TRANSPORTATION SCIENCE
Vol. 45, No. 3, August 2011, pp. 399–412
ISSN 0041-1655 | EISSN 1526-5447 | 11 | 4503 | 0399

DOI 10.1287/trsc.1100.0336
© 2011 INFORMS

Checking the Feasibility of Dial-a-Ride Instances Using Constraint Programming

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

Gilles Pesant
Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montréal, Montréal H3C 3A7, Canada, pesant@polymtl.ca

Louis-Martin Rousseau
Département de Mathématiques et de Génie Industriel, École Polytechnique de Montréal, Montréal H3C 3A7, Canada, louis-martin.rousseau@polymtl.ca

In the dial-a-ride problem (DARP), a fleet of vehicles must serve transportation requests made by users that need to be transported from an origin to a destination. In this paper we develop the first exact algorithm which is able to either efficiently prove the infeasibility or to provide a feasible solution. Such an algorithm could be used in a dynamic setting for determining whether it is possible or not to accept an incoming request. The algorithm includes solution space reduction procedures, and filtering algorithms for some DARP relaxations. Computational results show that the filtering algorithms are effective and that the resulting algorithm is advantageous on the more constrained instances.

Key words: dial-a-ride problem; algorithms; partial routes; scheduling; dynamic programming; constraint programming
History: Received: July 2009; revision received: June 2010; accepted: June 2010. Published online in *Articles in Advance* September 24, 2010.

1. Introduction

The *dial-a-ride problem* (DARP) is a *pickup and delivery problem* (PDP) in which a fleet of vehicles must serve requests which consist in transporting users from an origin to a destination. The main application of the DARP is the door-to-door transportation services offered for the elderly and handicapped people in many cities. Case studies have been conducted in Toronto (Desrosiers, Dumas, and Soumis 1986); Berlin (Borndörfer et al. 1997); Bologna (Toth and Vigo 1996); Copenhagen (Madsen, Ravn, and Rygaard 1995); and Brussels (Rekiek, Delchambre, and Saleh 2006). For a review of the different models and algorithms for the DARP, the reader is referred to Cordeau and Laporte (2007).

The DARP generalizes many problems of the vehicle routing literature, such as the *capacitated vehicle routing problem* (CVRP) and the *traveling salesman problem with time windows* (TSPTW), among others. Because the feasibility problem for the TSPTW is NP-complete (Savelsbergh 1985), checking whether a DARP instance is feasible or not is also NP-complete. In addition to the precedence constraints that are not present in the TSPTW and CVRP, the DARP generally assigns tight time windows to pickup and delivery vertices as well as a maximum user trip time to reduce their inconvenience. These constraints make the problem of finding a feasible solution for the DARP a challenge.

Detecting whether a DARP instance is feasible is relevant in static and in dynamic settings. In a static setting, finding a feasible solution could be the first step inside an optimization algorithm that is executed the night before the service day. Even on the cases where the feasible solution found is not close to the optimal, many local search methods are known to perform better when a feasible solution is given. In a dynamic setting, an algorithm to detect whether or not a DARP instance is feasible can be used as a tool to accept or reject incoming user requests as follows. When a new request is received, a new DARP instance I is constructed that contains all the previously known requests plus the new incoming request. The algorithm used to find a feasible solution is executed with the constraint that the new solution for I must not modify the partial routes already traveled. If such a feasible solution is found, the request can be accepted whereas, if no solution exists, the request is rejected. From a quality-of-service point of view, proving that a given request cannot be inserted, while satisfying the constraints, is a more convincing statement than simply saying that no solution has been found.

In this article, we present an algorithm to determine whether a given instance of the DARP is feasible, based on constraint programming, which is particularly effective to solve feasibility problems. To achieve this, we have modeled the DARP as a constraint satisfaction problem (CSP), and we then developed filtering algorithms to determine whether or not a partial solution can be extended into a complete solution, taking into account different sets of constraints. Results show that the algorithm becomes advantageous on the more constrained instances. Its strength lies in finding solutions where feasible solutions are hard to find and also on cases where a certificate of infeasibility becomes relevant. Two applications in which our method could also be useful are the problem of scheduling the transfer of patients inside and between hospitals (Beaudry et al. 2010) and the transfer of personnel to sea drilling platforms, where the number of helicopters is limited and weather conditions impose strict time windows (Velasco-Rodríguez 2006). In addition, the filtering algorithms proposed in this article could also be applied in other contexts, such as the implementation of arc-exchange procedures like *k-opt* (De Backer, Furnon, and Shaw 2000) to eliminate insertions positions that would lead to an infeasible solution.

The remainder of this paper is organized as follows. In §2 we give a brief introduction to constraint programming (CP), and we survey the most relevant literature on the use of CP to solve vehicle routing problems. The CP model of the DARP and some important definitions that are needed later are given in §§3 and 4, respectively. Section 5 describes two algorithms we have developed to use in CP to speed up the search. The methods for selecting variables and values in the branching tree are described in §6, while techniques to reduce the search space are described §7. Finally, computational results and some conclusions are given in §§8 and 9, respectively.

