INFORMS Journal on Computing
Vol. 24, No. 3, Summer 2012, pp. 343–355
ISSN 1091-9856 (print) | ISSN 1526-5528 (online)

http://dx.doi.org/10.1287/ijoc.1110.0454

A Hybrid Tabu Search and Constraint Programming Algorithm for the Dynamic Dial-a-Ride Problem

Gerardo Berbeglia
HEC Montréal, Montréal, Québec H3T 2A7, Canada, gerardo.berbeglia@hec.ca

Jean-François Cordeau
Department of Operations Management and Logistics, HEC Montréal, Montréal, Québec H3T 2A7, Canada, jean-francois.cordeau@cirrelt.ca

Gilbert Laporte
Department of Management Sciences, HEC Montréal, Montréal, Québec H3T 2A7, Canada, gilbert.laporte@cirrelt.ca

This paper introduces a hybrid algorithm for the dynamic dial-a-ride problem in which service requests arrive in real time. The hybrid algorithm combines an exact constraint programming algorithm and a tabu search heuristic. An important component of the tabu search heuristic consists of three scheduling procedures that are executed sequentially. Experiments show that the constraint programming algorithm is sometimes able to accept or reject incoming requests, and that the hybrid method outperforms each of the two algorithms when they are executed alone.

Key words: dial-a-ride problem; dynamic; constraint programming; tabu search; scheduling
History: Accepted by David Woodruff, Area Editor for Heuristic Search and Learning; received November 2009; revised October 2010; accepted February 2011. Published online in *Articles in Advance* May 17, 2011.

1. Introduction

In the dial-a-ride problem (DARP), a fleet of vehicles must serve transportation requests between given origins and destinations. The main application of the DARP arises in door-to-door transportation services offered to elderly and handicapped people in many cities. Case studies have been described for the cities of Toronto (Desrosiers et al. 1986), Berlin (Borndörfer et al. 1997), Bologna (Toth and Vigo 1996), Copenhagen (Madsen et al. 1995), and Brussels (Rekiek et al. 2006). The minimization of user inconvenience often has to be balanced with operation costs because these objectives usually conflict. User inconvenience is taken into consideration, for instance, by assigning time windows to pickups or deliveries and by imposing a maximum ride time for each user.

An important dimension of the DARP relates to the availability of information. In the *static* DARP, all requests are assumed to be known a priori, before routes are constructed. A solution therefore consists of a static output specifying the routing and scheduling information. In the *dynamic* DARP, some or all requests for service are received in real time while routing operations take place. Instead of a static output, a solution to a dynamic DARP consists of a solution strategy specifying which routing and scheduling actions should be performed in the light of newly received service requests and of the current state of the system.

Over the past 30 years, most studies on the DARP have focused on the static version (see the survey by Cordeau and Laporte 2007). In this paper, we develop a hybrid algorithm for the dynamic DARP, which has been less studied but has recently attracted some interest. One of the first studies on the dynamic DARP was carried out by Psaraftis (1980), who considered the single vehicle case. The author developed an exact $O(n^2 3^n)$ dynamic programming algorithm for the static DARP. Whenever a new request arrives, the static instance is updated and reoptimized by fixing the partial route already performed. Madsen et al. (1995) presented an insertion-based algorithm for a real-life multivehicle dynamic DARP for the transportation of elderly and handicapped people in Copenhagen. An algorithm for demand-responsive passenger services such as taxis, including time window restrictions for the dynamic requests, capacity constraints, and booking cancellations, has been developed by Horn (2002). A parallel algorithm for the dynamic DARP, proposed by Attanasio et al. (2004), works as follows. When a new request arrives, each of the parallel threads inserts the request randomly in the current solution and runs a tabu search algorithm to obtain a feasible solution. Another algorithm for a dynamic DARP was

developed by Coslovich et al. (2006). In the problem considered by these authors, a driver may unexpectedly receive a trip demand by a person located at a stop and must decide quickly whether or not to accept it. An efficient insertion algorithm attempts to insert incoming requests in at least one of the solutions in the repository, and a request is accepted only if the insertion algorithm succeeds. A two-phase algorithm for solving a complex dynamic DARP arising in the transportation of patients in hospitals was proposed by Beaudry et al. (2010). In the first phase, a fast insertion scheme is used, and the second phase involves a tabu search that attempts to improve the current solution. Finally, Xiang et al. (2008) studied a sophisticated dynamic DARP in which travel and service times have a stochastic component. New requests are inserted into the established routes by means of a local search procedure based on simple intertrip moves. See Berbeglia et al. (2010a) for a recent survey of the dynamic DARP and of other dynamic pickup and delivery problems.

The dynamic DARP studied in this paper can be described as follows. Let $G = (V, A)$ be a complete and directed graph with vertex set $V = \{0\} \cup R$, where vertex 0 is the depot, and let R represent the customer vertices. The set R is partitioned into $R^+ = \{1, \ldots, n\}$ (pickup vertices) and $R^- = \{n+1, \ldots, 2n\}$ (delivery vertices). Let $H = \{1, \ldots, n\}$ be the set of requests, and let T be the end of the planning horizon. Request i has an associated pickup vertex $i^+ = i \in R^+$, a delivery vertex $i^- = n + i \in R^-$, and a time t_i at which it is received. It is worth noting that at any time t, only the requests received up to time t are known, and therefore only the subgraph of G associated to those requests is needed. With each vertex $i \in V$ are associated a time window $[e_i, l_i]$, a service duration D_i, and a load q_i (with $D_0 = 0$, $q_0 = 0$, and $q_{n+j} = -q_j$ for $j = 1, \ldots, n$).

Requests are divided into two classes: *outbound* and *inbound*. In an outbound request, the passenger typically asks for service to travel from home to a destination, and in an inbound request, the user asks for a return trip. In our DARP model, users only impose a time window of a prespecified width on the arrival time for an outbound request and on the departure time for an inbound request. Thus, if request i is outbound (i.e., from home to a destination), the time window associated to the pickup vertex i is $[0, T]$, whereas if it is inbound, the time window associated to the delivery vertex $n+i$ is $[0, T]$. The delivery vertex of an outbound request and the pickup vertex of an inbound request are called *critical*. The maximum allowed ride time of a user, defined as the difference between the arrival time at destination and the departure time at origin, is L. Each arc (i, j) has a nonnegative routing cost c_{ij} and a routing time T_{ij}, both satisfying the triangular inequality.

A *route* is a circuit over some vertices, starting and finishing at the depot. A request is said to be *served* when it is part of a route. The set of routes must satisfy the following constraints:

(i) the pickup and delivery vertices of any request are either both in the same route or none of them are;

(ii) all requests known at the beginning of the time horizon must be served;

(iii) the pickup vertex of a request must precede its delivery vertex;

(iv) the load of any vehicle may never exceed the vehicle maximum load capacity, denoted by Q;

(v) the ride time of each served request cannot exceed L; and

(vi) the pickup and delivery of each served request are performed in their respective time windows.

Our solution strategy for the dynamic DARP is as follows. An initial solution to serve the known requests is obtained by first assigning every request to a randomly selected vehicle and inserting the pickup and delivery vertices of the request at the end of the partially constructed routes. Then, using this solution as a seed, the tabu search procedure finds a feasible solution. As time evolves, service requests are received, and a quick decision on whether to accept or reject each of them has to be made. This decision is final, meaning that no rejected request can later be accepted, and all accepted requests must be served. The algorithm must

(i) decide whether or not to accept an incoming request; and

(ii) serve the accepted requests in such a way that at the end of the time horizon, all routes respect the properties just described.

