The Dial-A-Ride Problem with Costly Transfer Facilities

Jari Meevis
Eelco Timmerman

March 2019

Abstract

The Dial-A-Ride problem aims to find a set of routes that satisfies a set of passenger requests, where each passenger has a pickup point, a delivery point and a a maximum ride time. Every passenger is allowed to change vehicles once. This happens at specific locations called "transfer facilities". These transfer facilities are not active by default but have an associated opening cost. We apply an adaptive large neighbourhood search meta-heuristic to find solutions that minimize travel costs and facility opening costs and detail how we check the feasibility of a solution. The heuristic is tested on 195 artificial instances. We find solutions that (on average) halve the cost of the initial solution, which assigns a distinct vehicle to each request.

1 Introduction

In the pickup- and delivery problem with time windows (PDPTW), a set of passenger requests is given. A request consists of a pickup and a delivery. Both the pickup and the delivery have an associated time window. A vehicle is allowed to wait at a location until the start of the time window. It cannot arrive after the end of the time window. Passengers can share a vehicle, as long as the maximum capacity of that vehicle is not exceeded. The problem can either be static, when all requests are known in advance, or dynamic, when request can be added at any time. If one also constrains the maximum time a passenger can be on the road (called maximum ride time), the problem is called the Dial-A-Ride Problem (DARP). Both are extensions of the more general vehicle routing problem (VRP).
We study the static case of the DARP, where we allow passengers to transfer from one vehicle to another during its ride at a transfer facilities. This specification is also known as the Dial-A-Ride Problem with Transfers (DARPT). In contrast to the standard DARPT we consider a problem setting where opening a transfer facility has an associated cost. The objective is to minimize the total costs, which consist of the sum of the travel costs and the costs of the opened

transfer facilities.
The DARPT is computationally expensive. An algorithm to determine an optimal solution has been designed (Capelle, Cortés, Gendreau, Rey, and Rousseau, 2019). A more common solution is take a heuristic approach (Ropke and Pisinger (2006); Masson, Lehuédé, and Péton (2014, 2013).
Although there have been various researches conducted on similar problems as ours, none faces the exact same problem. Already in 2000, Nanry and Barnes (2000) developed a metaheuristic, in the form of a tabu search, to solve the PDPTW, where each request corresponded to a specific amount of goods to be transported. They tested their heuristic on instanced of up to 50 requests. Later on, Li and Lim (2003) improved the results on these same instances by performing a hybrid approach, combining simulated annealing with a tabu search. Parragh, Doerner, and Hartl (2010) took a different approach, proposing a variable neighbourhood search, which provided them with competitive results. The first to combine the PDP with transfers were Shang and Cuff (1996), where they considered every point in the problem as a possible point for a transfer. Furthermore, they only considered a transfer when they could not find a feasible location for a request without deploying a new vehicle. Then the first to combine the pickup and delivery with both transfers and a maximum ride time is the study conducted by Stein (1978). They did not, however, take into account capacity constraints or time windows. Mitrović-Minić and Laporte (2006) present a local search algorithm, with uncapacitated vehicles, where they use the Manhattan-distance.
Ropke and Pisinger (2006) introduced Adaptive Large Neighbourhood Search (ALNS), an improvement over the earlier Large Neighbourhoud Search metaheuristic (LNS). LNS repeatedly destroys and repairs an existing solution to arrive at new solutions. ALNS is characterized by its ability to adapt to the problem at hand by changing the selection of the repair and destroy operators based on performance in previous iterations. This way, it is possible to assign more weight to operators that yield good solutions in a certain problem instance. They tested their heuristic on more than 350 instances with up to 500 requests each and were able to improve the best solutions from literature for more than 50 percent of the cases. Another example of the implementation of an ALNS is provided by Masson et al. (2013). In contrast to Ropke and Pisinger (2006), they did include transfers, tested their method on both real life and generated instances and found out that this could improve their solutions with up to 8 percent. Finally, we note that Masson et al. (2014) applied the ALNS heuristic to their DARPT setting with maximum ride and route constraints and showed good performance on benchmark instances.
In this paper, we will adapt the ALNS meta-heuristic as given by Masson et al. (2014) to our problem setting, which is a modification of their original DARPT. The two differences between their work and ours are as follows. Firstly, in our problem setting, opening a transfer facility incurs a cost. Secondly, we consider only a maximum ride time constraint, and not a maximum route time constraint.

2 Problem description

Let R be a set of passenger requests, where each request r corresponds to the transportation of one passenger from a pickup point p_r to a delivery point d_r. Let P and D denote the set of pickup and delivery nodes, respectively. There are time windows $[e_i, l_i]$, associated with each pickup and delivery node i, where e_i and l_i represent the earliest and latest times at which service (i.e. loading or unloading) can begin at node i. A vehicle is allowed to wait at a node in order to serve it within its time window. Each passenger request has a maximum ride time L_r, which is the maximum time allowed between the end of loading at p_r and the beginning of unloading d_r.

In addition to pickup and delivery nodes, there are also transfer nodes in the network, where passengers can be dropped off by a vehicle to be later picked up by another vehicle. Note that (i) transfers can only happen at a transfer facility, (ii) the temporal dimension in synchronizing transfers cannot be ignored; i.e. a passenger can be picked up from a transfer node only after they are dropped off, and (iii) passengers can be transferred from one vehicle to another at most once. The number and locations of transfer facilities has to be determined. T denotes the set of candidate locations to open a transfer facility. There is a cost, f_t, associated with opening a facility at each candidate site t. It is possible to open no transfer facilities, one facility, or more.

Travel distance from node i to node j is given by θ_{ij}. Travel cost is linearly proportional to travel distance, i.e. travel cost is $c_{ij} = c_0 \times \theta_{ij}, c_0 \in \mathbb{R}_+$ being a constant. Service duration at node i is denoted with s_i. Service corresponds to loading (in pickup and transfer nodes) or unloading (in delivery and transfer nodes). When a vehicle performs both loading and unloading at node i, each of these two operations takes s_i units of time (for a total of $2s_i$). Similarly, when multiple loading/unloading operations are performed at node i, each of these operations takes s_i units of time.

