, $u_{p_i}$ and $u_{d_i}$ are the service time of pickup and drop-off of request $i$ and $t_{n_1,n_2}$ is the travel time between nodes $n_1$ and $n_2$. This operator is also randomized as for the Worst operator. The parameter controlling the randomness is called $p_w$. In this case $p_w = p_r$.

#### 4.2.2. Repair step

Once the requests to be removed have been selected, they are removed from their current routes, then we try to reinsert them including the new request (line 6 of Algorithm 1). When all requests are inserted while satisfying constraints of already planned requests, we obtain a feasible solution. An overview of the insertion procedure is described in Algorithm 2. An empty list of feasible solutions is initialized at the beginning of the algorithm (line 1). A list of operators is then executed sequentially (line 2) to allow each operator to insert all requests into the solution (line 3). If a feasible solution is found, then it is stored in memory (line 5). If several feasible solutions are found, the best one is selected and used as an output of the current iteration (line 9). Note that our algorithm is different from the pure LNS algorithm, where the objective of Algorithm 2 is to find a feasible solution. The use of all the operators allows to maximize the opportunity to find a feasible solution, and potentially to find several solutions to be proposed to the customer.

In our algorithm we use two operators of the literature, namely, "deep greedy" [22] and "regret" [7] operators, and a new one specific to our problem called "priority operator".

**Algorithm 2.** Reinsert requests procedure.

```
   Input: List of requests to reinsert $LR$, list of repair operators $LRO$, current solution $S$
   Output: best feasible solution $s_b$
1  $L \leftarrow EmptyList()$
2  for each operator $o$ in $LRO$ do
3      $s_{new} \leftarrow InsertRequests(o, LR, S)$
4      if $feasible(s_{new})$ then
5          $L \leftarrow L \cup s_{new}$
6      end
7  end
8  if $L$ not empty then
9      return $GetBestSolution(L)$
10 end
```

Deep greedy operator: Perform the best insertion among all feasible insertions of all remaining requests to be reinserted. The best insertion is defined as the insertion with a minimal increasing cost. The procedure is repeated until no further insertion is feasible or all requests are reinstated.

Regret operator: Insert the requests with the largest regret. For each request $i$ we define a cost $c_{ik}$ of the best insertion of $i$ in vehicle $k$. If no feasible insertion exists $c_{ik}$ is set to a large value. The regret of a request $i$ is computed as:

$$\sum_k (c_{\mathrm{ik}} - min_j c_{\mathrm{ij}})$$

Request with largest regret is then chosen and inserted in its best position. The procedure is repeated until no further insertion is feasible or all requests are inserted.

Priority operator: Insert requests in positions that will have a minimal impact on the insertion of the future requests. More precisely, we proceed with the following steps until all requests are inserted or no feasible solution is found: (1) for each request, calculate the number of routes in which it can be inserted, (2) select the request with the smallest number of routes, (3) for each route in which the insertion of the selected request is possible, calculate the number of requests of LR that can be inserted in that route, (4) select the route with the smallest number, and insert the selected request in that route.

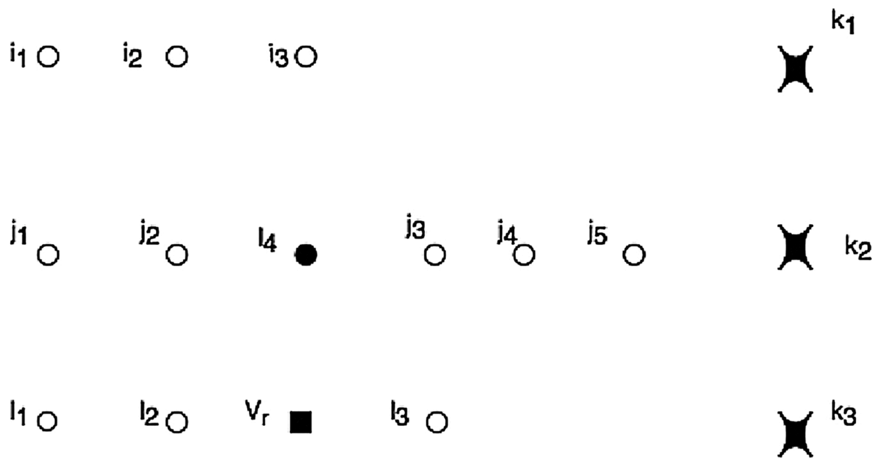


**Fig. 2.** The solution after inserting $V_r$.

Deep greedy operator: Perform the best insertion among all feasible insertions of all remaining requests to be reinserted. The best insertion is defined as the insertion with a minimal increasing cost. The procedure is repeated until no further insertion is feasible or all requests are reinstated.

Regret operator: Insert the requests with the largest regret. For each request $i$ we define a cost $c_{ik}$ of the best insertion of $i$ in vehicle $k$. If no feasible insertion exists $c_{ik}$ is set to a large value. The regret of a request $i$ is computed as:

$$\sum_{k}\left(c_{\mathrm{ik}} - min_{j}c_{\mathrm{ij}}\right)$$

Request with largest regret is then chosen and inserted in its best position. The procedure is repeated until no further insertion is feasible or all requests are inserted.

Priority operator: Insert requests in positions that will have a minimal impact on the insertion of the future requests. More precisely, we proceed with the following steps until all requests are inserted or no feasible solution is found: (1) for each request, calculate the number of routes in which it can be inserted, (2) select the request with the smallest number of routes, (3) for each route in which the insertion of the selected request is possible, calculate the number of requests of LR that can be inserted in that route, (4) select the route with the smallest number, and insert the selected request in that route.

## *4.3. Reinsertion heuristics based on ejection Chain*

In this section, we present a reinsertion heuristics based on ejection chains [10]. Ejection chain is a neighborhood structure already used in the literature to solve the DARP [20,19]. Our heuristics are based on the neighborhood structure similar to that defined in [8]. We formalized the reinsertion problem as a constrained shortest path problem in a directed graph built from a current solution, then two heuristics are proposed: graph heuristic (GH) which allows finding a solution using a modified Bellman-Ford algorithm and improved graph heuristic (IGH) which is a recursive version of GH allowing more intensive exploration of the neighborhood.

### *4.3.1. Graph construction*

Let $S$ be the current solution and $r$ be the request that are not inserted in $S$ by the online algorithm. The reinsertion problem of $r$ in $S$ is modeled as a directed graph $G' = (V', A')$, where $V'$ is a set of nodes and $A'$ is a set of arcs. The set $V' = V_1 \cup V_2 \cup \{V_r\}$, where $V_1$ corresponds to the planned requests in $S$ (one node per request), $V_2$ corresponds to the routes used in $S$ (one node per route) and $V_r$ is a node corresponding to $r$. The set $A'$ of arcs models the ejection and the insertion of requests from a route to another one. Therefore, $A'$ contains two subsets of arcs: (i) the ejection arcs $A_E$ with $A_E = \{(i, j) \mid i \in V_1 \cup \{V_r\}, j \in V_1\}$ where $(i, j) \in A_E$ represents the move of a request $i$ from its current route $r_i$ to route $r_j$, and the removal of $j$ from its route $r_j$ and (ii) the insertion arcs $A_I$, with $A_I = \{(i, k) \mid i \in V_1 \cup \{V_r\}, k \in V_2\}$ where $(i, k) \in A_I$ represents the move of a request $i$ from its current route $r_i$ to the route $r_k$ without ejecting another request from $r_k$. The cost $c_{ij}$ of an arc $(i, j) \in A_E$ is defined as the best insertion cost of $i$ in $r_j$ while removing $j$ from $r_j$ minus the savings obtained by removing $i$ from its current route $r_i$. When the insertion of $i$ in $r_j$ is infeasible we set $c_{ij}$ to $+\infty$. The cost of an arc $(i, k) \in A_I$ is defined as the insertion cost of $i$ in its new route $r_k$ minus the savings obtained by removing $i$ from $r_i$. When the insertion of $i$ in $r_k$ is infeasible we set $c_{ik}$ to $+\infty$. The construction of a graph $G'$ can be done in $O(n^3 + n^2m)$, where ejections arcs $A_E$ and insertion arcs $A_I$ need $O(n^3)$ and $O(n^2m)$ time complexity, respectively, to be constructed.