2. Constraint Programming and Its Applications to Vehicle Routing Problems

Constraint programming (CP) is a programming paradigm applicable to the solution of combinatorial problems and based on inference and search techniques (Rossi, van Beek, and Walsh 2006). In CP, a problem is modeled as a *constraint satisfaction problem* (CSP). 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 combination of values from the variable domains are permitted and which of them are not.

A CSP is a triple $P = \langle X, D, C\rangle$, where $X = (x_1, \ldots, x_n)$ is an n-tuple of variables, $D = (D_1, \ldots, D_n)$ is an n-tuple of domains, such that $x_i \in D_i$ for $i \in= \{1, \ldots, n\}$ and $C = \{C_1, \ldots, C_m\}$ is a set of constraints. A *constraint* C_i is defined on a subset $\{x_{i_1}, \ldots, x_{i_k}\}$ of the variables in X and describes the allowed combinations of values for these variables as a subset of $D_{i_1} \times \cdots \times D_{i_k}$.

A tuple $A = (a_1, \ldots, a_n)$, such that $a_i \in D_i$, is said to *satisfy* a constraint $C_i = \langle R_{S_i}, S_i\rangle$ of the CSP P if the projection of A onto S_i belongs to R_{S_i}. A *solution* for the CSP P is a tuple $A = (a_1, \ldots, a_n)$ such that $a_i \in D_i$, and it satisfies every constraint $c \in C$. A CSP is *consistent* (or *feasible*) if it has a solution and *inconsistent* (or *infeasible*) otherwise.

CP solves a model using inference algorithms to reduce the search space, and search methods to explore the space. 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 consist generally of backtracking or branch and bound combined with constraint propagation.

Constraint propagation algorithms manage the scheduling of specific rules to reduce the domain of the variables of a CSP. These algorithms generally terminate when they can achieve a property called *domain consistency*. On a CSP, this property states that for every constraint C_i and for each domain value β of every variable x_{i_j} of C_i, there exists a tuple $\alpha \in S_i$ whose value at position j is β. Constraints can be partitioned according to the number of variables they relate. A constraint is said to be *unary* when only one variable is affected (e.g., the constraint $x_1 < 5$). When a constraint affects two variables, it is called *binary*, such as the constraint $x_1 + x_2 < 5$. A *global constraint* is a constraint that relates a nonfixed number of variables. A well-known example of a global constraint is the constraint *allDifferent*$(x_1, \ldots, x_n)$, which states that the variables $x_1, \ldots, x_n$ must be pairwise different. Generally, a global constraint can be substituted by a set of simpler constraints. However, global constraints are usually preferred: because of their global view, their filtering algorithms are able to eliminate more values from the variable domains than by using the equivalent set of simpler constraints. Whenever a filtering algorithm eliminates all the values from a variable domain, it means that the actual CSP is infeasible and the filtering algorithm returns "fail." There are many global constraints in the constraint programming literature (see, e.g., Beldiceanu, Carlsson, and Rampon 2005; van Hoeve and Katriel 2006). Most global constraints studied in the literature have polynomial time algorithms to achieve domain consistency, such as the *allDifferent* constraint.

There are many fields in which CP is being successfully applied, such as scheduling, planning, molecular biology, finance, and numerical analysis. These and other applications of CP are surveyed in Rossi, van Beek, and Walsh (2006).

The construction of vehicle routes constitutes a major problem for many organizations and has been studied in the literature for more than 50 years (Laporte and Osman 1995). We describe the main applications of constraint programming to solve vehicle routing problems that were carried out in the last decade. In the next paragraph we survey the literature on exact methods, while in the one that follows we focus on heuristics.

A constraint programming algorithm to solve efficiently small instances (up to 30 vertices) of the TSP was presented by Caseau and Laburthe (1997). The authors have proposed a propagation technique to prohibit subtours, a simple binary branching scheme and a bounding procedure weaker (but faster) than the assignment problem. Pesant et al. (1998) have developed a constraint programming algorithm to solve the TSPTW. By maintaining a lower bound and an upper bound at each node, the search tree is explored by a branch and bound. Their algorithm obtained better results than heuristics on some difficult instances and provided new best solutions (at that time) for some others. Pesant et al. (1999) have extended their previous CP model for the case in which multiple time windows for each vertex are present. Focacci, Lodi, and Milano (2002) have proposed a hybrid algorithm for the TSPTW, based on the model of Pesant et al. (1998), where they have included an effective global constraint based on cost propagation. Computational results from a set of symmetric and asymmetric instances show the effectiveness of the inclusion of cuts in particular and of this hybrid method in general. Constraint programming was also used in vehicle routing to solve the subproblem of a column-generation approach. Rousseau et al. (2004) have solved, with constraint programming and new redundant constraints, the resource constrained shortest-path problem that appears as a subproblem in the TSPTW and VRPTW. The use of constraint programming in column generation was originally proposed by Junker et al. (1999) for solving a crew assignment problem.