The hybrid algorithm we have developed consists of a *tabu search* (TS) heuristic procedure combined with an exact *constraint programming* (CP) algorithm that is able to determine whether a given instance of the DARP is feasible or not. The role of the TS heuristic is to continually optimize the current solution and to try to insert incoming requests into the current solution. When an incoming request is received, the constraint programming algorithm is also executed, in parallel to the tabu procedure, in the hope of finding a feasible solution or to prove that no feasible solution compatible with the past actions exists. The incoming request is accepted only when either the TS algorithm or the CP algorithm identifies a feasible solution. The request is rejected when the CP algorithm proves the infeasibility or after a preset time limit, generally of one or two minutes.

As a rule, the TS algorithm can easily insert a new request in the current solution when it is not too tightly constrained. In contrast, the CP algorithm is rather effective in proving that no insertion is feasible in very tight scenarios. Our goal is to develop an

algorithm that combines the advantages of these two solution methodologies. There are two main benefits in applying CP in conjunction with TS. First, CP is sometimes able to find a feasible solution when the TS cannot or takes longer to do so. Second, in many instances when the TS has not found a solution, the CP algorithms can actually prove that no feasible insertion exists. From a quality-of-service point of view, proving that a given request cannot be inserted is a more convincing statement than simply stating that no solution has been found.

This type of methodology applies to several situations where it is required to construct a feasible solution for a tightly constrained combinatorial optimization problem, e.g., the traveling salesman problem with time windows (Focacci et al. 2002). It can be used for the construction of an initial solution and within a local search. One can also use a hybrid algorithm within a local search phase based on a destroy and reconstruct mechanism. Online problems, other than the dynamic DARP, arise in a variety of contexts. Two examples are machine scheduling with jobs arriving in real time (Hoogeveen and Vestjens 1996) and dynamic assignment problems (Spivey and Powell 2004) with tight constraints.

The remainder of this paper is organized as follows. In §2, we present the main components of the TS heuristic. In §3, we give a brief description of the constraint programming paradigm and present a model of the DARP as a constraint satisfaction problem. The three scheduling algorithms used by the TS heuristic are then described in §4. The main scheme of the proposed hybrid algorithm is presented in §5, and computational results are given in §6. We close this paper with some conclusions in §7.

2. Tabu Search

Tabu search is a metaheuristic that combines local search with a memory scheme in order to avoid visiting the same solutions repetitively (Glover and Laguna 1997). It has been proved to be very successful in vehicle routing (see, e.g., Cordeau et al. 2001, Gendreau et al. 1994). In this section we present the basic TS concepts applied to our algorithm for the dynamic DARP. The algorithm we have developed is based on the TS procedure for the static DARP developed by Cordeau and Laporte (2003). We will provide a summary of the main features and dynamic aspects of the procedure. We refer the reader to the original paper for a more extensive description.

One of the important characteristics of the TS algorithm is the allowance of infeasible solutions during the search. A solution is represented by a set of m routes such that

(i) each route starts and ends at the depot,

(ii) each accepted request is assigned to exactly one route, and

(iii) for each accepted request, the pickup vertex precedes the delivery vertex.

Therefore, an intermediate solution may violate the ride time constraints and the time window constraints associated to the requests, as well as the capacity constraints associated to the vehicles.

2.1. Relaxation Mechanism and Objective Function

Let $r = (i_0, \ldots, i_k)$ be a route of a given solution s, and let $c(r)$, $q(r)$, $w(r)$, and $t(r)$ denote the routing cost, load violation, time window violation, and ride time violation of route r, respectively. Stated formally, $c(r) = \sum_{u=0}^{k-1} c_{i_u, i_{u+1}}$ and $q(r) = \sum_{u=1}^{k} (q_{i_u} - Q)^+$, where $x^+ = \max\{0, x\}$. The time window violation is defined as $w(r) = \sum_{u=0}^{k} (BT_{i_u} - l_{i_u})^+$, where BT_i specifies the start of service at vertex i. Finally, the ride time violation is given by $t(r) = \sum_{u=1}^{k} (L_{i_u} - L)^+$, where $L_i = 0$ if i is a pickup vertex and is equal to the ride time of the request associated to vertex i in case i is a delivery vertex. The total routing cost of a given solution s with routes $\{r_1, \ldots, r_m\}$ is $c(s) = \sum_{u=1}^{m} c(r_u)$. Similarly, the total load, total time window, and total ride time violations of solution s are equal to the sum of their respective violations for each route.

The total cost of a solution s is equal to $f(s) = c(s) + \alpha q(s) + \gamma w(s) + \tau t(s)$. Initially, the parameters α, γ, and τ are set equal to 1. They are dynamically adjusted after each iteration as follows. If the current solution respects the load constraint, the value α is divided by $1 + \delta$; otherwise, it is multiplied by $1 + \delta$, where δ is a uniformly distributed random number between 0 and 0.5 and is updated every 10 iterations. The same procedure applies to γ and τ, regarding the time window and ride time violations, respectively. In computational experiments, we have observed that controlling the value of the parameters in this way allows the tabu search algorithm to alternate between periods of diversification and intensification during the search.

2.2. Neighbourhood Definition and Evaluation

A request i is said to be *fixed* in a solution s and at current time t if the request cannot be moved to another route. This happens when either the pickup vertex has already been served at time t or the vehicle has already left the vertex preceding the pickup vertex of i, because diversion of vehicles is not allowed.

A solution s is characterized by the set $U(s) = \{(i, k)$: request i is assigned to vehicle $k\}$. The neighbourhood $N(s, t)$ of a solution s at time t consists of all solutions that can be reached by removing an attribute (i, k) from $U(s)$, whose request is not fixed at time t, and replacing it with a new attribute (i, k'),

where $k' \neq k$. When a request is removed from a route, the order of the remaining vertices in the route is unchanged. When the insertion of a request into a route r takes place, the order of the other vertices in r remains unchanged, and the pickup and delivery are located to minimize the total cost function, described in §2.4.

After the removal or insertion of a request, the cost of the route must be updated. Computing the new routing cost as well as the capacity violations can be achieved easily in linear time. More complex computations are needed to update the time window and ride time violations. To compute these two violations, a route scheduling algorithm is required. We have developed three scheduling algorithms that are described in §4.

2.3. Route Optimization

Intraroute optimization is performed every κ iterations by sequentially removing one vertex at a time and reinserting it in a position that minimizes $f(s)$. As an additional search intensification, this procedure is also performed whenever a new incumbent is identified. In our implementation κ was set to 10.

2.4. Tabu Control, Aspiration, and Diversification

To avoid repeating solutions, a request i removed from a route r cannot be inserted back into this route for the next θ iterations. The value of θ is a random number uniformly distributed between 0 and $7.5 \log_{10} n$, and it is updated every 10 iterations. As an aspiration mechanism, the tabu prohibition is disabled when the reinsertion would produce a solution with smaller cost than the best-known solution having request i in route r.