The reinsertion of $V_r$ in $S$ corresponds to a constrained shortest path problem from $V_r$ to any node $k \in V_2$, where each arc of the path has a finite cost. Such a path is defined by a sequence of arcs starting from $V_r$ and ending in a node $k \in V_2$. Fig. 1 shows an example with $V_1 = \{i_1, i_2, i_3, j_1, j_2, j_3, j_4, j_5, l_1, l_2, l_3, l_4\}$, $V_2 = \{k_1, k_2, k_3\}$. The graph of Fig. 1 depicts 3 routes, one route per line. The route $r_{k_1}$ is represented by the node $k_1$, which serves the set $\{i_1, i_2, i_3\}$ of requests, thus $r_{k_1} = \{i_1, i_2, i_3\}$, $r_{k_2} = \{j_1, j_2, j_3, j_4, j_5\}$, and $r_{k_3} = \{l_1, l_2, l_3, l_4\}$. Only arcs with a finite cost are illustrated in Fig. 1. In this example, we have $A_E = \{(V_r, j_3), (i_1, j_1), (l_2, j_4), (j_5, l_3), (V_r, l_4)\}$, and $A_I = \{(j_1, k_1), (l_4, k_2)\}$.

The blue path represents the unique path leading to a feasible solution in this graph and represents the nodes to be ejected/inserted without specifying the insertion positions.

Applying the sequence of movements defined by this path enables us to switch from the solution of Fig. 1 to the new solution shown in Fig. 2. In Fig. 2, $V_r$ (black square) is inserted in route $r_{k_3}$ in its best insertion position (after removal of $l_4$), while $l_4$ (black circle) is ejected from $r_{k_3}$ and inserted in its best insertion position in route $r_{k2}$. The total cost of the blue path allows computing the objective function of the new solution represented in Fig. 2. The shortest path calculated on this graph is constrained since the path must visit at most one node in a given route. This restriction is needed, otherwise the ejection cost would not be correct for the second visit due to the previous ejection in the same route. Thus, we impose a non-return constraint to ensure that the cost of solution defined by a given path is feasible.

### *4.3.2. Graph heuristic – GH*

The problem of constrained shortest path encountered in our reinsertion problem is NP-hard [9]. We have therefore chosen to adapt the Bellman-Ford algorithm to solve this problem, ensuring that at each iteration of the Bellman-Ford algorithm, adding an arc to the calculation of a path is performed only if the two vertices of the arc are in two different routes. The complexity of the Bellman-Ford algorithm remains unchanged by our modification. Algorithm 3 presents the reinsertion heuristic called graph heuristic – GH. Algorithm 3 receives as input the current solution, the request $r$ to be reinserted and a parameter $T$. First, the graph described in Section 4.3.1 is constructed (line 1). Only requests with a pickup or drop-off time in the range $[d_r - W - T,\ d_r + W + T]$ are considered, where $d_r$ is the requested hour of $r$, and $W$ is a fixed parameter defined in Section 4.1. Once the graph is built, an heuristic is run to determine the shortest path from $V_r$ to all nodes in the graph (line 2). The node $k \in V_2$ with finite shortest cost from node $V_r$ to node $k$ is then selected (line 3). If no nodes are found, the search is ended by returning an empty path.

**Table 1**
Characteristics of sets $A$ and $B$.

| Set | Nodes | Service duration | Vehicle capacity | Area (km$^2$) | Fleet size | Requests |
|---|---|---|---|---|---|---|
| A | 473 | 3 h 30 | 8 | 25 | 13 | 120–360 |
| B | 90 | 12 h | 30 | 5 | 6 | 200–1011 |

**Algorithm 3**. Graph heuristic – GH.

```
Input: current solution $S$, request $r$, $T$
Output: feasible best path if exist
1 graph ← constructGraph(S, r, T)
2 dist_dict, path_dict ← BellmanFordModified(graph, r)
3 best_route ← getBestRoute(dist_dict)
4 if best_route exist then
5     best_path ← getBestPath(path_dict)
6     return best_path
7 end
```

### 4.3.3. *Improved graph heuristic – IGH*

In the case where no feasible path is found by GH, the search stops even when the time limit has not reached. To overcome this drawback, we propose a new and improved version of GH called improved graph heuristic (IGH) based on *recursive* calls of GH. Algorithm 4 presents the pseudo-code of IGH. The input parameters of IGH are the same as those of GH. The search begins with the construction of graph (line 1) defined in Section 4.3.1. A list of already reinserted requests is maintained throughout the search and initialized with *r* at the beginning of the algorithm (line 2). Then the main recursive function of the algorithm is called (line 3) and the shortest path is then returned if exists (line 5).

**Algorithm 4**. Improved graph heuristic – IGH.

```
Input: current solution $S$, request $r$, $T$
Output: best feasible path if there is one
1 graph ← constructGraph(S, r, T)
2 req_already_ins ← List(r)
3 best_path ← RecursiveReinsertion(r, graph, EmptyList(), req_already_ins, T)
4 if best_path exist then
5     return best_path
6 end
```

**Algorithm 5**. Recursive reinsertion function.

```
Output: best feasible path if exists
1  Function RecursiveReinsertion(request, graph, path_act, req_already_ins, T)
2  dist_dict, path_dict ← BFSModified(graph, request)
3  if NumberFeasiblePath(dist_dict) = 0 then
4      new_req ← ExecuteBestPartialPath(dist_dict, req_already_ins, graph)
5      if new_req does not exist then
6          return
7      end
8      req_already_ins ← req_already_ins ∪ new_req
9      path_to_process ← path_dict[new_req]
10     path_act ← updatepath(path_act, path_to_process)
11     list_impacted_routes ← FindListofRouteInPath(path_to_process)
12     graph ← RemoveObsoleteArcs(graph, new_req, list_impacted_routes)
13     graph ← updateGraph(request, graph, list_impacted_routes, T)
14     return RecursiveReinsertion(new_req, graph, path_act, req_already_ins, T)
15 end
16 best_path ← getBestPath(path_dict)
17 best_path ← updatepath(path_act, best_path)
18 return best_path
19 End Function
```