A large neighborhood search (LNS) for the VRP and the VRP with time windows was developed by Shaw (1998). In his algorithm, constraint programming is used to optimize the reinsertion of the set of clients removed at each move of the LNS. This method was competitive and it was able to find, at that time, new best solutions for some instances. A hybrid method to solve the vehicle routing problem with time windows is presented by Rousseau, Gendreau, and Pesant (2002). Their method is based on the work of Pesant and Gendreau (1996) that describes how a constraint framework can be used to explore large neighborhoods efficiently using a branch-and-bound procedure. The hybrid algorithm of Rousseau, Gendreau, and Pesant (2002) uses three operators, which define three different neighborhoods. During neighborhood exploration through branch and bound, propagation and pruning are used to reduce the search space. The resulting method has produced good results on all Solomon's benchmark problems (Solomon 1987). De Backer, Furnon, and Shaw (2000) have developed local search and metaheuristics algorithms for the vehicle routing problem that make use of constraint programming. The idea is to use constraint programming as an efficient way to tell whether a solution is valid or not and to determine the values of constrained variables. The search of solutions is handled by the local search algorithm. Their method relies on a representation, called *active*, that holds the constrained variables and is where constraint propagation takes place. These techniques were embedded in the commercial package ILOG Dispatcher, which is widely used to solve the VRPTW and its variants. Computational results over a set of benchmark instances for the VRP have shown that the method is effective.

3. Problem Definition and a Constraint Programming Model

The DARP can be defined as follows. Let $G = (V, A)$ be a complete and directed graph with vertex set $V = \{0\} \cup R$, where vertex 0 represents the depot, and R ($|R| = 2n$) represents the customer vertices. The set R is partitioned into sets R^+ (pickup vertices) and R^- (delivery vertices). Each arc $(i, j) \in A$ has a nonnegative travel time T_{ij} satisfying the triangle inequality. 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 $D_0 = 0$ and $q_0 = 0$). Let $H = \{1, \dots, n\}$ be the set of requests and let L be the maximum ride time for any request. Request i has pickup vertex $i^+ \in R^+$ and delivery vertex $i^- \in R^-$, and its load is $q_{i^+} = -q_{i^-}$. Let $K = \{1, \dots, m\}$ be a set of available vehicles, each of capacity Q. A *route* is a circuit over some vertices, starting and finishing at the depot. The DARP consists of constructing m vehicle routes (possibly empty) such that:

(i) for every request i the pickup vertex and the delivery vertex are visited by the same route, and the pickup vertex is visited before the delivery vertex;

(ii) the load of the vehicles never exceeds their capacity at any time;

(iii) the ride time of each user is at most L;

(iv) the service at vertex i begins in the interval $[E_i, L_i]$.

We now give a standard formulation of the DARP as a constraint satisfaction problem based on successor variables. Some definitions and results presented in the remainder of this article will be based on this formulation. We first extend the graph G. 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 $i \in K = \{1, \dots, m\}$ is represented by vertices $start(i) \in V^+$ (starting depot) and $end(i) \in V^-$ (ending depot). Under this transformation, the route of vehicle i is represented by the circuit $(start(i))$: S_i: $(end(i))$ where S_i 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;

(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 $j \in K$, $s[end(j)] = start(j)$;

(ii) for each vehicle $j \in K$, $v[end(j)] = v[start(j)] = j$;

(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 windows 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 i, $\ell[start(i)] = 0$;

(ix) for each client vertex $j \in R$, $\ell[s[j]] = \ell[j] + q_{s[j]}$ and $\ell[j] \le Q$;

Ride time constraints:

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

During the search to obtain a feasible solution to a DARP instance using the constraint programming model, the successor variables s are given values one at a time. The fixed successor variables create partial routes that the constraint programming algorithm will try to extend into complete routes to obtain a feasible solution. If at some point during this process a constraint propagation algorithm realizes that no solution can exist from that node of the tree, backtracking occurs. In the next section, we formalize the concept of partial route, partial solution, and what it means to extend a partial solution into a complete solution. In later sections we develop filtering algorithms for two relaxations to help the constraint programming engine to backtrack from infeasible subtrees earlier and therefore speed up the search.