The tabu search algorithm evaluates a solution s using the objective function $f(s) + p(s)$, where $p(s)$ is used to diversify the search and penalizes a neighbour solution s' of s, only when $f(s') > f(s)$. This penalty is proportional to the frequency of addition of its distinguishing attributes and of a scaling factor. More precisely, suppose that (i, k) is the attribute that must be added to the current solution s in order to obtain the new solution $\bar{s}$, and let ρ_{ik} denote the number of times attribute (i, k) has been added to the solution during the search. The penalty term used to evaluate solution $\bar{s}$ is then

$$p(\bar{s}) = \mu c(\bar{s}) \sqrt{nm} \rho_{ik},$$

where m is the number of vehicles, and μ is a random number uniformly distributed between 0 and 0.015, and it is also updated every 10 iterations.

3. Constraint Programming

We now provide a brief introduction to constraint programming, and we present a model of the DARP as a constraint satisfaction problem. Constraint programming is a programming paradigm based on reasoning and search techniques, which is applied to the solution of combinatorial problems. It originally emerged from the artificial intelligence community in the 1970s, when the concept of a constraint satisfaction problem was formulated. In the 1980s, logic programming researchers developed several constraint solving algorithms that have led to the development of *constraint logic programming*. This paradigm extends the *logic programming* concept through the use of constraints. Constraint programming then appeared in the 1990s through a transformation of constraint logic programming, in which a constraint-orientated view and more sophisticated propagation techniques were developed. For an introduction to these concepts, see Van Hentenryck (1989).

In CP, a problem is modeled as a *constraint satisfaction problem* (CSP). Stated informally, a CSP consists of a set of variables and a set of restrictions, called *constraints*, over the variables. A constraint on a sequence of variables is a relation on the variable domains. It states which combinations of values from the variable domains are permitted and which of them are not. Once we have modeled a problem as a CSP, we proceed to solve it. Constraint programming solves a model using inference algorithms to reduce the search space, as well as search methods. The inference algorithms, called *constraint propagation algorithms* or *filtering algorithms*, try to simplify the problem by removing values from variable domains while preserving the same set of solutions. Search methods generally consist of backtracking or branch-and-bound combined with constraint propagation. Constraint programming has been successfully applied to scheduling, planning, molecular biology, finance, and numerical analysis. These and other applications of CP are surveyed in van Hoeve and Katriel (2006).

We now give a formulation of the static DARP as a constraint satisfaction problem based on successor variables presented by Berbeglia et al. (2010b). In §5 we show how to use this model for the dynamic version of the problem. We first extend graph G as follows. Vertex 0, corresponding to the depot, is replaced by the depot set $V = V^+ \cup V^-$ with $|V^+| = |V^-| = m$. The new graph G has $|V| + |R| = 2m + 2n$ vertices. Vehicle $k \in K = \{1, \ldots, m\}$ is represented by vertices $\text{start}(k) \in V^+$ (starting depot) and $\text{end}(k) \in V^-$ (ending depot). Under this transformation, the route of vehicle k is represented by the circuit $(\text{start}(k)):S_k:(\text{end}(k))$, where S_k is a sequence, possibly empty, of client vertices.

We list the variables for the constraint programming formulation. For each vertex $i \in V \cup R$,

(i) $s[i] \in V \cup R$ identifies the direct successor of vertex i;

(ii) $\ell[i] \in [0, Q]$ states the vehicle load just after performing the pickup or delivery at vertex i;

(iii) $v[i] \in K$ indicates the vehicle serving vertex i; and

(iv) $t[i] \in [e_i, l_i]$ represents the time at which vertex i is served.

The constraints for the DARP are the following.

Basic constraints

(i) For each vehicle $j \in K$, $s[\text{end}(j)] = \text{start}(j)$.

(ii) For each vehicle $j \in K$, $v[\text{end}(j)] = v[\text{start}(j)] = j$.

(iii) *allDifferent*(s).

(iv) For each request $i \in H$, $v[i^+] = v[i^-]$.

(v) For each vertex i, $v[i] = v[s[i]]$.

Precedence and time window constraints

(vi) For each request $i \in H$, $t[i^+] \le t[i^-] - T_{i^+, i^-} - D_{i^+}$.

(vii) For each vertex $j \in V^+ \cup R$, $t[j] \le t[s[j]] - T_{j, s[j]} - D_j$.

Capacity constraints

(viii) For each vehicle j, $\ell[\text{start}(j)] = 0$.

(ix) For each client vertex $i \in R$, $\ell[s[i]] = \ell[i] + q_{s[i]}$ and $\ell[i] \le Q$.

Ride time constraints

(x) For each request $i \in H$, $t[i^-] - (t[i^+] + D_{i^+}) \le L$.

The global constraint *allDifferent(s)* ensures that $s[i] \ne s[j]$ whenever $i \ne j$. This CSP is solved with the constraint programming algorithm proposed by Berbeglia et al. (2010b), which contains filtering methods, symmetry-breaking strategies, and variable-fixing techniques for improving the efficiency.

4. Scheduling

An important aspect of an algorithm for the dynamic DARP consists of deciding at which time the vehicles arrive, start service, and depart from each vertex. As will be shown in §6, the scheduling strategy alone has a considerable impact on algorithm performance.

In this section we present three scheduling algorithms: *basic scheduling*, *lazy scheduling*, and *eager scheduling*. Given a fixed route r and a current time t, these algorithms output the arrival time, start of service time, and departure time for each vertex in r without modifying the actions taken before time t.

Consider a vehicle route $r = (0, \ldots, o)$ with 0 and o being the depot vertex. We define the following scheduling variables for each vertex $j = 0, \ldots, o$:

AT_j: the arrival time at vertex j.

BT_j: the start of service at vertex j (defined as $t[j]$ in the previous section).

DT_j: the departure time at vertex j.

WT_j: the waiting time at vertex j before service ($WT_j = BT_j - AT_j$).

For clarity of exposition, we assume that the service duration D_j for each vertex is equal to 0. The algorithms presented in this section can easily be adapted to the case where the service duration has a positive value. At vertex 0, which represents the depot at the start of the route, $BT_0 = DT_0$, $AT_0 = 0$, and $e_0 = e_o = 0$. For the vertex o, which is the depot at the end of the route, $AT_o = BT_o = DT_o$ represents the arrival time.

A *schedule* for route r consists of an assignment of values to the variables AT_{j+1}, BT_j, and DT_j for $0 \le j \le o-1$. It is assumed that $e_j \le BT_j$ for $0 \le j \le o$. Observe that a schedule must also satisfy

$$AT_{j+1} = DT_j + T_{j,j+1} \quad \text{for all } 0 \le j \le o-1, \quad \text{and} \tag{1}$$

$$DT_j \ge BT_j \quad \text{for all } 0 \le j \le o. \tag{2}$$

Thus, to define a schedule it is sufficient to fix either $BT_0, \ldots, BT_{o-1}$ and $AT_1, \ldots, AT_o$ or $BT_0, \ldots, BT_{o-1}$ and $DT_0, \ldots, DT_{o-1}$.

A schedule is *feasible* if

(i) $e_j \le BT_j \le l_j$ for $0 \le j \le o$; and

(ii) given any request i such that the pickup vertex and the delivery vertex are in r, i.e., $i^+ \in r$ and $i^- \in r$, then $BT_{i^-} - BT_{i^+} \le L$.

Assume we are given a time value $t < BT_{o-1}$, a route r, and a schedule for the route. We are interested in modifying the schedule for route r without altering the arrival, the start of service, and the departure times of any vertex that was served before time t.

Three cases can be distinguished.

Case 1. If the vehicle has not yet started the route (i.e., $t < BT_0$), then the departure time at the depot can be modified but cannot occur before t.