**Table 2**
Service quality for each set of instances.

| Set | PWB/PWA | DWB/DWA | Gamma levels | Gamma values | *W* | *D* |
|---|---|---|---|---|---|---|
| A | 0/10 | 10/13 | ]0, 10]/]10, 20]/]20, ∞] | {2, 1.8, 1.7} | 20 | 1 |
| B | 5/8 | 10/10 | ]0, 5]/]5, 10]/]10, ∞] | {2.5, 2, 1.8} | 20 | 0.5 |

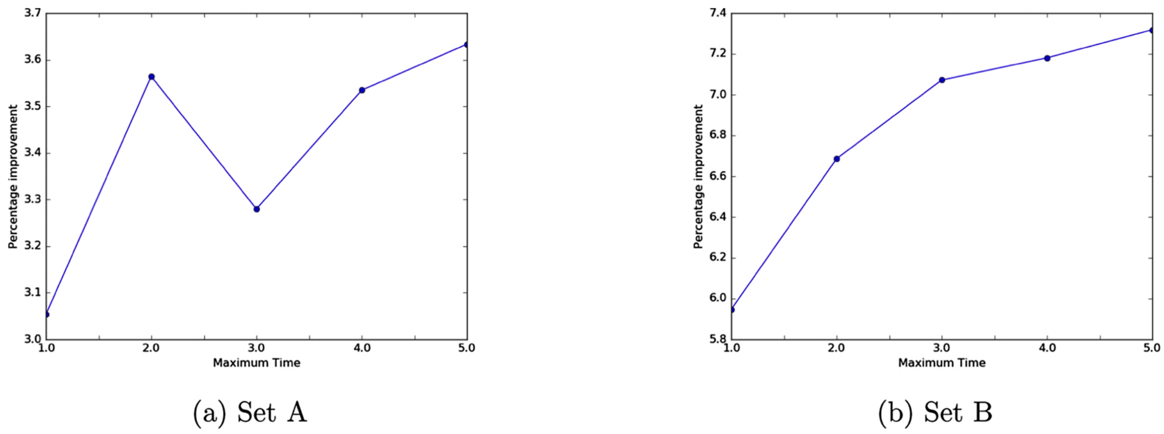


**Fig. 3.** Relative improvement of HDR based on the values of MT. *y*-axis represents the average relative improvement over all instances and the *x*-axis represents the values of MT (s).

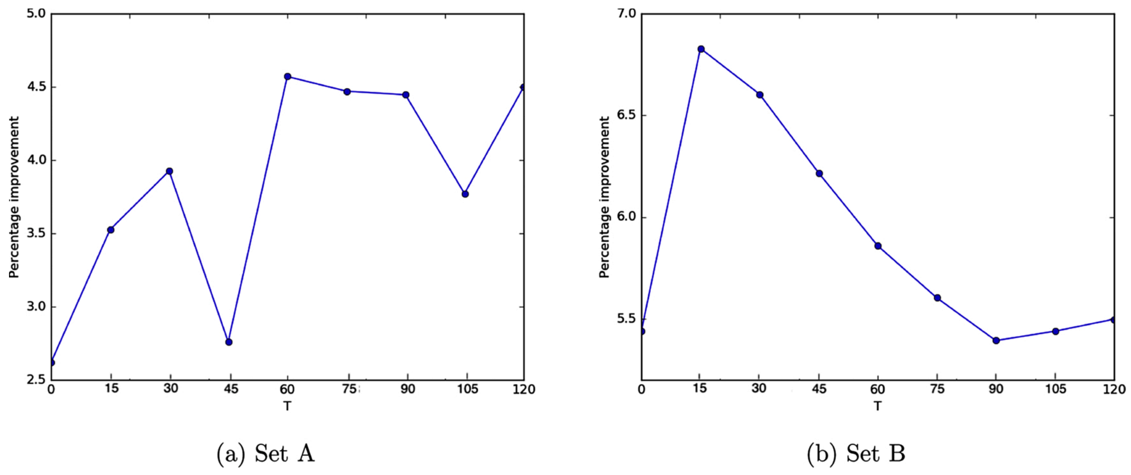


**Fig. 4.** Relative improvement of HDR according to the value of *T*. *y*-axis represents the average of the relative improvement on all instances and *x*-axis the values of *T* (min).

Algorithm 5 gives the pseudo-code of *recursive reinsertion* function used in Algorithm 4. The main idea of this function is as follows: when a search of the shortest path does not yield a feasible path while a partial path from $r$ to $i \in V_1$ is found, we apply this partial path to solution $S$. We thus obtain a new solution with $r$ inserted and $i$ ejected. The procedure is restarted with $i$ instead of $r$. The procedure is repeated as many times as necessary until a time limit is reached or a solution is found with all requests inserted.

The input parameters of the function are as follows: "request" corresponds to the request currently ejected, "graph" corresponds to the graph of the current solution, "path_act" represents the current path and allows to save all successive paths performed by recursive calls of the function which allow to return the complete path at the end of the algorithm, "req_already_ins" represents requests which were ejected at the previous iterations and $T$ is a parameter with the same role as in Algorithm 3. The recursive reinsertion function starts by searching for the shortest path in the current graph using a modified version of Bellman-Ford algorithm as explained in Section 4.3.2. If no feasible path from "request" to a node $k \in V_2$ is found (line 4), a partial path is examined (line 5). If no partial path is found (line 8), the search is ended. Otherwise, the new ejected request is added to list of ejected requests (line 8). The current path is then updated (lines 9 and 10) and the list of routes impacted by this path is then determined. All arcs in the graph connected to at least one node belonging to the impacted routes are potentially obsolete. These arcs are therefore removed from the graph (line 12), which is then updated by recomputing the missing arcs with the new information (line 13). This step avoids having to recompute the entire graph if only 1 or 2 routes have been impacted. The end of the function (lines 16 and 17) aims to find the best final path. In order to select the best partial path (line 4), we define three criteria, namely,

PPA: Partial path that ends with a request that has the most outgoing arcs. If several partial paths are equivalent, a partial path is selected randomly
PPB: Partial path with shortest cost
PPC: Partial path that ends with a request that has the most outgoing arcs. If several partial paths are equivalent, the partial path with shortest cost is selected

**Table 3**
Average number of requests and feasible moves based on the value of *T* (min).

| *T* | 0 | 60 | 120 |
|---|---|---|---|
| *Set A* | | | |
| Neighborhood | 23.71 | 55.78 | 62.81 |
| Feasible | 1.72 | 1.68 | 1.43 |
| *Set B* | | | |
| Neighborhood | 13.51 | 46.29 | 66.20 |
| Feasible | 13.49 | 8.23 | 6.12 |