4. Partial Routes, Partial Solutions, and Extensions

A *partial route* of a graph $G = (V, E)$ is a sequence of vertices of V that do not repeat. Consider the DARP constraint programming formulation given in §3. A *partial solution* of a DARP instance I consists of a set of partial routes P of G such that each vertex $i \in D \cup R$ appears exactly in one partial route $p \in P$. A solution S of the DARP instance I is said to *extend* a partial solution P if every partial route $p \in P$ is contained as a subsequence in one of the vehicle routes of the solution S. Given an instance of the DARP and a partial solution P, we are interested in determining whether or not there exists a solution for the DARP that extends the partial solution P. We call this problem the *partial route extension problem*. Observe that this problem generalizes the feasibility problem of the DARP, because the latter can be seen as a special case in which each partial route is made up of just one vertex. Determining whether or not a DARP instance is feasible is NP-complete (Savelsbergh 1985) and therefore, the *partial route extension problem* is also NP-complete. Consider now any relaxation r of the DARP. Given a partial solution P, the *partial route extension problem* of the relaxation r consists of determining whether or not there exists a solution of the relaxation r that extends the partial solution P.

We give some notation and definitions regarding partial routes. Consider the partial route $p = (p_0, \dots, p_u)$. We define $\alpha(p) = \max\{\sum_{i=0}^{j} q(p_i): 0 \le j \le u\}$, $\delta(p) = \sum_{i=0}^{u} q(p_i)$, and $\gamma(p) = \max\{-\sum_{i=0}^{j} q(p_i): 0 \le j \le u\}$. Thus, $\alpha(p)$ accounts for how much more load will the vehicle attain along the path p with respect to the load it had at the beginning. The value $\delta(p)$ accounts for the difference between the vehicle load after and before the partial route p. Finally, $\gamma(p)$ accounts for how much less load the vehicle will attain along the path p in respect with the load it started with. See Figure 1 for an illustration. Observe that $-\delta(p) \le \gamma(p)$ and that $\alpha(p) \ge \delta(p)$ for any partial route p. Given partial routes p_1 and p_2, the partial route p that consists of the concatenation of the partial routes p_1 and p_2 (in this order) is written $p = (p_1, p_2)$.

A *route* $r = (v_0, \dots, v_k)$ of a DARP instance is a sequence of vertices such that (1) $v_0 = start(i)$ and $v_k = end(i)$ for some $i \in K$; (2) for each $i \in H$, $i^+ \in \{v_0, \dots, v_k\}$ if and only if $i^- \in \{v_0, \dots, v_k\}$; and (3) $v_i \neq v_j$ for all $1 \le i, j \le k$ with $i \neq j$. A route is said to be *empty* if

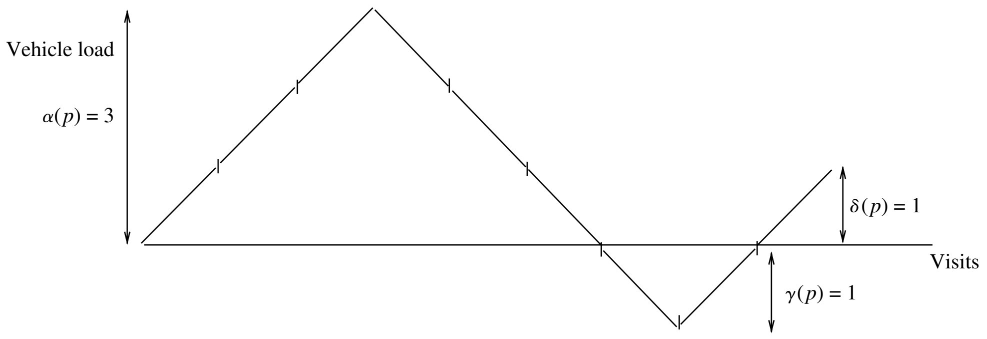

Figure 1 An Example of a Partial Route with a Sequence of Three Pickups, Followed by Four Deliveries and Finishing with Two Pickups

$k = 1$. We say that a vertex i belongs to the route $r = (v_0, \ldots, v_k)$ and we write it, $i \in r$, if $i = v_j$ for some $0 \leq j \leq k$. We say that a request $i \in H$ belongs to the route if $i^+ \in r$ and $i^- \in r$. Similarly, a vertex x belongs to a partial route $p = (v_1, \ldots, v_u)$, written $x \in p$, if $x = v_i$ for some $1 \leq i \leq u$. A vertex x belongs to a set of partial routes $P = \{p_1, \ldots, p_k\}$, written $x \in P$, if $x \in p_i$ for some $1 \leq i \leq k$.

Given a partial solution S for a DARP instance I, we say that S is *elementary feasible* if S can be extended into a complete DARP solution that satisfies the constraints from (i) to (v), i.e., the basic constraints. A solution satisfying the basic constraints consists of m routes such that (i) each vertex is visited exactly once and (ii) the pickup vertex and the delivery vertex of the same request are visited in the same route. In the development of the filtering algorithms, we assume that the partial solution that is taken because the input is elementary feasible. This is because the constraint programming engine we used (ILOG Solver) will efficiently verify whether a partial solution S is elementary feasible at each branch when given the CSP model of §3.