Each vehicle traverses a single route k. The set of routes is K. O denotes the set of starting and ending depots for vehicles, and each vehicle can start and end at any depot (they do not need to start and end at the same depot). The carrier has homogeneous vehicles with capacity Q; each passenger occupies 1 unit of vehicle capacity. Vehicle capacity cannot be exceeded at any point in time. There is an unlimited number of vehicles.

Let $G = (V, A)$ denote the network representation of this problem, where V is the set of nodes and A is the set of arcs. The travel time θ_{ij} and cost c_{ij} are associated with each arc $(i, j) \in A, i \neq j$. $V = P \cup D \cup T \cup O$ includes all pickup, delivery, transfer and depot nodes. The arcs Given that there are n passenger requests, m candidate transfer locations and q depots, the set of nodes is indexed as $V = \{1, \ldots, n, n+1, \ldots, 2n, 2n+1, \ldots, 2n+m, 2n+m+1, \ldots, 2n+m+q\}$, where nodes $1, \ldots, n$ and $n+1, \ldots, 2n$ represent pickup, resp. delivery lo-

cations, nodes $2n+1, \ldots, 2n+m$ represent candidate transfer locations, and nodes $2n+m+1, \ldots, 2n+m+q$ represent the depots.

The object is to minimize the total cost of travel and opening transfer facilities.

The symbols used in this description (and other symbols used later) are summarized in Table 1.

3 Data structure

The problem description describes what data determines a problem instance. The focus of this section will therefore be on the data structure of the solution.

3.1 Main solution structure

A solution is defined as a set of routes K that service all requests $r \in R$ and a set of open transfer facilities. Each route satisfies the associated constraints (e.g. maximum ride time, L_r; time window, $[e_i, l_i]$ and various timing and precedence constraints as detailed in Section 2; etc.). Feasibility can in principle be determined from the service time epochs and the order of the nodes. However, this is computationally expensive and quite difficult to check conceptually. To remedy this, some data structures that make this more convenient are detailed in Section 3.2.

3.1.1 Transfer facilities

The list of open (usable) transfer facilities is called T^o, while the list of closed (unusable) transfer candidates is called T^c. Note that $T = T^o \cup T^c$. Opening of a transfer candidate $t \in T^c$ is modelled by removing it from T^c, adding it to T^o and adding cost f_t to the cost of the solution. To close transfer facility $t \in T^o$, all requests that make use of this facility are removed (for later reinsertion), f_t is subtracted from the cost of the solution and the facility is moved from T^o to T^c.

3.1.2 Routes

A route is an ordered list of nodes that are visited by some vehicle k. The order of the nodes determines the cost of the route (since the cost of a route is directly proportional to the travel distance). The nodes that are visited do not have a one to one relationship with the locations as detailed in Section 2, since some nodes are visited more than once or not at all.

To keep track of the path followed by transferred requests, we adopt the same approach as Masson et al. (2014). For each request $r \in R$ transferred from route $k_1 \in K$ to route $k_2 \in K$ at transfer point $t \in T^o$, t is duplicated in an inbound

transfer vertex $\tau^-_{t,r}$ and an outbound transfer vertex $\tau^+_{t,r}$ and stored in set T^- and T^+ respectively. The outbound transfer node $\tau^+_{t,r}$ represents the loading of request r at transfer point t, while $\tau^-_{t,r}$ represents the unloading of request r at (the same) transfer point t.

To model depots in routes, we define vector O' with elements o'_i, $i \in V \setminus O$. $o'_i = \arg\min_{j \in O} \theta_{ij}$. Since there are no time or capacity constraints on the depots, we can minimize travel costs by always departing from (arriving at) the depot nearest to the first (last) visited node in route k. This node can readily be found in O' and it becomes unnecessary to explicitly add starting and ending depots to a route. The cost calculation of a route becomes slightly more involved, since the costs of travelling from the starting depot and to the ending depot need to be calculated separately.

To model the other locations, no adjustments are necessary since any pickup point $p_r \in P$ or delivery point $d_r \in D$ is already associated with request r, and (by definition) with a loading or unloading operation. Moreover, it can be visited by at most one vehicle so duplication is not necessary.

In addition to this, for each node $i \in \mathrm{T}^+ \cup \mathrm{T}^- \cup P \cup D$, we store the arrival, a_i, the start of service, H_i, and the number of passengers in the vehicle after departure from the node, q_i.

3.2 Convenience structures

Each request r has an associated helper structure r' that keeps track of where the request is handled. This structure contains the pickup node, p_r, the delivery node d_r and optionally the transfer nodes $\tau^+_{t,r}$ and $\tau^-_{t,r}$ for some $t \in T^o$.
This helper structure allows us the check the maximum ride time constraint and precedence constraints of a request in constant time by simply comparing the service epochs of nodes.

4 ALNS format

This section describes how Adaptive Large Neighbourhood Search has been applied to the DARPT. An extensive description of the ALNS scheme and its application to a similar problem can be found in Ropke and Pisinger (2006). The ALNS includes an adaptive layer which automatically adjusts certain parameters. These adjustments are based on the performance of various subsections (called operators) of the algorithm and strive to adapt the heuristic to the problem instance at hand.

Symbol	Set	Description
Problem related symbols		
r	R	Request
t	T	Candidate transfer location
o	O	Depot location
p_r	P	Pickup location of request r
d_r	D	Delivery location of request r
L_r		Maximum ride time of request r
e_i		Earliest start service time of node $i \in P \cup D$
l_i		Latest start of service time of node $i \in P \cup D$
f_t		Cost of opening candidate transfer location t
s_i		Service duration at node $i \in P \cup D \cup T$
$\theta_{i,j}$		The distance between node i and j
Solution related symbols		
s		(Partial) solution
k	K	Route
o_k	O	Starting depot of route k
o'_k	O	Ending depot of route k
$\tau^-_{t,r}$	T^-	Inbound transfer point (unloading of request r at transfer point t)
$\tau^+_{t,r}$	T^+	Outbound transfer point (reloading of request r at transfer point t)
a_i		Arrival epoch at node $i \in \mathrm{T}^+ \cup \mathrm{T}^- \cup P \cup D$
H_i		Start of service epoch at node $i \in \mathrm{T}^+ \cup \mathrm{T}^- \cup P \cup D$
q_i		Amount of passengers at node $i \in \mathrm{T}^+ \cup \mathrm{T}^- \cup P \cup D$
$\sigma(i)$		The successor of node $i \in \mathrm{T}^+ \cup \mathrm{T}^- \cup P \cup D$
$\rho(i)$		The predecessor of node $i \in \mathrm{T}^+ \cup \mathrm{T}^- \cup P \cup D$
$\bar{U}$		The planned number of requests to destroy
	U	Set of destroyed (unplanned) requests
$G(V, E)$		Directed graph with vertices V and edges E