Case 2. If the vehicle is moving toward a vertex (i.e., $DT_j \le t < AT_{j+1}$ for some $0 \le j \le o-1$), then the arrival time at vertex $j+1$ cannot be modified.

Case 3. If the vehicle is waiting to serve a customer (i.e., $AT_j \le t < BT_j$ for some $1 \le j \le o-1$), then the start of service at the vertex can be modified with the restriction that the new time x for the start of service must satisfy $t \le x$.

Let $k+1$ (with $0 \le k+1 \le o-1$) be the first vertex at which it is possible to modify the start of service (or the departure time when $k+1$ represents the depot), i.e., BT_{k+1}. Stated formally, $k = \max\{\min\{j \in \{1, \ldots, o\}: BT_{j+1} > t\} \cup \{-1\}$. Therefore, BT_j, DT_j, and AT_{j+1} cannot be modified for all $0 \le j \le k$.

4.1. Basic Scheduling

The basic scheduling procedure is described in Algorithm 1.

Algorithm 1 (Basic scheduling algorithm)

Input: A route $r = (0, \ldots, o)$, a time t, a number $k \in \{-1, \ldots, o-2\}$, and a schedule for route r.

$BT_{k+1} = \max\{e_{k+1}, AT_{k+1}, t\}$

$DT_{k+1} = BT_{k+1}$
for $j = k+2$ to $o-1$ **do**
  $AT_j = BT_{j-1} + T_{j-1,j}$
  $BT_j = \max\{e_j, AT_j\}$
  $DT_j = BT_j$
end for
$AT_o = BT_{o-1} + T_{o-1,o}$

The resulting schedule has the following properties.

(i) It minimizes the time window violation for any vertex $j = k+1, \ldots, o$ defined as $(BT_j - l_j)^+$, and thus, it also minimizes the total violation $\sum_{j=k+1}^{o}(BT_j - l_j)^+$.

(ii) It minimizes the start of service time B_j of any vertex j with $k+1 \le j \le o$.

The algorithm serves each vertex as early as possible but always ensures that service at vertex j cannot begin before e_j. As stated in Cordeau and Laporte (2003), the schedule produced by the basic scheduling algorithm may not be feasible, even though there actually exists a feasible schedule. This is because it may sometimes be worthwhile to delay the service of a vertex in order to reduce the ride time of the associated request. The algorithm presented in the next section, called lazy scheduling, overcomes this problem.

4.2. Lazy Scheduling Algorithm

We present here a procedure called the lazy scheduling algorithm, which is the dynamic version of an algorithm for the static DARP proposed by Cordeau and Laporte (2003). The algorithm transforms a schedule into another schedule called *lazy*, which minimizes the ride time violation of every request without increasing the time window violation of any vertex. The idea behind the lazy scheduling algorithm is to delay as much as possible the time BT_j at which service starts at vertex j, starting with vertex $k+1$ and finishing with vertex $o-1$. This is the reason why the algorithm is called *lazy*. The maximum possible delay at any vertex $j \in \{k+1, \ldots, o-1\}$ will be constrained so that there is no increase in the time window violation or in the ride time violation of any vertex of the route. Because the delay of the pickup vertex of every request precedes the delay of the delivery vertex, the procedure will sequentially minimize the ride time violation of each request.

When the input schedule is generated by the basic scheduling algorithm, the schedule produced by the lazy algorithm will be infeasible if and only if no feasible schedule actually exists. Thus, by applying the lazy scheduling algorithm after the basic scheduling algorithm, we can determine whether or not a given route possesses a feasible schedule.

The ride time of a request i is defined as $P_i = BT_{i^-} - BT_{i^+}$. We now derive the algorithm for producing the lazy schedule. We assume in this derivation that the variables AT_j, BT_j, WT_j, and DT_j contain the scheduling values of the input schedule, and we denote by BT'_j the new departure time at vertex j. We wish to determine the latest time at which service at vertex $k+1$ can start without increasing the time window violation of any vertex and without increasing the ride time violation of any request. Let $J_k^- = \{i^- \in \{k+1, \ldots, o\}$ be such that $i^+ \in \{1, \ldots, k\}\}$. To not increase the ride time violation of a request i with $i^- \in J_k^-$, the start of service at vertex $k+1$ cannot be performed later than

$$AT_{k+1} + \sum_{j=k+1}^{i^-} WT_j + (L - P_i)^+.$$

Thus,

$$BT'_{k+1} \le AT_{k+1} + \min_{i:\, i^- \in J_k^-} \left\{ \sum_{j=k+1}^{i^-} WT_j + (L - P_i)^+ \right\}.$$

For any vertex $j \in \{k+1, \ldots, o\}$, the start of the service at vertex $k+1$ cannot be later than

$$AT_{k+1} + \sum_{u=k+1}^{j} WT_u + (l_j - BT_j)^+.$$

Thus,

$$BT'_{k+1} \le AT_{k+1} + \min_{j \in \{k+1, \ldots, o\}} \left\{ \sum_{u=k+1}^{j} WT_u + (l_j - BT_j)^+ \right\}.$$

Therefore, the latest time at which it is possible to serve vertex $k+1$ without increasing the time window violation of any vertex and without increasing the ride time violation of any request i with $i^- \in J_k^-$ is

$$BT'_{k+1} = AT_{k+1} + \min\left\{ \min_{j \in \{k+1, \ldots, o\}} \left\{ \sum_{u=k+1}^{j} WT_u + (l_j - BT_j)^+ \right\}, \min_{i:\, i^- \in J_k^-} \left\{ \sum_{u=k+1}^{i^-} WT_u + (L - P_i)^+ \right\} \right\}.$$

The lazy scheduling algorithm is presented in Algorithm 2.

Algorithm 2 (Lazy scheduling algorithm)

1: Input: A route $r = (0, \ldots, o)$, a number $k \in \{-1, \ldots, o-2\}$, and a schedule for route r.
2: **for** $h = k+1$ to $o-1$ **do**
3: $BT_h = AT_h + \min\{\min_{j \in \{h, \ldots, o\}}\{\sum_{u=h}^{j} WT_u + (l_j - BT_j)^+\}, \min_{i:\, i^- \in J_{h-1}^-}\{\sum_{u=h}^{i^-} WT_u + (L - P_i)^+\}\}$
4: $DT_h = BT_h$
5: $AT_{h+1} = DT_h + T_{h,h+1}$
6: $WT_h = BT_h - AT_h$
7: **for** $f = h+1$ to $o-1$ **do**
8: $BT_f = \max\{e_f, AT_f\}$

9: $AT_{f+1} = B_f + T_{f,f+1}$
10: $WT_f = BT_f - AT_f$
11: **end for**
12: **for** each request i such that $\{i^+, i^-\} \subseteq \{0, \ldots, o\}$ **do**
13: $P_i = BT_{i^-} - BT_{i^+}$
14: **end for**
15: **end for**

Once B_{k+1} is computed, we set $DT_{k+1} = B_{k+1}$, and the arrival time at vertex $k+2$ (A_{k+2}) is obtained by Algorithm 1. This delay on the service at vertex $k+1$ propagates along all the following vertices as shown in lines 7 to 11. Once the effects of delaying vertex $k+1$ have been computed, the algorithm proceeds with delaying vertices $k+2$ up to $o-1$.