**Table 4**
Performance of HDR on the instances of *A*.

| Ve | 12 | | | | 13 | | | | 14 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| INS | O | O+R | Imp | ImpD | O | O+R | Imp | ImpD | O | O+R | Imp | ImpD |
| A_120 | 67.5 | 74.33 | **10.12** | **−0.40** | 74.17 | 82.00 | **10.56** | **−1.40** | 75.83 | 83.5 | **10.11** | 0.74 |
| A_145 | 53.10 | 55.66 | **4.81** | 1.06 | 57.24 | 60.76 | **6.14** | **−0.73** | 62.06 | 61.52 | −0.89 | **−2.91** |
| A_159 | 60.38 | 62.77 | **3.96** | 3.21 | 66.04 | 68.49 | **3.71** | 2.57 | 65.41 | 71.95 | **10.00** | 2.04 |
| A_177 | 42.37 | 46.67 | **10.13** | 4.88 | 44.63 | 46.87 | **4.56** | **−0.86** | 46.33 | 47.23 | **1.95** | **−2.55** |
| A_216 | 43.52 | 42.78 | −1.70 | 1.04 | 44.44 | 45.09 | **1.46** | **−0.43** | 45.37 | 48.29 | **6.43** | 0.38 |
| A_233 | 51.07 | 52.58 | **2.94** | 0.62 | 55.79 | 58.67 | **5.15** | 3.57 | 60.94 | 61.46 | **0.85** | **−0.38** |
| A_271 | 34.32 | 34.98 | **1.93** | **−0.21** | 36.53 | 39.34 | **7.68** | **−3.90** | 38.75 | 41.92 | **8.19** | **−0.91** |
| A_288 | 34.03 | 34.06 | **0.10** | 0.88 | 34.72 | 35.49 | **2.20** | 0.39 | 35.07 | 37.92 | **8.12** | 0.95 |
| A_288b | 45.49 | 51.18 | **12.52** | **−0.58** | 54.86 | 53.96 | −1.65 | **−0.14** | 55.55 | 57.53 | **3.56** | **−0.89** |
| A_360 | 26.39 | 27.47 | **4.11** | 0.30 | 28.33 | 30.00 | **5.88** | **−2.86** | 30.27 | 32.56 | **7.52** | 0.48 |
| Avg. | 45.82 | 48.25 | **4.89** | 1.08 | 49.68 | 52.05 | **4.57** | **−0.38** | 51.56 | 54.39 | **5.58** | **−0.31** |

Values in bold are the best values.

**Table 5**
Performance of HDR on the instances of *B*.

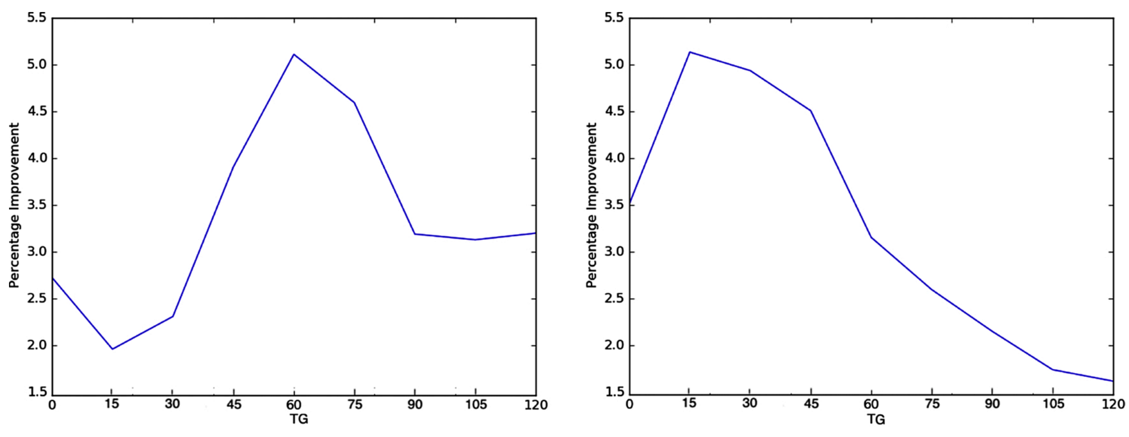
| Ve | 5 | | | | 6 | | | | 7 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| INS | O | O+R | Imp | ImpD | O | O+R | Imp | ImpD | O | O+R | Imp | ImpD |
| B_200 | 84.5 | 88.95 | **5.27** | 5.27 | 90.5 | 92.75 | **2.49** | **−2.42** | 89.5 | 92.0 | **2.79** | 1.93 |
| B_301 | 82.06 | 91.06 | **10.97** | 3.95 | 89.04 | 92.29 | **3.66** | **−1.61** | 91.03 | 94.15 | **3.43** | **−0.80** |
| B_398 | 86.93 | 90.70 | **4.34** | 1.29 | 88.94 | 91.21 | **2.54** | 0.39 | 87.44 | 91.88 | **5.08** | 1.99 |
| B_498 | 76.51 | 85.94 | **12.34** | 2.42 | 85.94 | 90.84 | **5.70** | 2.40 | 87.35 | 93.88 | **7.47** | 0.97 |
| B_607 | 72.32 | 82.19 | **13.64** | 0.85 | 80.72 | 88.37 | **9.47** | 1.88 | 84.51 | 91.57 | **8.34** | **−0.38** |
| B_709 | 69.25 | 78.22 | **12.95** | 0.87 | 80.82 | 84.79 | **4.92** | 0.96 | 84.34 | 89.01 | **5.53** | 1.32 |
| B_802 | 67.83 | 75.01 | **10.58** | **−0.29** | 72.57 | 82.77 | **14.05** | 2.08 | 78.05 | 86.66 | **11.02** | 0.06 |
| B_909 | 61.45 | 75.01 | **17.29** | 0.48 | 73.68 | 81.38 | **10.45** | 1.17 | 76.98 | 85.45 | **11.00** | 2.90 |
| B_1011 | 61.03 | 69.00 | **13.06** | **−0.02** | 70.23 | 77.51 | **10.37** | 0.44 | 75.57 | 82.11 | **8.65** | 0.38 |
| Avg. | 73.54 | 81.46 | **11.16** | 1.65 | 81.38 | 86.88 | **7.07** | 0.59 | 83.86 | 89.64 | **7.04** | 0.93 |

Values in bold are the best values.

## 5. Computational experiments

The proposed algorithms are implemented in Python and tested on realistic instances. All simulations are performed on a Personal Computer with 32GB of RAM and an Intel Xeon E3-1245 quad-core processor running at 3.40 GHz. For each instance, we run a simulation with *online algorithm* (Section 4.1) to obtain reference results denoted *RV*. Furthermore, as HDR and IGH methods are stochastic, for each method, we run 10 simulations on each instance and we use the average results, denoted *V*. The comparison between *V* and *RV* is performed in relative percentage terms, $\frac{V-RV}{RV} * 100$.

In the following, detailed characteristics of the real instances are provided, then the performance of reinsertion heuristics are evaluated. The impact of different parameters of reinsertion heuristics is also presented. Section 5.2 discusses parameters tuning and performance of the HDR method. Section 5.3 exposes parameters tuning and performance of GH and IGH methods. Sections 5.3 and 5.4 provide a comparative study on the performance of the three heuristics.



(a) Set A (b) Set B

**Fig. 5.** Relative improvement according to the values of *T*. The *y*-axis represents the average of the relative improvement on all instances and the *x*-axis represents the different values of *T* (min).

**Table 6**
Average performance of *PPA*, *PPB* and *PPC* on both sets of instances.

| Set | PPA | PPB | PPC |
|---|---|---|---|
| A | 5.11% | 4.54% | 3.84% |
| B | 5.13% | 5.06% | 5.00% |

### *5.1. Instances*

All experiments are conducted on 19 real instances provided by *Padam*, divided into two sets *A* and *B*, containing 10 and 9 instances respectively. Instances of the same set differ by the number of requests, and each instance is denoted $G\_N$, where *G* is the set name and *N* the number of requests. Fleet information and geographical data of each set are presented in Table 1 in which columns *nodes* denotes the number of pickup/drop-off locations, *service duration* refers to the operating time of each vehicle, *vehicle capacity* is the number of seats in each vehicle, *area* is the surface in km$^2$ covered by pickup/drop-off nodes, *fleet size* is the number of vehicles, and *requests* refers to the minimum and maximum number of requests among all instances of the set.