Table 1: An overview of all symbols used in the paper. The second column denotes the set to which each instance of the symbol belongs, if applicable. E.g. the set R contains all requests r. The table is split in to two parts. The upper part shows the symbols that are used (mainly) in the problem description, while the symbols in the lower part are mainly used to describe the solution.

4.1 Main scheme

The main principle of the ALNS is to iteratively destroy and repair a solution. To do this, it uses various operators that either remove requests from a feasible solution or insert unplanned requests into a partial solution. These operators are the destroy and repair operators respectively. A high level overview of the heuristic is shown in Figure 1.

The algorithm begins by creating an initial feasible solution. Our approach

```
Require: InitialSolution
    BestSolution = InitialSolution
    CurrentSolution = InitialSolution
    while the termination criterion is not satisfied do
        Selection of a Destroy and a Repair operator
            according to past performances
        S = CurrentSolution
        S = Destroy(S)
        S = Repair(S)
        if S < BestSolution then
            BestSolution = S
            CurrentSolution = S
        else if AcceptationCriterion(S, CurrentSolution) then
            CurrentSolution = S
        end if
    end while
    return BestSolution
```

Figure 1: Pseudocode for an ALNS, taken from Masson et al. (2014)

is to let each request be handled by a separate vehicle and disregard transfers. At each iteration, the algorithm selects a destroy and a repair operator from the pool of possible operators. This selection is based on the performance of these operators in previous iterations. After this selection, the destroy operator is applied. The destroy operator removes one or more requests from the current solution and stores them in the bank of unplanned requests. This results in a partial solution. Afterwards, the repair operator is applied. The repair operator inserts the unplanned requests into the partial solution. If the resulting solution is better than the current solution, it is accepted. If it is worse than the current solution, an acceptation criterion is used to determine whether the solution will be accepted. If it is equal to the current solution, it is rejected. The acceptation criterion used is simulated annealing. Finally, the algorithm returns the best solution it has seen.

The operators are selected with a roulette wheel selection principle. The chance of selection is proportional to the weight of the operator. The weight calculations are similar to those of Ropke and Pisinger (2006): after initialization of

the algorithm, all weights are equal. For each segment of 10 iterations, the score of the operators is updated in the same manner as Masson et al. (2014): a new best solution increases the score by 33, a new improving solution increases the score by 20 and a new accepted solution increases the score by 15. The operator weights are updated every 10 iterations using exponential smoothing with a smoothing factor of 0.01. These values are 10% of the values that Masson et al. (2014) use. The reasoning is that we want to adapt the weights more often since we use less iterations than them.
Using the simulated annealing acceptance criterion, the chance of accepting a worse solution is given by

$$accept(s, s', t) = \exp\left(\frac{c(s) - c(s')}{T(t)}\right)$$

where s is the current solution, $s' < s$ is the new solution, $c(\cdot)$ is the cost function, t is the time (or the current iteration) and $T(t)$ is the temperature, a decreasing function that ensures that the chance that a worse solution is accepted decreases as time passes. We define $T(t) = T_0 c^t$, where T_0 is the initial temperature and $c \in (0, 1)$ is the cooling rate. We select the initial temperature T_0 such that the chance of accepting a solution 5% worse than the initial solution (s_0) is 50%, i.e. $T_0 = -0.05 \times s_0 / \ln(0.5) \approx 0.0721 s_0$. Additionally, we set the maximum number of iterations to 2000, and set the cooling rate such that the temperature is 1 after 2000 iterations: $c = \left(\frac{1}{s_0}\right)^{\frac{1}{2000}}$. We abort the algorithm if no solutions are accepted for 100 iterations.

In accordance with Ropke and Pisinger (2006), we apply noise to the objective function in an adaptive manner to prevent the myopic nature of most operators from disregarding a lot of neighbourhoods. We calculate the noise as follows: first we determine whether to use noise using the same adaptive mechanism as in operator selection. Then, we select the amount of noise to be applied randomly from the interval $[-maxN, maxN]$, with $maxN = \eta \cdot \max_{i,j \in V}(\theta_{i,j})$. Following Ropke and Pisinger (2006), we have selected $\eta = 0.025$.

4.2 Destroy operators

We use 4 distinct destroy operators. The operators remove one or more requests from a route. The number of requests to be removed ($\bar{U}$) is defined as

$$\begin{aligned}
\bar{U} &= f(i) \times \bar{R} + \bar{U_0}, \qquad \text{where} \\
\bar{R} &= 0.05 \times |R|, \\
\bar{U_0} &\sim \mathcal{U}(1, \bar{R} + 1), \\
f(i) &= \begin{cases} 0 & \text{if } i \leq 25 \\ i/25 + 1 & \text{if } i > 25 \end{cases},
\end{aligned}$$

where i is the number of iterations since the last accepted solution. The idea here is that we try a relatively low-impact destroy operator until we have not

found an accepted solution for 25 iterations. We then increase the impact of the destroy operator until we have found an accepted solution or the algorithm aborts (after 100 unaccepted solutions). One exception is detailed below. The set of requests to be removed during the current iteration is called U.
The destroy operators concerning transfers have been taken from Masson et al. (2014), while the general destroy operators have been adapted from Ropke and Pisinger (2006).

Transfer point removal: this operator removes all requests r that make use of a given transfer point ($\exists \tau^{+}_{r,t} \Rightarrow r \in U$). The transfer point is randomly selected from the set of open transfer points. An open transfer points serves at least one request. This destroy operator can remove more than $\bar{R}$ requests if more are routed through the transfer point. The idea is to remove all requests that make use of a given transfer point to save us the cost of opening this transfer point and re-route them in another way.