5. Filtering Algorithms

In this section we develop an inconsistency checking algorithm and a filtering algorithm for two relaxations of the DARP called the *pickup and delivery problem with fixed partial routes* and the *basic DARP with ride time constraint*. We now define the two relaxations by stating the constraints that are considered by each of them.

The *pickup and delivery problem with fixed partial routes* (PDP-FPR) relaxation takes into account the basic constraints, the precedence constraints, and the capacity constraints. Specifically, this relaxation has the constraints (i) to (ix) of the constraint programming model presented in §3. It is also assumed that the time windows are unbounded, i.e., the domain of the variables t is $[0, \infty]$. Therefore, a feasible solution of the PDP-FPR consists of m vehicle routes such that the pickup and delivery vertices of each request are both in the same route, the pickup vertex precedes the corresponding delivery vertex, and the capacity of each vehicle is never exceeded. In §5.1.2 we present a consistency checking algorithm for this relaxation based on dynamic programming.

The *basic DARP with ride time constraint* relaxation has the constraints (i) to (v) and constraint (x) of the constraint programming model. This relaxation takes into account only the basic constraints and the maximum ride time constraint. A feasible solution therefore consists of m vehicles routes such that the pickup and delivery vertices of each request are both in the same route and the maximum ride time is never exceeded. A partial filtering algorithm for this constraint is given in §5.2.

5.1. Pickup and Delivery Problem with Fixed Partial Routes

The problem of determining the feasibility of *pickup and delivery problem with fixed partial routes* is strongly NP-complete (Berbeglia, Pesant, and Rousseau 2011). In this section, we present an exact consistency checking algorithm for the PDP-FPR based on dynamic programming. We begin by giving some definitions about partial solutions.

Let I be an instance of the DARP and S be a partial solution of I. A set of partial routes $C \subseteq S$ is said to be *complete* if (1) for each request $i \in H$, either $i^+ \in C$ and $i^- \in C$, or $i^+ \notin C$ and $i^- \notin C$ (2) for each vehicle $j \in K$, either start $(j) \in C$ and end$(j) \in C$, or start $(j) \notin C$ and end$(j) \notin C$. A set of partial routes $M \subseteq S$ is said to be *minimally complete* if M is complete and there is no complete set M' that is nonempty and is a proper subset of M (see Figure 2 for an example).

Consider now the set of partial routes S of a partial solution. The set S can be partitioned in a unique way into a family of minimally complete partial sets. We call this family $\mathcal{F}(S)$. Observe also that in any solution satisfying the PDP-FPR relaxation that extends the partial solution S, the vertices that are contained in a minimally complete set $C \in \mathcal{F}(S)$ must be served by the same vehicle.

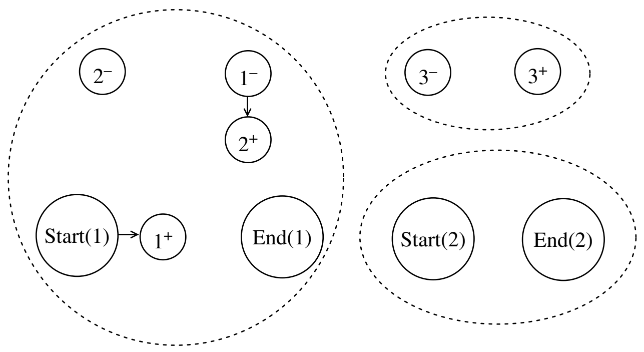

Figure 2 A Partial Solution of a Two-Vehicles and Three-Requests Instance of the DARP, Exhibiting the Three Minimally Complete Sets

Let $\mathcal{F}(S)$ be the partition of a partial solution S. If the partial solution S is elementary feasible, then the depot vertices of different vehicles must be in different sets of the partition. Otherwise, it would mean that there must be a route that visits two depots associated with different vehicles, which is impossible. Then the family partition can be written as $\mathcal{F}(S) = \{C_1, \ldots, C_m, E_1, \ldots, E_\rho\}$ with $\rho \geq 0$. The set C_i is the minimally complete set of partial routes that has the depot vertices associated to the vehicle i. Each set C_i, with $1 \leq i \leq m$, is called a *depot associated minimally complete set*. The sets E_i with $1 \leq i \leq \rho$, which do not have any depot vertices, are called *depot independent minimally complete sets*.

Observe that the precedence constraints between the pickup and the delivery of each request induce precedence constraints between the partial routes in S. Consider a depot independent minimally complete set $A \subseteq S$. We define the relation $\hat{R}(A)$ as follows. A pair $(a, b) \in A^2$ belongs to $\hat{R}(A)$ if there is a request i such that $i^+ \in a$ and $i^- \in b$. Now we define the relation $<_p$ over A as the transitive closure of the relation $\hat{R}(A)$. Observe that the relation $<_p$ over A is a partial order (i.e., it is antisymmetric and transitive) if and only if $\hat{R}(A)$ is acyclic. Note that a cycle in the relation $\hat{R}(A)$ certifies that there is no way to schedule the partial routes along a single route in such a way that the precedence constraint is respected. Thus, any instance that has a depot-independent minimally complete set A such that $\hat{R}(A)$ has a cycle is infeasible and can be discarded efficiently. For clarity of exposition we assume from now on that the relations $\hat{R}(A)$ for any depot-independent minimally complete set are acyclic.