A potential problem of the lazy schedule is that it may become hard to insert a new request into the route. This is because the vehicle serves the vertices as late as possible, which means that when a new request arrives, there may be insufficient available slack time to insert it. In contrast, it may be easier to perform the insertion if the vertices were served earlier. In the next section we present the eager scheduling algorithm, which produces a schedule in which each vertex is served as early as possible without increasing the time window and ride time violations.

4.3. Eager Scheduling Algorithm

The eager scheduling algorithm transforms a given schedule into another one that minimizes the start of service time BT_i of every vertex i without increasing the time window violation of any vertex and without increasing the ride time violation of any request. Unlike the lazy scheduling algorithm, this procedure does not minimize ride time violations but only ensures that they will not be increased. Thus, to obtain a schedule that minimizes (1) the time window violations, (2) the ride time violations, and (3) the service starting time of each vertex, we can apply first the basic scheduling algorithm, then apply the lazy scheduling algorithm to its output, and finally use this schedule as an input to the eager scheduling algorithm.

Yuen et al. (2009) developed a scheduling algorithm called drive first (DF) for the dynamic DARP in which the vehicles serve vertices as soon as possible. However, they have modeled the problem in such a way that the maximum ride time restrictions can be expressed through the time window constraints. This is not possible in our definition of the DARP. As a result, their algorithm for minimizing the start of service time at each vertex is much simpler than the one we present here and is equivalent to the methods employed for the solution of pickup and delivery problems in which there are no ride time constraints (see, e.g., Mitrović-Minić and Laporte 2004).

The idea of the algorithm is as follows. Starting from the last vertex of the route, we compute the minimum time required to arrive at that vertex. Once this value is determined, we move to the previous vertex until we finish with vertex $k+1$. Let $\tilde{AT}_{j+1}$ and $\tilde{BT}_j$ be the arrival time at vertex j and the start time of the service at vertex j in the basic schedule for $k+1 \le j \le o-1$, respectively. The sequences $(\tilde{AT}_{k+2}, \ldots, \tilde{AT}_o)$ and $(\tilde{BT}_{k+1}, \ldots, \tilde{BT}_{o-1})$ can be computed using Algorithm 1. Consider again the route $r = (0, \ldots, o)$ and a schedule for r. The amount of time by which it is possible to antepone the arrival at h, with the only restriction of serving each vertex j with $0 \le j \le h-1$ not before e_j, is equal to $AT_h - \tilde{AT}_h$. Assume that $\{BT_h, \ldots, BT_{o-1}\}$ and $\{AT_{h+1}, \ldots, AT_o\}$ are fixed; i.e., the service time of vertices h up to $o-1$ cannot be changed (with $0 \le h \le o-1$). Assume also that the starting time of the vertices in $\{0, \ldots, h-1\}$ cannot be increased. This is coherent with our objective of minimizing the starting time BT_j of every vertex j. Therefore, at time t, to not increase the ride time of a request i with $i^- \in J^-_{h-1}$, the arrival time at vertex h cannot be earlier than

$$AT_h - \left((L - P_i)^+ + \sum_{j=\lambda}^{h-1} (BT_j - \max\{e_j, AT_j, t\}) \right), \quad (3)$$

where $\lambda = \max\{i^+ + 1, k+1\}$. The validity of this inequality can be explained as follows. The ride time of request i is measured by $P_i = BT_{i^-} - BT_{i^+}$. It is assumed that BT_{i^-} cannot be modified and that BT_{i^+} cannot be increased. Thus, without increasing the ride time P_i, the only feasible time margin for arrival at vertex h is equal to the sum of the waiting times that we can potentially reduce. These are the waiting times over the vertices $\{\lambda, \ldots, h-1\}$, whose sum is equal to $\sum_{j=\lambda}^{h-1} BT_j - \max\{e_j, AT_j, t\}$. This is an upper bound on the total time that it is possible to gain by serving vertices $\{\lambda, \ldots, h-1\}$ earlier. Because it is sometimes possible to increase the ride time, we add the term $(L - P_i)^+$ that states by how much the ride time can be increased without producing a violation. Therefore, the following inequality must hold:

$$AT'_h \ge AT_h - \min_{i: i^- \in J^-_{h-1}} \left\{ (L - P_i)^+ + \sum_{j=\lambda}^{h-1} (BT_j - \max\{e_j, AT_j, t\}) \right\}.$$

The eager scheduling procedure is described in Algorithm 3. Proceeding backward from vertex $h = o$ to vertex $k+2$, the algorithm computes the earliest arrival time at vertex h using (3) and then sets the departure time and service time at the previous vertex in lines 5 and 6, respectively. This advance in the departure at vertex $h-1$ propagates backward into an update of the arrival and start of service of the vertices between $h-1$ and $k+1$. Basically, the new arrival

time at a vertex j is equal to the minimum between the previous arrival time and the new start of service time. At the end of each step in the main cycle that iterates on h, defined between lines 2 and 16, the algorithm has computed the final value of the arrival time at vertex h and the start of service at vertex $h-1$.

Algorithm 3 (Eager scheduling algorithm)

1: Input: A route $r=(0,\dots,o)$, a number $k\in\{-1,\dots,o-2\}$, and a schedule for route r.
2: **for** $h=o$ to $k+2$ **do**
3: $\Delta_h=\min\{AT_h-\tilde{A}T_h,\ \min_{i:\,i^-\in J^-_{h-1}}\{(L-P_i)^+ + \sum_{j=\max\{i^++1,k+1\}}^{h-1}(B_j-\max\{e_j,AT_j,t\})\}\}$
4: $AT_h=AT_h-\Delta_h$
5: $BT_{h-1}=AT_h-T_{h-1,h}$
6: $DT_{h-1}=BT_{h-1}$
7: $j=h-1$
8: **while** $j\geq k+2$ **do**
9: $AT_j=\min\{BT_j,AT_j\}$
10: $BT_{j-1}=AT_j-T_{j-1,j}$
11: $j=j-1$
12: **end while**
13: **for** each request i such that $\{i^+,i^-\}\subseteq\{0,\dots,o\}$ **do**
14: $P_i=BT_{i^-}-BT_{i^+}$
15: **end for**
16: **end for**

4.3.1. Delaying the Departure. In the three scheduling algorithms just described, the vehicle departs from a vertex immediately after service takes place; i.e., $DT_j=BT_j$ for all $j=k+1,\dots,o-1$. It is possible, however, to modify this by applying Equations (4) and (5) to the schedule produced by any of the three scheduling algorithms:

$$DT_j=BT_{j+1}-T_{j,j+1}\quad\text{for all } k+1\leq j\leq o-1, \tag{4}$$

$$AT_j=BT_j\quad\text{for all } k+1\leq j\leq o-1. \tag{5}$$

This modification does not change the properties of the output schedules for any of the three algorithms. The advantage in delaying the departure time is that it creates a waiting period that allows the TS and CP algorithms to change the next vertex to visit and thus increase the space in which to find a feasible solution.

5. A Hydrid Algorithm

We now present the most important aspects of the hybrid algorithm combining the TS heuristic described in §§2 and 4 and the exact CP algorithm proposed by Berbeglia et al. (2010b). We recall that given an instance I of the static DARP, the constraint programming algorithm returns either a feasible solution for I or proves that none exists. Our purpose, however, is slightly different. We wish to determine whether it is possible or not, in a dynamic context, to accept and satisfy an incoming request by updating the current solution. We explain below how the CP algorithm was adapted for this purpose.