Pickup/Delivery cluster removal: this operator removes all requests that have either pickup or delivery points that are located closely together. The idea is that these requests might benefit from being re-routed through a common transfer points, since their pickups or deliveries can be serviced by a single vehicle. The operator works by selecting a random root node, which can be either a pickup or a delivery node and sorting all other nodes of the same type based on distance to the root node. It will then iteratively select the next request to destroy by generating a random number $y \in (0, 1)$, calculating the index of the request to be removed as $index = \lfloor y^p \times (|R| - |U|) \rfloor$, where p is a parameter that determines how deterministic the selection is. A higher p corresponds to lower randomness. The request associated with the node at that index will be added to U and its node will be removed from the list of sorted nodes. This repeats until $|U| = \bar{U}$. The value of parameter p has been set to 9, as in Ropke and Pisinger (2006).

The general destroy operators are as follows.

Random removal: this operator randomly selects requests to be removed among the planned requests.

Shaw removal: this operator aims to remove requests that are alike. The idea is similar to that of the Pickup/Delivery cluster removal, but not specifically adapted to transfers. It randomly selects a root request ($r_0 \in R$) to remove and calculates the relatedness measure of this request to all other requests. We have adapted the relatedness measure that Ropke and Pisinger (2006) proposed because in our problem setting to only consider time, distance and capacity similarity and not which-vehicle-can-serve-which-request similarity, since in our setting all vehicles can serve all requests. The relatedness is given by

$$R(i, j) = \Omega(\theta_{p_i,p_j} + \theta_{d_i,d_j}) + \Psi(|H_{p_i} - H_{p_j}| + |H_{d_i} - H_{d_j}|) + \Phi(|q_i - q_j|)$$

where Ω, Ψ and Φ are weighing factors and $i, j \in R$. A lower $R(i, j)$ corresponds to more relatedness between two requests. We sort all requests on relatedness to the root node (i.e. in increasing $R(i, r_0)$). The selection of requests to be removed is then the same as in the Pickup/Delivery cluster removal. For weights, we have selected $\Omega = \Psi = \Phi = 1$.

4.2.1 Destruction principle

The actual removal of a request is fairly straightforward. We find the nodes that need to be removed by looking at the convenience structure r' for each $r \in U$. A linear search over all nodes in all routes allows us to remove the nodes. After removing the first node i, we update the subsequent node by recalculating the arrival time using $a_{\sigma(i)} = H_{\rho(i)} + s_{\rho(i)} + \theta_{\rho(i),\sigma(i)}$. During iteration over the rest of nodes of the route, we update the number of passengers at each node by setting $q_j = q_j - 1$, where j is any node after the first removal but before the second removal in a route. The idea here is that the first removal is always a pickup, so the number of passengers at each subsequent node is 1 less than it currently is. After the second removal, the net influence on passengers carried is 0 (one pickup and one delivery). A route containing two nodes of which any is part of the nodes to be removed, is removed from the solution in its entirety. This also works for transfers. Note that the feasibility of a route is unchanged, since the start of service epochs are unchanged and the number of passengers has decreased or remains unchanged.

4.3 Repair operators

A repair operator takes a partial solution s and a set of unhandled requests U as input, inserts the unhandled requests back into the solution and then returns a feasible solution. Each request is evaluated by iterating over all possible routes and all possible insertion locations (this includes 'inserting' it in a new route, which implies that we can always find a feasible insertion). It checks the feasibility of an insertion (this is quite difficult, see Section 5) and the associated insertion cost. After is has selected the insertions to perform, it finally inserts the requests in the current partial solution and calculates the start of service epochs.
We have implemented 3 repair operators, adapted from Masson et al. (2014). We first discuss the main adaptation we have performed, which allows us to open transfer locations.

Main adaptation: Unlike Masson et al. (2014), in our problem setter a transfer point can only be used after the cost of opening it has been incurred. It must thus be a deliberate choice if and when to open a transfer point. A heuristic to determine whether and which transfer point to open is as follows: at the start of each operator that uses transfers, we open a transfer with probability 70% if there are already transfers open, probability 100% if there are no transfers open and probability 0% if there are no closed transfers. At the end, we remove all

transfers that are not used. To select which transfer will be opened, we calculate the product of its cost and potential travel cost (i.e. the sum of distances to nodes of interest):

$$WC(t) = f_t \times \sum_{r \in U} (\theta_{t,d_r} + \theta_{t,p_r}), \qquad t \in T^c$$

and rank the transfers in ascending order of $WC(t)$. The idea here is that we want to find a balance between the cost of opening a certain transfer and the distance it is located from possible nodes of interest. The formula above ensures that both are weighted and that the function is both increasing in cost of opening and in potential travel cost. We then randomly select $y \in (0, 1)$ and calculate the index of the transfer to open as $index = \lfloor y^p \times |T^c| \rfloor$, where p is a (constant) parameter. We have selected $p = 9$. Both the formula and the value of p are inspired by or taken from Ropke and Pisinger (2006).

Best insertion with transfer: For each unplanned request the best insertion without considering transfers is evaluated. Then, for each open transfer point $t \in T^o$ and each unplanned request (p_i, d_i), we consider two different transfer insertions: (1) where we select the best insertion of (p_i, t) and take it as given before evaluating the best insertion of (t, d_i) and (2) where we do the same in reverse: we consider (t, d_i) and take it as given before evaluating the best insertion of (p_i, t). The costs of these insertions are simply the sum of costs of (p_j, t) and (t, d_j) for both options. Finally, the best among the three insertions is performed. This operator considers the requests to be inserted one by one and simply inserts the current request at its best location. It updates the partial solution s and start considering the next request.

Transfer first: We consider only transfer insertions in the same manner as *Best insertion with transfer*. Only if no transfer insertion is feasible do we consider an insertion without a transfer. After all requests have been inserted, we iteratively remove requests that make use of a transfer and compare their current insertion with an insertion without a transfer. We select the best of these two options as our final insertion. This operator considers the requests one by one, in precisely the same manner as *Best insertion with transfer*.