Consider now a minimally complete set $B \subseteq S$ associated to a depot. Let $s \in B$ be the partial route associated with the starting depot and t be the partial route associated with the ending depot. We define

$$\hat{R}(B) = \{(s,x)\in B^2 \mid x\in B\backslash\{s\}\}\cup\{(y,t)\in B^2 \mid y\in B\backslash\{t\}\} \cup\{(a,b)\in B^2 \mid \exists i\in H \text{ such that } i^+\in a \text{ and } i^-\in b\}.$$

In words, an ordered pair (x, y) belongs to $\hat{R}(B)$ when x is the partial route associated with the starting depot or y is the partial route associated with the ending depot, or when x has a pickup vertex i^+ whose associated delivery vertex i^- is at y. The relation $<_p$ over B is defined as the transitive closure of the relation $\hat{R}(B)$. As in the case of depot independent minimally complete sets, the relation $<_p$ over B is a partial order if and only if $\hat{R}(B)$ is acyclic. It holds also that a cycle in $\hat{R}(B)$ means that the associated instance is infeasible, and we therefore assume from now on that the relations $\hat{R}(B)$ for any depot-independent minimally complete set are acyclic.

5.1.1. A Dynamic Programming Algorithm. Given a DARP instance I and a partial solution S, the algorithm we present first decomposes the problem into a set of subproblems, each associated to a *minimally complete set* of S (see definition in §5.1). A dynamic programming algorithm is used to solve a decision problem associated with each of the subproblems. Using the results of each subproblem, we are then able to determine whether or not S can be extended into a PDP-FPR solution.

Consider a DARP instance I, an elementary feasible partial solution S and a depot-independent minimally complete set $A = \{a_1, \ldots, a_k\} \subseteq S$. If the relation $<_p$ over A is a partial order, we define $V(A)$ to be all the orders (or permutations) of the elements in A, $\sigma \in S_k$, such that they extend the partial order $<_p$. Formally,

$$V(A) = \{\sigma \in S_k \text{ such that if } a_i <_p a_j, \text{ then } \sigma(i) < \sigma(j) \text{ for all } i, j \in \{1, \ldots, k\}\}.$$

We define *height*(S) as the minimum capacity that a vehicle must have in order to serve all partial routes in A, respecting the precedence constraints. Formally,

$$\mathit{height}(A) = \min\left\{\max\left\{\sum_{j=1}^{i-1}\delta(a_{\sigma(j)}) + \alpha(a_{\sigma(i)}) \mid i=1,\ldots,k\right\} \mid \sigma\in V(A)\right\}.$$

Similarly, given a depot-associated minimally complete set $B = \{b_1, \ldots, b_k\}$, we define

$$V(B) = \{\sigma \in S_k \text{ such that if } b_i <_p b_j, \text{ then } \sigma(i) < \sigma(j) \text{ for all } i, j \in \{1, \ldots, k\}\},$$

where the partial order $<_p$ over B is the one defined at the beginning of §5.1. The *height* of B is defined in the same way as for the depot-independent minimally complete sets, i.e.,

$$\mathit{height}(B) = \min\left\{\max\left\{\sum_{j=1}^{i-1}\delta(b_{\sigma(j)}) + \alpha(b_{\sigma(i)}) \mid i = 1, \ldots, k\right\} \mid \sigma \in V(B)\right\}.$$

Consider now a depot-independent minimally complete set C. How can we tell whether or not it is possible put together in a feasible route the partial routes of C and those of the depot-associated minimally complete set B? The fact that $height(B) \leq Q$ and $height(C) \leq Q$, does not necessarily mean that this is possible. For example, consider a DARP instance of five requests with unitary loads and with vehicle capacities of 3. Let $B = \{(start(1), 1^+, 2^+), (1^-, 2^-, end(1))\}$ and $A = \{(3^+, 4^+, 5^+), (3^-, 4^-, 5^-)\}$. Observe that $height(B) = 2$ and $height(A) = 3$. However, it is impossible to put the partial routes of A together with the partial routes of B in such a way that the route obtained is feasible with respect to the capacity and the precedence constraints.

Consider a depot-associated minimally complete set $B = \{b_1, \ldots, b_k\}$. To be able to determine whether or not a depot-independent minimally complete set A can be put together with a depot-associated minimally complete set B into the same route, we define the *minimum insertion point* of B as

$$\kappa(B) = \min\left\{\left\{\sum_{j=1}^{i} \delta(b_{\sigma(j)}) \mid i = 1, \ldots, k-1,\ \sigma \in V(B)\right\} \cup \{\infty\}\right\}.$$

Observe that if $|B| = 1$, then $\kappa(B) = \infty$. The minimum insertion point of a depot-associated minimally complete set B can be explained as follows. Suppose we want to serve with a single vehicle all the requests that are in the set B, as well as a new request r extending the partial routes of B. The minimum insertion point of B represents the minimum load possible that the vehicle can have just before performing the pickup of the request r.