Set A corresponds to a dense municipality in United Kingdom in which *Padam*'s service completes the daily home-to-work/work-to-home trips by connecting people's homes in the north of the municipality with the main railway and transport stations in the south of the municipality. The service is mainly used at peak times and is operated with mini-buses. However, set B is a rural municipality in France without public transportation. *Padam* uses regular buses and the service operates during the whole day.

Table 2 presents, for each set of instances, the values of the different parameters. Columns *PWA/PWB* and *DWA/DWB* describe the time-

**Table 7**
Performance of IGH on the instances of the set *A*.

| Ve | 12 | | | | 13 | | | | 14 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| INS | O | O+R | Imp | ImpD | O | O+R | Imp | ImpD | O | O+R | Imp | ImpD |
| A_120 | 67.5 | 72.42 | **7.28** | **−1.12** | 74.17 | 80 | **7.87** | 0.58 | 75.83 | 81.08 | **6.92** | **−0.77** |
| A_145 | 53.10 | 54.48 | **2.60** | 0.77 | 57.24 | 60.69 | **6.02** | 0.57 | 62.07 | 63.45 | **2.22** | **−2.11** |
| A_159 | 60.38 | 62.26 | **3.12** | 1.79 | 66.04 | 69.81 | **5.71** | 4.18 | 65.41 | 71.07 | **8.65** | 8.23 |
| A_177 | 42.37 | 48.02 | **13.3** | 7.18 | 44.63 | 46.89 | **5.06** | **−0.26** | 46.33 | 49.15 | **6.1** | **−0.12** |
| A_216 | 43.52 | 42.13 | −3.19 | 2.49 | 44.44 | 46.30 | **4.17** | **−0.57** | 45.37 | 47.69 | **5.10** | 1.53 |
| A_233 | 51.07 | 54.51 | **6.72** | 3.58 | 55.8 | 58.37 | **4.62** | 4.27 | 60.95 | 60.6 | −0.56 | 0.12 |
| A_271 | 34.32 | 33.58 | −2.15 | 0.93 | 36.53 | 39.11 | **7.07** | **−1.72** | 38.75 | 40.92 | **5.62** | **−0.94** |
| A_288b | 45.49 | 47.57 | **4.50** | 1.81 | 54.86 | 52.01 | −5.19 | 0.46 | 55.55 | 56.81 | **7.33** | 2.96 |
| A_288 | 34.03 | 33.02 | −2.96 | 2.77 | 34.72 | 36.11 | **4.0** | 0.73 | 35.07 | 37.64 | **2.25** | **−1.11** |
| A_360 | 26.39 | 26.39 | 0 | 0.93 | 28.33 | 31.67 | **11.76** | **−1.46** | 30.27 | 32.31 | 6.70 | 0.43 |
| Avg. | 45.82 | 47.43 | **2.93** | 2.11 | 49.68 | 52.1 | **5.11** | 0.68 | 51.56 | 54.07 | **5.03** | 0.48 |

Values in bold are the best values.

**Table 8**
Performance of the IGH on the instances of the set *B*.

| Ve | 5 | | | | 6 | | | | 7 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| INS | O | O+R | Imp | ImpD | O | O+R | Imp | ImpD | O | O+R | Imp | ImpD |
| B_200 | 84.5 | 88.5 | **4.73** | 6.00 | 90.5 | 93.5 | **3.31** | 3.59 | 89.5 | 91.5 | **2.23** | 1.71 |
| B_301 | 82.06 | 89.37 | **8.91** | 6.85 | 89.04 | 93.36 | **3.73** | 0.61 | 91.03 | 93.02 | **2.19** | **−3.60** |
| B_398 | 86.93 | 88.69 | **2.02** | 0.88 | 88.94 | 88.47 | −0.54 | **−2.30** | 87.44 | 92.46 | **5.75** | 7.96 |
| B_498 | 76.51 | 84.60 | **10.58** | 6.47 | 85.94 | 89.98 | **4.70** | 4.10 | 87.35 | 92.70 | **6.11** | **−0.36** |
| B_607 | 72.32 | 78.98 | **9.20** | 0.50 | 80.72 | 86.82 | **7.55** | 5.14 | 84.51 | 90.12 | **6.63** | 2.18 |
| B_709 | 69.25 | 76.73 | **10.79** | 3.93 | 80.82 | 82.99 | **2.69** | 2.83 | 84.34 | 88.17 | **4.53** | 3.90 |
| B_802 | 67.83 | 73.48 | **8.33** | 2.28 | 72.57 | 79.69 | **9.81** | 2.40 | 78.05 | 84.71 | **8.53** | 2.39 |
| B_909 | 61.45 | 69.90 | **13.75** | 0.89 | 73.68 | 79.83 | **8.36** | 3.30 | 76.98 | 82.62 | **7.32** | 3.33 |
| B_1011 | 61.03 | 65.96 | **8.09** | 0.65 | 70.23 | 74.86 | **6.59** | 2.15 | 75.57 | 80.62 | **6.69** | 1.35 |
| Avg. | 73.54 | 79.58 | **8.49** | 3.16 | 81.38 | 85.39 | **5.14** | 2.43 | 83.86 | 88.44 | **5.55** | 2.10 |

Values in bold are the best values.

**Table 9**
Comparison between HDR and IGH on instances of set *A*.

| | Imp | | | | | | ImpD | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VE | 12 | | 13 | | 14 | | 12 | | 13 | | 14 | |
| INS | HDR | IGH | HDR | IGH | HDR | IGH | HDR | IGH | HDR | IGH | HDR | IGH |
| A_120 | **10.12** | 7.28 | **10.56** | 7.87 | **10.11** | 6.92 | −0.40 | **−1.12** | **−1.40** | 0.58 | 0.74 | **−0.77** |
| A_145 | **4.81** | 2.60 | **6.14** | 6.02 | −0.89 | **2.22** | 1.06 | **0.77** | **−0.73** | 0.57 | **−2.91** | −2.11 |
| A_159 | **3.96** | 3.12 | 3.71 | **5.71** | **10.00** | 8.65 | 3.21 | **1.79** | **2.57** | 4.18 | **2.04** | 8.23 |
| A_177 | 10.13 | **13.3** | 4.56 | **5.06** | 1.95 | **6.1** | **4.88** | 7.18 | **−0.86** | −0.26 | **−2.55** | −0.12 |
| A_216 | **−1.70** | −3.19 | 1.46 | **4.17** | **6.43** | 5.10 | **1.04** | 2.49 | −0.43 | **−0.57** | **0.38** | 1.53 |
| A_233 | 2.94 | **6.72** | **5.15** | 4.62 | **0.85** | −0.56 | **0.62** | 3.58 | **3.57** | 4.27 | **−0.38** | 0.12 |
| A_271 | **1.93** | −2.15 | **7.68** | 7.07 | **8.19** | 5.62 | **−0.21** | 0.93 | −3.90 | **−1.72** | −0.91 | **−0.94** |
| A_288 | **0.10** | −2.96 | 2.20 | **4.0** | **8.12** | 2.25 | **0.88** | 2.77 | **0.39** | 0.73 | 0.95 | **−1.11** |
| A_288b | **12.52** | 4.50 | **−1.65** | −5.19 | 3.56 | **7.33** | **−0.58** | 1.81 | **−0.14** | 0.46 | **−0.89** | 2.96 |
| A_360 | **4.11** | 0 | 5.88 | **11.76** | **7.52** | 6.70 | **0.30** | 0.93 | **−2.86** | −1.46 | 0.48 | **0.43** |
| Avg. | **4.89** | 2.93 | 4.57 | **5.11** | **5.58** | 5.03 | **1.08** | 2.11 | **−0.38** | 0.68 | **−0.31** | 0.48 |

Values in bold are the best values.