When a new request is received at time t, a new instance I of the DARP is created, containing all the static and accepted requests up to time t as well as the new request. Naturally, if the CP algorithm is executed with instance I as input and no additional constraints, it may find a feasible solution whose routing and scheduling actions up to time t do not correspond to the ones that were actually implemented. This difficulty is resolved through the introduction of additional constraints in the constraint programming model, which state that the solution must respect the partial routes followed up to time t.

Observe that in the CP model there are no variables to represent the arrival and departure times at each of the vertices. However, one must take the departure times of the current solution into account in order to properly fix the CP variables and thus avoid inconsistencies. The pseudocode of this procedure is given in Algorithm 4. It considers one route at a time and can be divided into two parts. In the **while** cycle (i.e., lines 6–17), it either sets a lower bound or fixes the service time for the relevant vertices. The binary variable *isFixed* is used to exit this cycle. In the **for** cycle (i.e., lines 18–20), it fixes the successor variables up to time t in the given solution.

Algorithm 4 (Procedure for fixing a partial solution to the CP algorithm)

1: Input: DARP instance with new request I, current solution s (without the new request), and actual time t.
2: Load the CP model for instance I
3: **for** each of route $r=(i_0,\dots,i_k)$ of solution s **do**
4: $isFixed=1$
5: $j=0$
6: **while** $j\leq k$ AND $isFixed=1$ **do**
7: **if** $BT_{i_j}\leq t$ **then**
8: $t[i_j]=BT_{i_j}$
9: **if** $DT_{i_j}<t$ **then**
10: $t[s[i_j]]\geq DT_{i_j}+T_{i_j,s[i_j]}$
11: **else**
12: $t[s[i_j]]\geq t+T_{i_j,s[i_j]}$
13: $isFixed=0$
14: **end if**
15: **else**
$t[i_j]\geq t$
16: **end if**
17: **end while**
18: **for** each u from 0 to $j-1$ **do**
19: $s[i_u]=i_{u+1}$

20: **end for**
21: **end for**

The main template of the hybrid algorithm is provided in Algorithm 5. First, a feasible solution is obtained by the TS algorithm, considering only the static requests. When no new incoming requests arrive, the solution is optimized using the tabu search algorithm. This requires special attention because any new optimized solution should not be different with respect to the previous solution up to the time at which the new solution is obtained. To this end, the tabu search is performed for a fixed duration of φ minutes, and the input solution is frozen up to φ minutes in advance of the current time, where φ is a parameter fixed to 2 in our implementation. When this period of time has elapsed, the current solution is updated. This procedure is repeated until a new request arrives, in which case the optimization is interrupted. After a new request is received, a new instance I' is created that contains all the data of the static and previously accepted requests, as well as the new request. The tabu search and the CP algorithms are then executed in parallel with the input instance I', freezing the partial routes up to the current time, plus φ minutes. Both procedures are terminated when one of them has found a solution, when the CP algorithm has proved that the instance I' is infeasible subject to the fixed partial routes, or when the time limit of φ minutes of computing time has elapsed. Naturally, the incoming request is accepted only when a feasible solution has been found by any of the two algorithms, and it is rejected otherwise.

Algorithm 5 (Main scheme of the hybrid algorithm)

1: Obtain a solution s considering the instance I that only has the static requests using the tabu search algorithm.
2: **while** Time horizon has not been reached **do**
3: **while** No new requests **do**
4: Reoptimize actual solution s of I using the tabu search algorithm
5: **end while**
6: Create a new DARP instance I' by adding the new request
7: Execute in parallel the tabu search procedure and the constraint programming algorithm with I and time limit φ and freeze all partial routes up to time $t + \varphi$
8: **if** Either the tabu or the constraint programming procedures have found a solution s' **then**
9: ACCEPT request
10: $I = I'$, $s = s'$
11: **else**
12: **if** Infeasibility was proved by CP or time limit φ has passed **then**
13: REJECT request
14: **end if**
15: **end if**
16: **end while**

6. Computational Results

To assess the performance of the hybrid algorithm, we have conducted a series of tests on two sets of synthetic instances. We have also tested the algorithm on a set of instances based on a real-life data set provided by a Danish transporter.

6.1. Instance Generation

The first set of dynamic instances was based on the set of static instances *a* and *b* used in Ropke et al. (2007). In the instance subset *a*, vertices are located in a 20×20 square, taking floating point values, and with a uniform random distribution. The distances are Euclidean and are measured in minutes; the time horizon is 12 hours, the time windows of critical vertices have a 15-minute length, and $Q = 3$. The instance subset *b* is similar, except that $Q = 6$. We have only used the instances with at least 40 requests. The instance labels are of the form "a*m*-*n*" or "b*m*-*n*." The letters a and b state whether the instance is from the subset *a* or *b*, *m* corresponds to the number of vehicles, and *n* is the number of requests. More details of these instances can be found in Cordeau (2006).

These static instances were converted into dynamic ones by using a pair of parameters (α, β). The value $\alpha \in [0, 1]$ gives the ratio of the requests that are known at the beginning of the time horizon. Thus, if $\alpha = 1$ the instance is completely static, whereas setting $\alpha = 0$ yields an instance with no requests known a priori. Given a request i, the value $\mathcal{U}(i)$ is an upper bound on the time at which the request must be known in order to be able to serve it. It is defined as $\mathcal{U}(i) = \min\{l_{i^+}, l_{i^-} - T_{i^+i^-} - D_{i^+}\}$. The parameter β states how much time before $\mathcal{U}(i)$ request i is known. If $\mathcal{U}(i) < \beta$, then request i is known at time 0.

The static instances of the subsets *a* and *b* were transformed into dynamic instances with the parameters $\alpha = 0.25$ and $\beta = 60$; i.e., 25% of the requests are static, and each dynamic request i becomes known 60 minutes before $\mathcal{U}(i)$. The hybrid algorithm was tested using the lazy scheduling algorithm and the eager scheduling algorithm presented in §4. Table 1 gives the number of accepted requests by the tabu search and by the CP algorithm, the number of rejected requests because of a timeout of two minutes, and the number of infeasible requests identified by the CP algorithm. These results show that the number of dynamic requests that were accepted using the eager scheduling algorithm compared with those accepted with the lazy algorithm was increased by 270%. Our results also show that approximately 77%

Table 1 Comparison of the Number of Accepted Requests Using the Eager and Lazy Scheduling Algorithms on the First Set of Instances

	Lazy scheduling				Eager scheduling			
	Accepted requests		Rejected requests		Accepted requests		Rejected requests	
Instance	By tabu	By CP	Timed out	Proved	By tabu	By CP	Timed out	Proved
a4-40	23	0	0	5	5	17	1	5
a4-48	12	0	1	20	29	4	0	0
a5-40	15	0	0	12	25	2	0	0
a5-50	18	0	0	18	34	1	0	1
a5-60	29	0	0	14	36	1	1	5
a6-48	13	0	0	21	34	0	0	0
a6-60	10	0	8	22	37	2	0	1
a6-72	31	1	2	19	51	2	0	0
a7-56	19	0	3	14	34	2	0	0
a7-70	18	0	9	21	30	18	0	0
a7-84	24	1	16	19	50	3	0	7
a8-64	25	0	1	17	39	4	0	0
a8-80	4	0	11	38	48	0	0	5
a8-96	33	0	15	21	58	2	2	7
b4-40	14	0	0	14	28	0	0	0
b4-48	20	1	0	13	32	2	0	0
b5-40	9	0	1	16	24	2	0	0
b5-50	22	0	0	12	32	0	0	2
b5-60	29	0	9	6	36	1	0	7
b6-48	23	0	1	8	32	0	0	0
b6-60	28	0	8	8	41	2	0	1
b6-72	31	0	6	11	46	1	0	1
b7-56	12	0	10	16	31	0	1	6
b7-70	12	0	9	29	2	27	5	16
b7-84	6	0	8	42	53	3	0	0
b8-64	19	0	6	21	42	1	3	0
b8-80	19	1	6	29	52	0	0	3
b8-96	9	2	9	50	56	6	7	1

of all the rejected requests were proved to be infeasible by the CP algorithm.