For our general (non-transfer) repair, we have the following operator:

Best insertion: At each iteration we calculate the best insertion for each unplanned request $r \in U$ and perform the best insertion, setting $U = U \setminus r$. This includes inserting the request in a new route. The operator stops when all requests have been planned (i.e. $|U| = 0$). Note that this operator considers all requests together (i.e. not one by one).

5 Feasibility

A solution of the DARPT is feasible if three types of constraints are satisfied: the capacity constraints, the temporal/precedence constraints and the transfer utilization constraints. Verifying capacity constraints is straightforward. Additionally, verifying that only opened transfer locations are used for transfer is also straightforward, as is ensuring that closed transfer locations are not used in the first place. These will briefly be discussed in Section 5.3. However, verification of temporal constraints is less straightforward and will therefore be described in detail. Most of it is similar to Masson et al. (2014) and differences will be explicitly mentioned.
A solution of any vehicle routing problem is only feasible if all routes of the solution are feasible. In the DARPT, routes are connected through transfer points. Because of this, any change in one route can influence any number of other routes. These influences can in turn be propagated to even more routes via the transfer points, ad infinitum. The need for synchronization at transfer points makes the feasibility of the routes interdependent, which makes independent verification of the feasibility of routes impossible.
In this section, we show that a solution of the DARPT can be modelled as a Simple Temporal Problem (STP) and describe how to solve this problem. Specifically, we detail how to check the feasibility of a solution and how to generate a feasible solution. Millions of feasibility checks are performed during the execution of ALNS, so it is important to make these checks as efficient as possible. We will adopt the same approach as Masson et al. (2014) and use necessary and sufficient conditions to speed up this process.

5.1 Formulation of the temporal feasibility problem

The key issue that complicates feasibility checking is that the use of transfers can create temporal constraints between different routes. A change in Route A can influence the time at which a passenger is unloaded at a transfer point. As a result, this may change the time at which that passenger can be picked up from the same transfer point by the vehicle that serves Route B. These changes can propagate throughout the entire set of routes and change the feasibility of the solution, even though the feasibility of the first route is unchanged.

The following description is taken almost directly from Masson et al. (2014). The problem under consideration does not include maximum route duration, so constraints pertaining to it have been omitted.

The set of service times at each vertex in a solution is called the schedule of the solution. The service time at vertex $i \in V$ is denoted H_i. The set of requests transferred at transfer point $\tau \in \mathrm{T}^o$ is represented by R_τ. The solution to the temporal part of the problem must then satisfy the following constraints (where

$\sigma(i)$, $\rho(i)$ denote the successor, resp. predecessor vertex to some vertex $i \in V$).

$$H_{\sigma(i)} \geq H_i + s_i + \theta_{i,\sigma(i)}, \qquad \forall i \in V \setminus O \tag{1}$$
$$H_{t^+_{\tau,r}} \geq H_{t^-_{\tau,r}} + s_{t^-_{\tau,r}}, \qquad \forall \tau \in \mathrm{T}^o, r \in R_\tau \tag{2}$$
$$e_i \leq H_i \leq l_i, \qquad \forall i \in V \tag{3}$$
$$H_{d_r} - (H_{p_r} + s_{p_r}) \leq L_r, \qquad \forall r \in R \tag{4}$$
$$H_i \geq 0, \qquad \forall i \in V \tag{5}$$

Constraints (1) establish the that the start of service time at the successor of vertex $i \in V$ must be greater then or equal to the service time at vertex i plus the service duration s_i and the travel time between these two vertices. Constraints (1) ensure that each request $r \in R$ has finished unloading at transfer $\tau \in \mathrm{T}^o$ before it can begin loading. Constraints (3) guarantee that requests are serviced within their time windows, while constraints (4) ensures that the maximum ride time is respected. Finally, constraint (5) establishes a reference point in time, 0, and ensures that no operations can begin before that.

Following the argumentation in Masson et al. (2014), we note that we can model the problem described above as a Simple Temporal Problem (STP) (Dechter, Meiri, and Pearl, 1991). The STP representation of the temporal constraints of Vehicle Scheduling Problem is a directed graph where each vertex denotes a point in time (H_i in the original problem) and each constraint on H_i is represented by an arc. To ensure that that time windows can be represented by arcs, we introduce the dummy vertex 0, which represents the beginning of the time window (i.e. $H_0 = 0$). The set of constraints can thus be represented by equations (6)-(12):

$$H_i - H_{\sigma(i)} \leq -s_i - \theta_{i,\sigma(i)}, \qquad \forall i \in V \setminus O \tag{6}$$
$$H_{t^-_{\tau,r}} - H_{t^+_{\tau,r}} \leq -s_{t^-_{\tau,r}}, \qquad \forall \tau \in \mathrm{T}^o, r \in R_\tau \tag{7}$$
$$H_0 - H_i \leq -e_i, \qquad \forall i \in V \tag{8}$$
$$H_i - H_0 \leq l_i, \qquad \forall i \in V \tag{9}$$
$$H_{d_r} - H_{p_r} \leq L_r - s_{p_r}, \qquad \forall r \in R \tag{10}$$
$$H_i \geq 0, \qquad \forall i \in V \tag{11}$$
$$H_0 = 0, \tag{12}$$

where constraints (6) - (7) are equivalent to constraints (1) - (2), constraints (8)-(9) are equivalent to constraint (3) (recall that H_0 is a dummy vertex with value 0). Constraints (10) are equivalent to constraints (4) and constraints (11) and (12) introduce and define the beginning of the planning horizon. This problem is a simple temporal problem, and is equivalent to constraints (1)-(5). It is important to note that the left hand side of equations (6)-(10) define two vertices of the arc and its direction, while the right hand side defines its weight (or distance). An example is shown in Figure 2.