**Table 10**
Comparison between HDR and IGH on instances of the set *B*.

| | Imp | | | | | | ImpD | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VE | 5 | | 6 | | 7 | | 5 | | 6 | | 7 | |
| INS | HDR | IGH | HDR | IGH | HDR | IGH | HDR | IGH | HDR | IGH | HDR | IGH |
| B_200 | **5.27** | 4.73 | 2.49 | **3.31** | **2.79** | 2.23 | **5.27** | 6.00 | **−2.42** | 3.59 | 1.93 | **1.71** |
| B_301 | **10.97** | 8.91 | 3.66 | **3.73** | **3.43** | 2.19 | **3.95** | 6.85 | **−1.61** | 0.61 | −0.80 | **−3.60** |
| B_398 | **4.34** | 2.02 | **2.54** | −0.54 | 5.08 | **5.75** | 1.29 | **0.88** | 0.39 | **−2.30** | **1.99** | 7.96 |
| B_498 | **12.34** | 10.58 | **5.70** | 4.70 | **7.47** | 6.11 | **2.42** | 6.47 | **2.40** | 4.10 | 0.97 | **−0.36** |
| B_607 | **13.64** | 9.20 | **9.47** | 7.55 | **8.34** | 6.63 | 0.85 | **0.50** | **1.88** | 5.14 | **−0.38** | 2.18 |
| B_709 | **12.95** | 10.79 | **4.92** | 2.69 | **5.53** | 4.53 | **0.87** | 3.93 | **0.96** | 2.83 | **1.32** | 3.90 |
| B_802 | **10.58** | 8.33 | **14.05** | 9.81 | **11.02** | 8.53 | **−0.29** | 2.28 | **2.08** | 2.40 | **0.06** | 2.39 |
| B_909 | **17.29** | 13.75 | **10.45** | 8.36 | **11.00** | 7.32 | **0.48** | 0.89 | **1.17** | 3.30 | **2.90** | 3.33 |
| B_1011 | **13.06** | 8.09 | **10.37** | 6.59 | **8.65** | 6.69 | **−0.02** | 0.65 | **0.44** | 2.15 | **0.38** | 1.35 |
| Avg. | **11.16** | 8.49 | **7.07** | 5.14 | **7.04** | 5.55 | **1.65** | 3.16 | **0.59** | 2.43 | **0.93** | 2.10 |

Values in bold are the best values.

windows of requests, detailed in Section 4.1, column *Gamma Values* lists the different values of gammas and column *Gamma levels* shows the time intervals in which Gamma values are applied, *W* is defined in Section 4.1 and *D* represents the time to serve a request at each pickup or drop-off node. All temporal parameters are given in minutes.

Based on *Padam*'s knowledge and experience, the instances of *A* are more difficult than those of *B*, although inserting a request in instances of *A* adds a smaller detour than in instances of *B*. This is mainly due to the fact that the area of *A* has a larger number of pickup and drop-off locations than the area of *B*. So there is more pooled requests in *B* than in *A*.

## 5.2. HDR results

### 5.2.1. Parameters tuning

Before assessing the performance of HDR, we study the impact of its important parameters *T* and *MT* defined in Section 4.2.1. Some preliminary experiments are conducted to calibrate the values of the other parameters as follow: $k_{min} = 3$, $k_{max} = 10$ and $p_r = 4$. In fact, the values of $k_{min}$ and $k_{max}$ are determined empirically after several preliminary tests. These tests are conducted with the objective of finding a trade-off between the size of the search space to find feasible solutions, and the time required to find these solutions. As presented in Section 4.2.1, the list *LI* of requests candidates are those with a pickup (PO) or drop-off (DO) time in the interval $[h - W - T, h + W + T]$. Assuming a uniform distribution of the arrival of the requests during the service time horizon, the values of $k_{min}$ and $k_{max}$ correspond to a destruction interval [3 %, 10 %] of requests for instances of set A and [10 %, 34 %] of requests for instances of set B. An extensive analysis on the impact of the number of requests destroyed in the ruin and recreate approach can be found in [4].

*5.2.1.1. Impact of MT.* In order to assess the impact of the maximum allowable computation time (*MT*) on the total percentage of served requests, the value of *T* is set to 15 min for both sets A and B of instances. Fig. 3 illustrates the results on the two sets of instances. In both sets, the number of served requests increases when *MT* goes from 1 to 2 s. This suggests that allocating at least 2 s of computation time to HDR is quite efficient. Beyond 2 s, there is no significant improvement for the set *A*. The small variation between $MT = 2$ and $MT = 3$ is probably due to the stochastic nature of the algorithm. For the set *B*, an improvement is observed up to 5 s. The major improvement is observed from 1 to 3 s. The set *B* benefits more from the MT increase, probably for the reasons explained below in the section on the impact of *T*. We have therefore chosen to limit the maximum computation time to 3 s. This limit is a good trade-off between the quality of service and the efficiency of the HDR method.

*5.2.1.2. Impact of T.* In order to evaluate the impact of *T* on the percentage of served requests, we set *MT* to 3 s. Fig. 4, illustrates the results on the two sets of instances. The HDR method performs well when *T* increases until 60 min for the set *A* and until 15 min for the set *B*. The only exception is the deterioration observed in set *A* when $T = 45$ min, which is difficult to explain. The behavior of HDR is different in the two sets. In *B* the performance decreases as *T* increases until it becomes worse than when $T = 0$, while in the set *A* the performance stabilizes around an average value comparable to that of $T = 60$ min.

Table 3 presents metrics on the behavior of HDR. *Neighborhood* shows the average number of requests in *LI* and *feasible* provides the percentage of possible moves obtained by performing a simulation on each instance. We can see that the sizes of the neighborhoods are, on average, close for both sets. However, it clearly appears that it is more difficult to find feasible moves in the set *A*, which confirms the hardness of this set (see Section 5.1). Table 3 also shows that the percentage of feasible moves decreases significantly in *B* when *T* increases. Therefore, it is important to limit the number of requests to be considered in *B*. In the rest of the experiments, we set $T = 60$ (min) for the instances of *A* and $T = 15$ (min) for the instances of *B*.

*5.2.1.3. Performance of HDR.* In order to evaluate the performance of the HDR method, we consider the percentage of served requests as the objective to be maximized and we present its impact on the total duration of routes, which is the objective minimized by the algorithms implemented in the *Padam* service. Note that the route duration includes the routing time, the time to serve each request of the route and the dwelling time.