Using the definition of κ and of *height* we can prove the following.

Theorem 1. *Let I be a single vehicle instance of the DARP and let S be a partial solution. Let $\mathcal{F}(S) = \{B, E_1, \ldots, E_k\}$ $(k > 0)$ be the partition of S into minimally complete sets such that B denotes the only depot-associated minimally complete set. The partial solution S can be extended into a feasible PDP-FPR solution for the instance I if and only if $\kappa(B) + \max\{height(E_j) \mid j = 1, \ldots, k\} \leq Q$.*

Proof. We denote the partial routes of B by $B = \{b_1, \ldots, b_l\}$ and the partial routes of E_j by $E_j = \{e_j^1, \ldots, e_j^{k_j}\}$. Now consider a permutation $\sigma^* \in V(B)$ and an integer $1 \leq i^* \leq l - 1$ such that $\kappa(B) = \sum_{j=1}^{i^*} \delta(b_{\sigma^*(j)})$. Also consider a permutation $\sigma_j^\dagger \in V(E_j)$ and the integer $1 \leq i^\dagger \leq k_j$ such that $height(E_j) = \alpha(a_{\sigma_j^\dagger(i^\dagger)}) + \sum_{j=0}^{i^\dagger - 1} \delta(a_{\sigma_j^\dagger(j)})$.

Suppose first that S can be extended into a feasible PDP-FPR solution. Consider any route for I feasible for the PDP-FPR. In any such route, the maximum load that the vehicle will attain cannot be smaller that the maximum load of the route that we now describe. First, order the partial routes of B according to the permutation $\sigma^* \in V(B)$. The load of the vehicle will reach the value $\kappa(B)$ at the end of the partial route $\sigma^*(i^*)$. Observe that the load at this point is the lowest possible to insert other partial routes among any feasible route composed of the partial routes of B (feasible with respect to the PDP-FPR). Between the partial routes $\sigma^*(i^*)$ and $\sigma^*(i^* + 1)$ we now insert one by one each of the minimally complete sets E_j for each $j = 1 \ldots, k$. When adding the partial routes of a particular minimally complete set, say E_ℓ, the order in which the partial routes E_ℓ are inserted is given by the permutation $\sigma_j^\dagger$ associated to E_ℓ. Now observe that the maximum load attained by the vehicle is $\kappa(B) + \max\{height(E_j) \mid j = 1, \ldots, k\}$. Thus, if the instance I can be extended into a feasible PDP-FPR solution, we have that $\kappa(B) + \max\{height(E_j) \mid j = 1, \ldots, k\} \leq Q$. Now suppose that $\kappa(B) + \max\{height(E_j) \mid j = 1, \ldots, k\} \leq Q$. Because the vehicle route we have constructed (in the previous paragraph) respects the precedence constraints and the maximum load is at most Q, then the partial solution S of the instance I can be extended into a PDP-FPR solution. □

5.1.2. Computing the Height and the Minimum Insertion Point.

We present first a dynamic programming algorithm to compute the height of a minimally complete set. Consider a minimally complete set $A = \{a_1, \ldots, a_k\}$. We recall that $height(A) = \min\{\max\{\sum_{j=0}^{i-1} \delta(a_{\sigma(j)}) + \alpha(a_{\sigma(i)}) \mid i = 1, \ldots, k\} \mid \sigma \in V(A)\}$. Before describing the algorithm we give some definitions. A nonempty subset $K \subseteq A$ is said to be *completed by precedence* if for every partial route $k \in K$, all predecessors of k are also in K. The family of subsets of A that are completed by precedence is denoted as $\mathscr{C}(A)$. Given a set of partial routes X, we define $fin(X) = \{x \in X \mid \nexists y \in X \text{ such that } x <_p y\}$. In words, a partial route x belongs to $fin(X)$ if x is not the predecessor of some other partial route in X. Finally, we define $\delta(X) = \sum_{x \in X} \delta(x)$, i.e., the sum of values of δ of the partial routes in X.