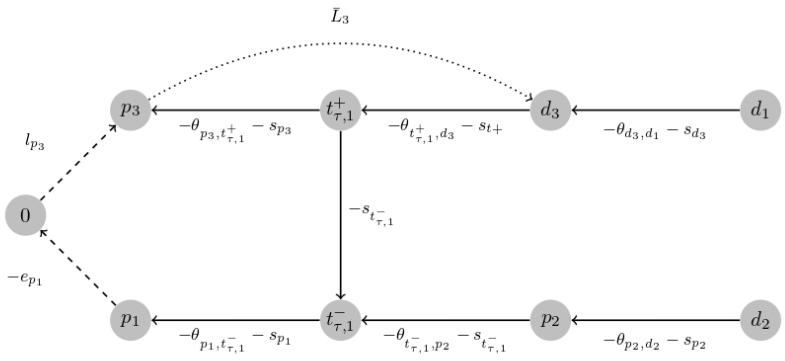

Figure 2: a STP distance graph where only a few constraints are shown. Taken from Masson et al. (2014).

5.2 Solving the feasibility problem

In accordance with Masson et al. (2014), we note that evaluating route feasibility for the DARPT is equivalent to proving the consistency of the corresponding STP. The STP can be represented by a directed graph $G' = (V', A')$ where each vertex represents a point in time (in our case, the start of service times) and each arc represents a constraint. The set of constraints that needs to be satisfied for a solution to be feasible is given by equations (6)-(12). Each constraint can be represented by an arc in the STP. Constraint (1) and (2) ensure that actions cannot begin before the previous action has started, finished and transportation has taken place. Constraint (3) and (4) ensure that the time windows are respected. Constraint (5) ensures that the maximum ride time is satisfied and the final constraints ensure that a reference point in time exists.
It has been proven Dechter et al. (1991) that if a STP represented by directed graph G' contains a negative cycle, it is inconsistent and thus infeasible. If it is feasible, a valid set of start-of-service times is given by the s e (H_0) to all other nodes (i.e. $\lambda_{0,i} \, \forall i \in V$), and vice versa (i.e. $\lambda_{i,0} \, \forall i \in V$). The BFM algorithm creates a shortest path tree by keeping track of via what node it reached the current node. To check whether the graph contains a negative cycle, we do the following: (1) for each node in the tree, we keep track of its parent node (2) for each node in the tree, we follow the path defined by its parent and their parents until we find the source node. If we find the node where we started while traversing the path, we have found a negative cycle and the solution is not consistent and thus not feasible. We amortize this negative cycle detection algorithm (of running time $O(|V|)$) so that we only check for negative cycles after scanning N nodes using the BFM algorithm. This ensures we do not change the total worst-case time complexity which is $O(|V|^2)$.
Since this algorithm is called often and is costly in terms of CPU time, we define some necessary and sufficient conditions that allow us to determine the

feasibility of an insertion without resorting to the full BFM with negative cycle detection algorithm. The necessary conditions are used to find infeasible insertions quickly, while the sufficient conditions establish feasibility efficiently.

5.2.1 Necessary conditions

In order to identify infeasible insertions in a feasible partial solution, we propose two necessary conditions. In the first necessary condition, we aim to quickly stop trying insertions that are not feasible because the time window of the inserted node cannot be managed. In the second, we relax the maximum ride time constraint and verify whether time window constraints of subsequent nodes can be satisfied.

Tightening time windows: Time windows (i.e. $[e_i, l_i]$) can be tightened according to Dechter et al. (1991). Following Masson et al. (2014), let $\lambda_{i,j}$ be the length of the shortest path between two vertices $i \in V'$ and $j \in V'$ in the distance graph G' of a consistent instance of the STP. A feasible solution for H_i must then have $H_i \in [-\lambda_{i,0}, \lambda_{0,i}]$. Any operator starts from a feasible partial solution s. If the partial solution has been proven feasible by the BFM algorithm, both $\lambda_{i,0}$ and $\lambda_{0,i}$ have been calculated for all $i \in V_s$ (where V_s is the set of vertices associated with partial solution s). If it has been proven feasible by a sufficient condition, applying the BFM algorithm will again yield the tightened time windows. Note that this does not mean that the sufficient conditions are useless: most operators insert one of the best feasible insertions, which means that feasibility has to be checked often but the time windows might only have to be calculated once. If the partial solution was created by destroying one or more requests, the solution is always feasible but the time windows have to be recalculated. This means that we calculate time windows after each modification of a (partial) solution.

NC0: Simple time windows The default necessary conditions (NC0) aims to ignore insertions that are infeasible due to their own time window constraints. Whenever we try an insertion into a route at a certain location, we verify that we the arrival time at the inserted node i (given by $H_\rho i + s_\rho i + \theta_{\rho(i),i}$) is earlier than the latest possible start of service time (l_i). If is not, we do not have to check later insertions in the same route. When we are inserting transfers, we verify that the earliest possible arrival time at the delivery node of the inserted request j still satisfies the time window of the delivery (i.e. $s_t + \theta_{t,d_j} \leq l_d$, t the transfer)

NC1: Relaxation as a PDPT: The second necessary condition is based on the relaxation of maximum ride time constraints. We take a simple linear time approach. After an insertion, the time windows $[-\lambda_{i,0}, \lambda_{0,i}]$ must still be satisfied for all subsequent nodes, since an insertion can only lengthen the shortest paths. As a result, the tightened time windows should be satisfied

after any request insertion. To check this, we verify that the time window in all subsequent nodes in the route is satisfied. If the insertion is such that arrival at some node j after the insertion is before $-\lambda_{j,0}$, all subsequent time windows are automatically satisfied and we do not need to check the rest of the time windows.

5.2.2 Sufficient conditions

We have implemented one sufficient condition that verifies that the relaxation in NC1 yields a valid solution when the constraints are added again.