Table 4 shows the results of set *A* with 12, 13 and 14 vehicles. *O* is the percentage of served requests with the online algorithm, *O*+*R* is the percentage of served requests with the HDR method, *Imp* is the relative improvement of HDR compared to the online algorithm in terms of served requests and *ImpD* the relative improvement of HDR compared to the online algorithm in terms of routes duration.

We can see that HDR improves the results of the most instances, with a relative improvement of 4.57%, 4.89% and 5.58% using 13, 12 and 14 vehicles, respectively. The routes obtained by HDR have, on average, a similar duration to the routes obtained by the online algorithm, with a slight improvement of this duration over several instances. Thus, HDR allows serving more requests without increasing the operating costs. Comparing the results obtained with 12 and 13 vehicles, we observe that the online algorithm serves 45.82% of requests with 12 vehicles compared to 49.68% with 13 vehicles, while HDR serves 48.25% with 12 vehicles. The results also show that HDR allows to serve, on average, more requests using only 13 vehicles, than the online algorithm with 14 vehicles. This means that HDR allows to save 1 vehicle on these instances, showing a significant interest even on difficult instances. We can see that in only three instances the HDR implies a slight deterioration in the percentage of served claims (<1.7%), but in two of these instances the route duration is slightly improved. This may be due to the fact that the reinsertion heuristic can accept requests implying a deterioration in the quality of the solution and making it more difficult to insert future requests.

Table 5 shows results for set B with 5, 6 and 7 vehicles. We can see that HDR improves the results of all instances. With 6 vehicles HDR provides an average improvement of 7.07% with a total route duration similar to that obtained by the online algorithm. With 5 vehicles, it provides an improvement of 11.16% to reach an average of 81.46% of served requests. This average percentage is higher than the average percentage achieved by the online algorithm when using 6 vehicles. This means that HDR allows, on average, to use one less vehicle, which represents a considerable gain. The same result holds when comparing the results with 6 and 7 vehicles.

## 5.3. IGH results

Since the IGH method is an improvement of GH, preliminary experiments allowed us to confirm this assertion, where IGH significantly improves the results of GH on all instances of sets *A* and *B*. In this section we present only the performances of the IGH method.

### 5.3.1. Parameters tuning

*5.3.1.1. Impact of T.* The objective here is to assess the impact of *T* on the IGH method. We performed simulations with different values of *T* and using *PPA* as the criterion to select the best partial path. Fig. 5 shows the results on both sets of instances. The same behavior is observed for both sets, i.e., the performance of IGH increases as *T* increases up to a maximum value and then the performance of IGH decreases as *T* continues to increase. We also note that the impact of *T* is very high since performance can vary by a factor of three depending on the value of *T*. This is due to the fact that constructing the graph becomes time consuming when the number of requests is higher. The best value for *A* is then $T = 60$ min and for *B* is $T = 15$ min. We will use these values of *T* in the rest of experiments.

*5.3.1.2. Impact of PPA, PPB and PPC criteria.* Table 6 shows the results obtained with the three rules on the two sets of instances. In set A, PPA provides a relative improvement of more than 20% on PPC and more than 10% on PPB. The contribution of PPA is less on set B where the difference between the different rules is smaller. The conclusion is nevertheless the same for both sets: PPA is the rule that gives the best results, followed by PPB and PPC. We can thus deduce that using partial decisions based on the shortest path is not the most relevant strategy. We will therefore use PPA for the following experiments.

*5.3.1.3. Performance of IGH.* Table 7 shows the performance of IGH on the set *A* with 12, 13 and 14 vehicles. As we have seen before, these instances are very difficult. However, the IGH algorithm allows to improve almost all instances, with a relative improvement of 5.11% (2.93% and 5.03%) using 13 (12 and 14) vehicles. These results are obtained with a similar total ride duration for 13 and 14 vehicles, which shows that the IGH algorithm allows to serve more requests with similar operating costs. The results also show that with 13 vehicles IGH allows to insert, on average, as many requests as the online algorithm with 14 vehicles, while improving results of 7 instances. Thus, IGH is able to save 1 vehicle on almost all instances. We also observe that the relative improvement of IGH does not decrease when the number of vehicles reaches 14. Similar to HDR algorithm, IGH allows better exploitation of the additional vehicle.

Table 8 presents the results of IGH on the set *B* with 5, 6 and 7 vehicles. IGH provides an improvement on all instances. With 6 vehicles IGH provides an average improvement of 5.14% of served requests and can reach 85.39%. Furthermore, IGH with 6 vehicles gives better results than the online algorithm with 7 vehicles on 7 instances which is a considerable gain. The relative improvement provided by IGH is 8.49%, 5.14% and 5.55% for 5, 6 and 7 vehicles, respectively.

## 5.4. Comparison between HDR and IGH

Table 9 compares the results obtained in Sections 5.2 and 5.3 on the instances of *A*. Regarding the percentage of served requests, the HDR algorithm obtains better results on 2/3 of the instances (i.e., on 20 runs over 30 runs performed on the 10 instances with 12, 13 and 14 vehicles) and allows a slight improvement of the average performances. When 13 vehicles are used, the IGH algorithm obtains the best performances on half of the instances with higher average performances (more than 10% improvement compared to the HDR algorithm). The IGH algorithm saves 1 vehicle in 7 instances while the HDR algorithm saves 1 vehicle in 5 instances. On the other hand, the HDR algorithm is better when 12 and 14 vehicles are used in the majority of instances. In terms of vehicle use, it can be seen that the HDR algorithm obtains average performances that are always better. In short, the HDR algorithm allows to serve a higher number of requests than the IGH algorithm while ensuring significantly lower operating costs.

Table 10 compares the results obtained in Sections 5.2 and 5.3 on instances of set *B*. The HDR algorithm obtains better average performance in terms of served requests for 5, 6 and 7 vehicles. Furthermore, HDR performs better on all instances with 5 vehicles, on 7 instances with 6 vehicles and on 8 instances with 7 vehicles. These results are obtained with a lower increase in total route duration than for the IGH algorithm in the majority of cases. In short, the HDR algorithm allows to serve more requests with smaller operating costs and saves 1 vehicle on more instances than the IGH algorithm.

## 6. Conclusion

In this paper, we have investigated the design of heuristics for a challenging dynamic DARP encountered in company *Padam*. We have developed three main reinsertion heuristics: the HDR method which is based on the destruction/repair concept, the GH method which is based on the ejection chain principle allowing to model the reinsertion problem as a constrained shortest path problem in an directed graph and the IGH method which is an improved version of GH method. It is worth mentioning that reinsertion techniques for dynamic DARP are less used in the literature. The originality of our approaches consists in exploring a large neighborhood of the current solution compared to the only approach found in the literature that exploits the similarity between the new request and the requests already planned. We conducted experiments on realistic instances with up to 1011 requests and 14 vehicles to evaluate the benefits of the proposed methods. The results showed the effectiveness of the proposed reinsertion methods compared to the online heuristics, confirming the importance of the use of these techniques in the dynamic DARP system. In our approaches several parameters are set empirically, for example $k_{min}$, $k_{max}$, $p_r$. As future research directions, it is interesting to develop learning methods allowing to automatically calibrate these parameters according to the characteristics of the instances. Although the developed algorithms contain specifications of the *Padam* problem, nevertheless the reinsertion ideas used are generic and can easily be generalized to the classical case of a dynamic DARP.