The second set of instances (labeled pr01, ..., pr20) is based on the 20 static instances of Cordeau and Laporte (2003), which contain between 24 and 144 requests and between 3 and 13 vehicles. In these instances each request has a load of one unit and a maximum ride time of 90 minutes, and the vehicles have a capacity of six. The critical vertices have time windows of length varying between 15 and 90 minutes. This set of static instances was transformed into a set of dynamic instances with the parameters $\alpha = 0.25$ and β being a random number uniformly distributed between 60 and 240. This means that 25% of the requests are static, and each dynamic request i becomes known between one and four hours before its deadline, $\mathcal{U}(i)$.

The third set of instances is based on real-life data from a door-to-door transport service company located in Denmark. The instance provided by the transporter contains 200 requests, and the traveling times as well as the distances between each pair of vertices are not Euclidean but were computed by a geographical information system. Vehicles have a capacity of $Q = 8$, and most requests have unit loads, except for some requests whose load is 8, meaning that they must be transported alone. The maximum ride time for those special requests is 120 minutes, and for all the others it is 60 minutes. More details about the instance can be found in Cordeau and Laporte (2003). Based on this instance, 12 instances have been constructed by changing the size of the time windows and the total number of requests. The instance labels are of the form "st-p," where $t \in \{15, 30, 45\}$ corresponds to the length of the time windows and $p \in \{25, 50, 75, 100\}$ states the percentage of the 200 requests of the original instances that were randomly selected. We then have transformed each of these 12 static instances into a dynamic one using the same parameters as in the second set of instances, i.e., $\alpha = 0.25$ and β being a random number uniformly distributed between 60 and 240.

6.2. Results

In Table 2 we compare the performance of the dynamic DARP on these instances using the lazy

Table 2 Comparison of the Number of Accepted Requests Using the Eager and Lazy Scheduling Algorithms on the Second Set of Instances

	Lazy scheduling				Eager scheduling			
	Accepted requests		Rejected requests		Accepted requests		Rejected requests	
Instance	By tabu	By CP	Timed out	Proved	By tabu	By CP	Timed out	Proved
pr01	14	4	0	0	18	0	0	0
pr02	28	4	0	0	32	0	0	0
pr03	31	14	7	0	52	0	0	0
pr04	33	11	15	0	58	1	0	0
pr05	52	10	24	0	83	1	0	2
pr06	38	12	41	2	93	0	0	0
pr07	17	5	1	0	23	0	0	0
pr08	23	11	13	0	47	0	0	0
pr09	34	8	28	0	68	2	0	0
pr10	42	13	42	0	88	2	0	7
pr11	17	1	0	0	18	0	0	0
pr12	31	3	0	0	34	0	0	0
pr13	42	4	7	0	53	0	0	0
pr14	39	10	18	0	67	0	0	0
pr15	39	0	13	33	83	0	2	0
pr16	56	9	36	0	101	0	0	0
pr17	20	3	3	0	26	0	0	0
pr18	38	9	3	0	50	0	0	0
pr19	47	9	20	0	76	0	0	0
pr20	47	12	43	0	101	0	0	1

and the eager scheduling algorithms. We can see that the eager algorithm still performs better than the lazy algorithm, but the difference between the two, although significant, is smaller than on the first set of instances. On average, the number of accepted requests using the eager algorithm has increased by 34% compared with the number of accepted requests using the lazy algorithm. On these instances the CP algorithm had more difficulty proving the infeasibility of the rejected requests. On average, approximately 10% of the rejected requests were proven to be infeasible by the CP algorithm in the available running time of two minutes. An explanation for this much lower rate, compared with the 70% reached in the first set of instances, is that in the first set, the critical time windows are smaller, and therefore the solution space is reduced considerably.

A comparison between the performance of the dynamic DARP using the lazy and eager scheduling algorithms on the third instances is shown in Table 3. On these instances, the number of accepted requests using the eager algorithm has increased, on average, by approximately 20% compared with the number of accepted requests with the lazy algorithm.

Table 3 Comparison of the Number of Accepted Requests Using the Eager and Lazy Scheduling Algorithms on the Third Set of Instances

	Lazy scheduling				Eager scheduling			
	Accepted requests		Rejected requests		Accepted requests		Rejected requests	
Instance	By tabu	By CP	Timed out	Proved	By tabu	By CP	Timed out	Proved
s15-25	28	7	0	3	34	3	1	0
s15-50	53	11	7	5	63	8	3	2
s15-75	75	16	8	14	102	9	1	1
s15-100	102	13	7	29	127	6	8	10
s30-25	31	3	1	3	31	7	0	0
s30-50	60	2	8	6	62	14	0	0
s30-75	70	4	23	16	92	21	0	0
s30-100	118	3	25	5	129	16	4	2
s45-25	33	0	5	0	29	9	0	0
s45-50	62	2	10	2	70	5	1	0
s45-75	94	3	14	2	99	12	2	0
s45-100	90	3	57	1	119	12	19	1

Table 4 Comparison of the Number of Accepted Requests Using the Eager and Lazy Scheduling Algorithms with the Slack Time Objective

	Lazy scheduling				Eager scheduling			
	Accepted requests		Rejected requests		Accepted requests		Rejected requests	
Instance	By tabu	By CP	Timed out	Proved	By tabu	By CP	Timed out	Proved
pr01	16	2	0	0	18	0	0	0
pr02	28	4	0	0	32	0	0	0
pr03	33	16	3	0	52	0	0	0
pr04	29	16	0	14	59	0	0	0
pr05	47	13	26	0	80	1	5	0
pr06	34	20	39	0	93	0	0	0
pr07	18	4	1	0	23	0	0	0
pr08	27	9	11	0	47	0	0	0
pr09	33	8	15	14	66	3	1	0
pr10	42	18	45	0	91	1	5	0
pr11	17	1	0	0	18	0	0	0
pr12	25	8	1	0	34	0	0	0
pr13	33	10	10	0	52	1	0	0
pr14	40	8	19	0	67	0	0	0
pr15	43	19	23	0	82	1	0	2
pr16	50	21	30	0	101	0	0	0
pr17	22	2	2	0	26	0	0	0
pr18	35	9	6	0	49	1	0	0
pr19	37	14	25	0	75	1	0	0
pr20	57	2	43	0	102	0	0	0

Results also show that the percentage of the rejected requests that were proved to be infeasible is, on average, approximately 30%. It is noteworthy that the average rate of proved rejections is greater on the instances with shorter time windows. If we restrict our attention to the subset of instances with 15-minute time windows, the average rate of proved rejections rises to approximately 64%.