SC1: No change after insertion According to Dechter et al. (1991), a consistent schedule H^s is given by $H^s_i = -\lambda_{i,0}$ for all vertices $i \in V_s$. Let j be an unrouted request made up of vertices p_j and d_j and let s' be the solution resulting from insertion of j into partial solution s that satisfies NC1. s' may or may not be feasible. s' is feasible if the maximum ride time constraint of j is reintroduced and satisfied, and the timing of the rest of the route is unchanged. Following Masson et al. (2014), this can be checked in constant time with the following set of constraints:

$$
\begin{aligned}
H^{s'}_{d_j} &= \max(e_{d_j}, -\lambda^s_{\rho(d_j),0} + s_{\rho(d_j)} + \theta_{\rho(d_j),d_j}) \\
H^{s'}_{p_j} &= \max(e_{p_j}, -\lambda^s_{\rho(p_j),0} + s_{\rho(p_j)} + \theta_{\rho(p_j),p_j}, H^{s'}_{d_j} - L_j - s_{p_j}) \\
H^{s'}_{p_j} &+ s_{p_j} + \theta_{p_j,\sigma(p_j)} \leq -\lambda^s_{\sigma(p_j),0} \\
H^{s'}_{d_j} &+ s_{d_j} + \theta_{d_j,\sigma(d_j)} \leq -\lambda^s_{\sigma(d_j),0}
\end{aligned}
$$

The first two constraints ensure the maximum ride time is satisfied while the second two constraints ensure the rest of schedule is unchanged. We have opted not to implement this sufficient condition for transfers.

5.3 Capacity and (closed) transfers

Verifying that capacity constraints are satisfied can be done in linear time by checking each node in the route and verifying that the number of passengers after leaving the node is not more than the maximum allowed capacity Q. Additionally, ensuring that only open transfer points are used for transfers is done by the repair and destroy operators themselves. The repair operators that include transfers do not consider closed transfer points. The destroy operators remove all requests transferred at a certain transfer before closing that point. This ensures that only open transfer points can be used for an insertion. No further verification is necessary.

6 Results

We have tested our method on a standard benchmark set of 195 instances, the results of which can be found in a separate file. We find an average reduction of

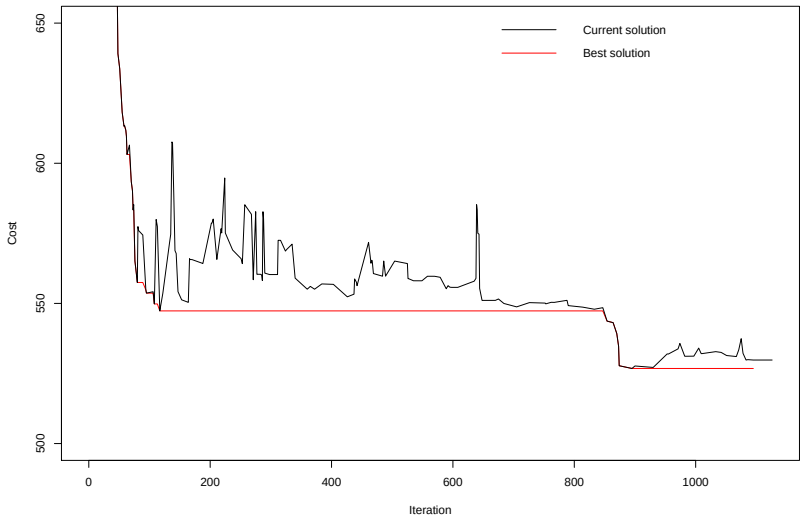

Figure 3: Sample paths of the cost of the current and best solution for instance 60

approximately 50% over the cost of the initial solution, which assigns a separate vehicle to each request.
Sample paths of the cost of the current and best solution for a small, medium and large instance are shown in Figure 3-5. These sample paths illustrate that the larger the instance, the longer it takes to converge to a local optimum (but also that it might be harder to get stuck in it in the first place).
In Figure 6 we show the evolution of destroy operator weights for instance 145. It is interesting to note that at first, all operators yield accepted solutions and their weights increase. After about 1200 iterations, it becomes more difficult to find solutions that are accepted, because the temperature decreases and most repair operators result in a worse solution. This can be seen in the fact that all weight start decreasing. The transfer removal operator starts decreasing even earlier: the requests that can make the most effective use of transfers are already making use of them, so destroying these cannot decrease the cost further. We note that removing the *Transfer removal* operator does not significantly influence the quality of the resulting solutions, but also does not slow down the algorithm, so we see no reason to remove it.

7 Conclusion and discussion

In this paper, we applied an ALNS to solve the Dial-A-Ride Problem with Transfers. We based the ALNS format mostly on research done by Ropke and

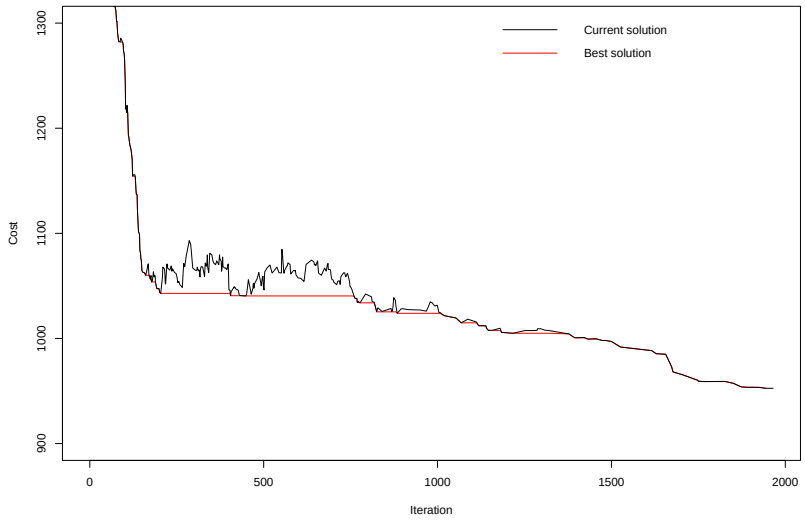

Figure 4: Sample paths of the cost of the current and best solution for instance 120