## Author contributions

Ammar Oulamara: Conceptualization, methodology, writing – editing. Wahiba Ramdane-Cherif: Methodology, writing. Sven Vallée: Methodology, software, experiments.

## Conflict of interest

The authors declare that there is no conflict of interest.

## Acknowledgements

The authors are grateful to the two anonymous referees for their constructive comments.

## References

[1] A. Attanasio, J.-F. Cordeau, G. Ghiani, G. Laporte, Parallel tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem, Parallel Comput. 30 (3) (2004) 377–387.
[2] A. Beaudry, G. Laporte, T. Melo, S. Nickel, Dynamic transportation of patients in hospitals, OR Spectr. 32 (1) (2010) 77–107.
[3] W.R. Cherif-Khettaf, M.H. Rachid, C. Bloch, P. Chatonnay, New notation and classification scheme for vehicle routing problems, RAIRO – Oper. Res. 49 (1) (2015) 161–194.
[4] J. Christiaens, G. Vanden Berghe, Slack induction by string removals for vehicle routing problems, Transp. Sci. 54 (2) (2020) 417–433.
[5] J.-F. Cordeau, G. Laporte, The dial-a-ride problem: models and algorithms, Ann. Oper. Res. 153 (1) (2007) 29–46.
[6] L. Coslovich, R. Pesenti, W. Ukovich, A two-phase insertion technique of unexpected customers for a dynamic dial-a-ride problem, Eur. J. Oper. Res. 175 (3) (2006) 1605–1615.
[7] M. Diana, M.M. Dessouky, A new regret insertion heuristic for solving large-scale dial-a-ride problems with time windows, Transp. Res. B: Methodological 38 (6) (2004) 539–557.
[8] M. Gendreau, F. Guertin, J.-Y. Potvin, R. Séguin, Neighborhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries, Transp. Res. C: Emerg. Technol. 14 (3) (2006) 157–174.
[9] M. Gendreau, F. Guertin, J.-Y. Potvin, R. Séguin, Neighborhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries, Transp. Res. C: Emerg. Technol. 14 (3) (2006) 157–174.
[10] F. Glover, Ejection chains, reference structures and alternating path methods for traveling salesman problems, Discrete Appl. Math. 65 (1–3) (1996) 223–253.
[11] C.H. Häll, M. Högberg, J.T. Lundgren, A modeling system for simulation of dial-a-ride services, Public Transport 4 (1) (2012) 17–37.
[12] C.H. Häll, J.T. Lundgren, S. Voß, Evaluating the performance of a dial-a-ride service using simulation, Public Transport 7 (2) (2015) 139–157.
[13] C.H. Sin, W.Y. Szeto, Y.-H. Kuo, J.M.Y. Leung, M. Petering, T.W.H. Tou, A survey of dial-a-ride problems: literature review and recent developments, Transp. Res. B: Methodological (2018).
[14] A. Lois, A. Ziliaskopoulos, Online algorithm for dynamic dial a ride problem and its metrics, Transp. Res. Procedia 24 (2017) 377–384.
[15] Y. Luo, P. Schonfeld, A rejected-reinsertion heuristic for the static dial-a-ride problem, Transp. Res. B: Methodological 41 (7) (2007) 736–755.
[16] Y. Luo, P. Schonfeld, Online rejected-reinsertion heuristics for dynamic multivehicle dial-a-ride problem, Transp. Res. Rec.: J. Transp. Res. Board (2011) 59–67.
[17] M. Maalouf, C.A. MacKenzie, S. Radakrishnan, M. Court, A new fuzzy logic approach to capacitated dynamic dial-a-ride problem, Fuzzy Sets Syst. 255 (2014) 30–40.
[18] A. Núñez, C.E. Cortés, D. Sáez, B. De Schutter, M. Gendreau, Multiobjective model predictive control for dynamic pickup and delivery problems, Control Eng. Pract. 32 (2014) 73–86.
[19] S.N. Parragh, J.-F. Cordeau, K.F. Doerner, R.F. Hartl, Models and algorithms for the heterogeneous dial-a-ride problem with driver-related constraints, OR Spectr. 34 (3) (2012) 593–633.
[20] S.N. Parragh, K.F. Doerner, R.F. Hartl, Variable neighborhood search for the dial-a-ride problem, Comput. Oper. Res. 37 (6) (2010) 1129–1138.
[21] D. Pisinger, S. Ropke, A general heuristic for vehicle routing problems, Comput. Oper. Res. 34 (8) (2007) 2403–2435.
[22] S. Ropke, J.-F. Cordeau, G. Laporte, Models and branch-and-cut algorithms for pickup and delivery problems with time windows, Networks 49 (4) (2007) 258–272.
[23] S. Ropke, D. Pisinger, An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows, Transp. Sci. 40 (4) (2006) 455–472.
[24] D.O. Santos, E.C. Xavier, Taxi and ride sharing: a dynamic dial-a-ride problem with money as an incentive, Expert Syst. Appl. 42 (19) (2015) 6728–6737.
[25] H.R. Sayarshad, J.Y.J. Chow, A scalable non-myopic dynamic dial-a-ride and pricing problem, Transp. Res. B: Methodological 81 (2015) 539–554.
[26] M. Schilde, K.F. Doerner, R.F. Hartl, Metaheuristics for the dynamic stochastic dial-a-ride problem with expected return transports, Comput. Oper. Res. 38 (12) (2011) 1719–1730.
[27] M. Schilde, K.F. Doerner, R.F. Hartl, Integrating stochastic time-dependent travel speed in solution methods for the dynamic dial-a-ride problem, Eur. J. Oper. Res. 238 (1) (2014) 18–30.
[28] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, International Conference on Principles and Practice of Constraint Programming (1998) 417–431.
[29] H. Stephan, Empty Seats Traveling: Next-Generation Ridesharing and Its Potential to Mitigate Traffic and Emission Problems in the 21st Century. Technical Report, Research Center Bochum, 2007.
[30] P. Toth, D. Vigo, Vehicle Routing, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2014.
[31] S. Vallée, A. Oulamara, W.R. Cherif-Khettaf, Maximizing the number of served requests in an online shared transport system by solving a dynamic DARP, International Conference on Computational Logistics – ICCL20017, Lecture Note on Computer Science, vol. 10572 (2017) 64–78.
[32] S. Vallée, A. Oulamara, W.R. Cherif-Khettaf, Reinsertion algorithm based on destroy and repair operators for dynamic dial-a-ride problems, International Conference on Computational Science – ICCS 2019, Lecture Note on Computer Science, vol. 11536 (2019) 81–95.
[33] K.I. Wong, A.F. Han, C.W. Yuen, On dynamic demand responsive transport services with degree of dynamism, Transportmetrica A: Transport Sci. 10 (1) (2014) 55–73.

**Ammar Oulamara** is a professor in Computer Science at the University of Lorraine. He received M.Sc. in Computer Science from National Polytechnic Institute of Grenoble, a Ph.D. in Computer Science from Joseph Fourier University, Grenoble, in 2001, and Habilitation in Computer Science from the National Polytechnic Institute of Lorraine, Nancy, in 2009. His research interests lie mainly in production scheduling and logistics, combinatorial optimization, methods applied to manufacturing systems. He has published papers in many Operational Research and Industrial Engineering journals.