6.3. Modification of the Objective Function

We have also performed some experiments with a modified version of the tabu search algorithm in which the objective function is changed. We have added a term we call $slack(s)$ to the objective function $f(s)$. This new term rewards solutions whose route schedules can easily be modified and penalizes solutions whose routes have a rigid schedule. The idea is that an incoming request is unlikely to be inserted in a route whose schedule is very rigid, and therefore it is preferable to have solutions whose routes are more "schedule flexible." Let $r = (i_1, \ldots, i_k)$ be a route, and let BT_j^e and BT_j^l denote the start of service at vertex i_j using the eager and lazy schedules, respectively. We define the slack time of a route r as $slack(r) \max\{BT_j^l - BT_j^e\colon j = 1, \ldots, k\}$. The *slack* of a solution is equal to the sum of the slacks of each route. Although not perfect, this measure is global in that it takes all requests of the route into account. For instance, if the slack of a route is 30 minutes, this means that, at least for a given period of time, there are at least 30 extra minutes available to serve a new request without increasing the time window and ride time violation on any of the requests on the route.

Table 4 shows the results for the second set of instances when the objective function was modified to include the slack of the routes. When the lazy schedule is used, there is a slight increase of approximately 4%, on average, in the number of accepted requests compared with the algorithm without the slack measure in the objective function. However, this improvement is not observed when the eager algorithm is applied. In this case, the number of accepted requests is almost the same for both versions of the objective function.

7. Conclusions

We have developed a new hybrid algorithm for the dynamic DARP, combining a tabu search procedure and an exact constraint programming algorithm. Experiments performed on dynamic instances created from static instances have shown that the CP algorithm is sometimes able to accept or reject incoming requests. On the other hand, the tabu search tends to accept requests faster. This shows that the hybrid method outperforms either of the two algorithms when they are executed alone. The capability of the CP procedure to prove infeasibility varies considerably, depending on the type of instance and particularly on the width of the time windows.

Given a fixed route, we have developed scheduling algorithms to determine the times at which the arrival and the start of service at each vertex should take place. The basic and lazy scheduling algorithms are

the natural dynamic extensions of the procedures presented by Cordeau and Laporte (2003) for the static problem. We have then developed a new scheduling algorithm called *eager*, which serves each vertex as early as possible without increasing the time window or the ride time violation of any request. Results have shown that the eager algorithm leads to the acceptance of considerably more requests than is possible with the lazy algorithm.

Acknowledgments

This work was supported by the Canadian Natural Sciences and Engineering Research Council under Grants 227837-04 and 39682-05. This support is gratefully acknowledged. Thanks are due to the reviewers for their valuable comments.

References

Attanasio, A., J.-F. Cordeau, G. Ghiani, G. Laporte. 2004. Parallel tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem. *Parallel Comput.* **30**(3) 377–387.

Beaudry, A., G. Laporte, T. Melo, S. Nickel. 2010. Dynamic transportation of patients in hospitals. *OR Spectrum* **32**(1) 77–107.

Berbeglia, G., J.-F. Cordeau, G. Laporte. 2010a. Dynamic pickup and delivery problems. *Eur. J. Oper. Res.* **202**(1) 8–15.

Berbeglia, G., G. Pesant, L.-M. Rousseau. 2010b. Checking the feasibility of dial-a-ride instances using constraint programming. *Transportation Sci.*, ePub ahead of print September 24, http://transci.journal.informs.org/cgi/content/abstract/trsc.1100.0336v1.

Borndörfer, R., F. Klostermeier, M. Grötschel, C. Küttner. 1997. Telebus Berlin: Vehicle scheduling in a dial-a-ride system. Technical Report SC 97–23, Konrad-Zuse-Zentrum fur Informationstechnik, Berlin.

Cordeau, J.-F. 2006. A branch-and-cut algorithm for the dial-a-ride problem. *Oper. Res.* **54**(3) 573–586.

Cordeau, J.-F., G. Laporte. 2003. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Res. Part B* **37**(6) 579–594.

Cordeau, J.-F., G. Laporte. 2007. The dial-a-ride problem: Models and algorithms. *Ann. Oper. Res.* **153**(1) 29–46.

Cordeau, J.-F., G. Laporte, A. Mercier. 2001. A unified tabu search heuristic for vehicle routing problems with time windows. *J. Oper. Res. Soc.* **52**(8) 928–936.

Coslovich, L., R. Pesenti, W. Ukovich. 2006., A two-phase insertion technique of unexpected customers for a dynamic dial-a-ride problem. *Eur. J. Oper. Res.* **175**(3) 1605–1615.

Desrosiers, J., Y. Dumas, F. Soumis. 1986. A dynamic programming solution of the large-scale single-vehicle dial-a-ride problem with time windows. *Amer. J. Math. Management Sci.* **6**(3–4) 301–325.

Focacci, F., A. Lodi, M. Milano. 2002. A hybrid exact algorithm for the TSPTW. *INFORMS J. Comput.* **14**(4) 403–417.

Gendreau, M., A. Hertz, G. Laporte. 1994. A tabu search heuristic for the vehicle routing problem. *Management Sci.* **40**(10) 1276–1290.

Glover, F., M. Laguna. 1997. *Tabu Search*. Kluwer Academic Publishers, Boston.

Hoogeveen, J. A., A. P. A. Vestjens. 1996. Optimal on-line algorithms for single-machine scheduling. W. H. Cunningham, S. T. McCormick, M. Queyranne, eds. *Conf. Integer Programming Combin. Optim.* Lecture Notes in Computer Science, Vol. 1084. Springer-Verlag, Berlin, 404–414.

Horn, M. E. T. 2002. Fleet scheduling and dispatching for demand-responsive passenger services. *Transportation Res. Part C* **10**(1) 35–63.

Madsen, O. B. G., H. F. Ravn, J. M. Rygaard. 1995. A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives. *Ann. Oper. Res.* **60**(1) 193–208.

Mitrović-Minić, S., G. Laporte. 2004. Waiting strategies for the dynamic pickup and delivery problem with time windows. *Transportation Res. Part B* **38**(7) 635–655.

Psaraftis, H. N. 1980. A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Sci.* **14**(2) 130–154.

Rekiek, B., A. Delchambre, H. A. Saleh. 2006. Handicapped person transportation: An application of the grouping genetic algorithm. *Engrg. Appl. Artificial Intelligence* **19**(5) 511–520.

Ropke, S., J.-F. Cordeau, G. Laporte. 2007. Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks* **49**(4) 258–272.

Spivey, M. Z., W. B. Powell. 2004. The dynamic assignment problem. *Transportation Sci.* **38**(4) 399–419.

Toth, P., D. Vigo. 1996. Fast local search algorithms for the handicapped persons transportation problem. I. H. Osman, J. P. Kelly, eds. *Meta-Heuristics Theory and Applications*. Kluwer Academic Publishers, Boston, 677–690.

Van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA.

van Hoeve, W.-J., I. Katriel. 2006. Global constraints. F. Rossi, P. Van Beek, T. Walsh, eds. *Handbook of Constraint Programming*. Elsevier, Amsterdam, 169–208.

Xiang, Z., C. Chu, H. Chen. 2008. The study of a dynamic dial-a-ride problem under time-dependent and stochastic environments. *Eur. J. Oper. Res.* **185**(4) 534–551.

Yuen, C. W., K. I. Wong, A. F. Han. 2009. Waiting strategies for the dynamic dial-a-ride problem. *Internat. J. Environment Sustainable Dev.* **8**(3–4) 314–329.