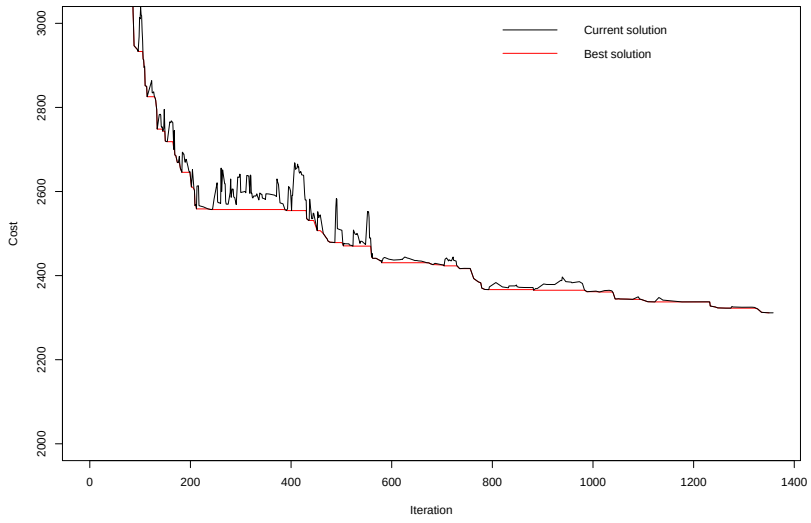

Figure 5: Sample paths of the cost of the current and best solution for instance 180

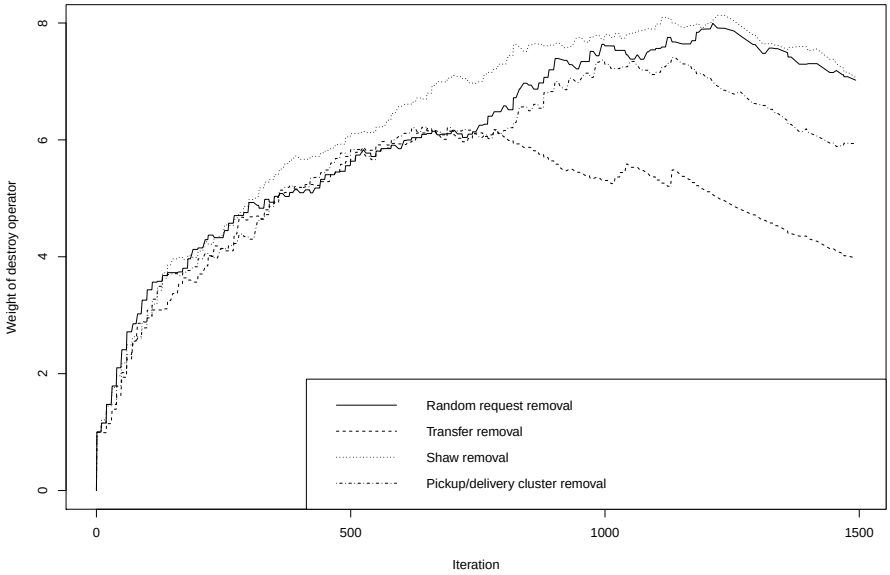

Figure 6: The evolution of destroy operator weights for instance 145

Pisinger (2006), and operators and feasibility checking mostly on Masson et al. (2014). We introduced a new decision mechanism for opening a transfer facility that allows us to open and close facilities at will. We were able to reduce the costs of our initial feasible solution, where each request was handled by a different vehicle, with approximately 50%. Furthermore, we found that certain instances significantly benefit from the possibility of transfers (e.g. 145, 173, 174).

To improve solution quality, it is possible to add operators that operate in a slightly less myopic way than the operators we currently have implemented. For example, we have not implemented a regret heuristic, nor have we made the opening of transfer facilities adaptive (or indeed, provided more than one heuristic to determine what facility to open). Implementing these two changes will increase the randomness of our model, which may in turn allow us to escape more easily from local minima. However, without a comparison to existing problems with known lower bounds or optimal solutions, we cannot begin to estimate the actual impact of these changes. This in turn leads us to the following: it is impossible to verify our solution quality, since we have no lower bound or other comparable instances. A lower bound could be found by solving the ILP for smaller instances, but this is probably not tractable for the larger instances (> 50 requests). Using our approach, we are able to solve the instances used by Masson et al. (2014) by setting the cost of opening a transfer facility to zero, which might provide a baseline to compare against.

Finally, we have not tuned the parameters of the heuristic but focused on the higher level concepts. Where possible, we have taken parameters from existing literature (Ropke and Pisinger, 2006; Masson et al., 2014) or provided reasoning for our parameter choice.
A small comparison to other groups has lead us to believe that we currently have very competitive solutions and thus that the need for this small enhancements is relatively small in comparison to the impact of the general, high-level design of the heuristic and the associated operators.

References

Capelle, Thomas, Cristián E Cortés, Michel Gendreau, Pablo A Rey, and Louis-Martin Rousseau (2019). A column generation approach for location-routing problems with pickup and delivery. *European Journal of Operational Research 272*(1), 121–131.

Dechter, Rina, Itay Meiri, and Judea Pearl (1991). Temporal constraint networks. *Artificial intelligence 49*(1-3), 61–95.

Li, Haibing and Andrew Lim (2003). A metaheuristic for the pickup and delivery problem with time windows. *International Journal on Artificial Intelligence Tools 12*(02), 173–186.

Masson, Renaud, Fabien Lehuédé, and Olivier Péton (2013). An adaptive large neighborhood search for the pickup and delivery problem with transfers. *Transportation Science 47*(3), 344–355.

Masson, Renaud, Fabien Lehuédé, and Olivier Péton (2014). The dial-a-ride problem with transfers. *Computers & Operations Research 41*, 12–23.

Mitrović-Minić, Snežana and Gilbert Laporte (2006). The pickup and delivery problem with time windows and transshipment. *INFOR: Information Systems and Operational Research 44*(3), 217–227.

Nanry, William P and J Wesley Barnes (2000). Solving the pickup and delivery problem with time windows using reactive tabu search. *Transportation Research Part B: Methodological 34*(2), 107–121.

Parragh, Sophie N, Karl F Doerner, and Richard F Hartl (2010). Variable neighborhood search for the dial-a-ride problem. *Computers & Operations Research 37*(6), 1129–1138.

Ropke, Stefan and David Pisinger (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science 40*(4), 455–472.

Shang, Jen S and Carolyn K Cuff (1996). Multicriteria pickup and delivery problem with transfer opportunity. *Computers & Industrial Engineering 30*(4), 631–645.

Stein, David M (1978). Scheduling dial-a-ride transportation systems. *Transportation Science 12*(3), 232–249.