The dynamic algorithm is based on the following equations, which define the height of any subset A of S that is completed by precedence.

$$height(\varnothing) = 0. \tag{1}$$

$$height(A) = \min\{\max\{height(A\backslash\{i\}), \delta(A\backslash\{i\}) + \alpha(i)\},\ i \in fin(A)\} (\text{with } A \in \mathscr{C}(S)). \tag{2}$$

The recursive Equation (2) allows us to compute the height of the minimally complete set efficiently using dynamic programming. To do so, we need a procedure to enumerate all the subsets A of S that are completed by precedence in an order such that

$A\backslash\{i\}$ is enumerated before A. To this end we have used the enumerative procedure presented by Schrage and Baker (1978). In this procedure (see Algorithm 1), given a subset completed by precedence $A \subseteq S$, the next subset A' that is completed by precedence is returned. Therefore, to compute $height(S)$, the number of recursive calls is equal to the total number of subsets of S completed by precedence, which is always less than or equal to $2^{|S|} - 1$. The maximum number of recursive calls is $2^{|S|-2} + 1$, and this limit is reached in the case where the partial order is defined as follows. There is a vertex v that precedes all the others, there is a vertex w preceded by all the others, and no other precedence relation is present. However, the number of subsets completed by precedence is considerably less in practice. In our computational results, the maximum number of subsets completed by precedence was 54. Observe that the dynamic programming algorithm to compute the height, defined by Equations (1) and (2), works for depot-independent as well as depot-associated minimally complete sets.

Algorithm 1 (Enumerative procedure, Schrage and Baker 1978):

```
Input: (i) A set of partial routes $S = \{1, \ldots, k\}$
    such that whenever $i$ precedes $j$ (i.e., $i <_p j$)
    then $i < j$. (ii) A current subset $A \subseteq S$ completed
    by precedence. The subset $A$ is represented by
    a vector $v$ such that $i \in A \Leftrightarrow v[i] = 1$.
Output: The next subset of $A$, $A' \subseteq S$ such that $A'$
    is completed by precedence.
Find the smallest partial route index $j$ such that
    $v[j] \neq 1$. (If $v[j] = 1$ for all $j = 1, \ldots, k$ then all
    subsets completed by precedence were already
    enumerated.)
Set $v[j] = 1$
for $i = j - 1$ to 1 do
if $v[i] = 1$ and $i \in fin(A)$ then
$v[i] = 0$
end if
end for
return $v$
```

In a similar way, given a depot-associated minimally complete set B we can calculate the minimum insertion point $\kappa(B)$. Assume that $s \in B$ is the partial route associated with the starting depot and $t \in B$ is the partial route associated with the ending depot. We assume that $|B| > 1$, otherwise $\kappa(B) = \infty$. $\kappa(B)$ can be calculated recursively using the following equations.

$$\kappa(A = \{s\}) = \delta(s). \tag{3}$$

$$\kappa(A) = \min\{\min\{\kappa(A\backslash\{i\}), \delta(A\backslash\{i\}) + \delta(i)\} i \in fin(A)\}$$
$$(\text{with } A \in \mathscr{C}(B) \text{ and } t \notin A). \tag{4}$$

$$\kappa(B) = \kappa(B\backslash\{t\}). \tag{5}$$

5.1.3. Inconsistency Checking Algorithm for the PDP-FPR. Now that we have procedures to compute $height(A)$ and $\kappa(B)$ for any minimally complete set A and any depot associated minimally complete set B, we are able to decide whether or not a partial solution S can be extended into a feasible PDP-FPR solution. This procedure is described in Algorithm 2.

Algorithm 2 (Inconsistency checking algorithm for the PDP-FPR):

```
Partition the partial solution $S$ into the family of
    minimally complete sets. $\mathscr{F}(S) = \{C_1, \ldots, C_m,$
    $E_1, \ldots E_j\}$. The sets $\{C_1, \ldots, C_m\}$ are depot
    associated complete sets while the others sets in
    the family are depot independent complete sets.
if $\hat{R}(X)$ has a cycle for some $X \in \{C_1, \ldots, C_m,$
    $E_1, \ldots E_j\}$ then
fail
end if
Compute $height^\dagger = \max\{height(C_1), \ldots, height(C_m),$
    $height(E_1), \ldots, height(E_j)\}$.
if $height^\dagger > Q$ then
fail
end if
if $j > 0$ then
Compute $height^* = \max\{height(E_1), \ldots, height(E_j)\}$.
Compute $\kappa^* = \min\{\kappa(C_1), \ldots, \kappa(C_m)\}$.
if $\kappa^* + height^* > Q$ then
fail
else
return true
end if
end if
```

This filtering algorithm is applied every time the successor of a vertex is fixed. Suppose the successor variable of vertex x becomes fixed to vertex y. How different is the new partial solution S' with respect to the partial solution S that we had just before this assignment? It is clear that the total number of partial routes in S' is one less than the total number of partial routes in S. If the partial routes associated to vertices x and y belonged to the same minimally complete set, then the number of minimally complete sets in S remains the same. If the partial routes belonged to different minimally complete sets, these two sets would be joined, and then there would be in S' one fewer minimally complete set. Observe that it is not necessary to compute the height of each minimally complete set and the minimum insertion point of each depot-associated minimally complete set. It is indeed sufficient to compute these values only for the modified minimally complete set, because all of the others remain the same and we can use their values of the previous partial solution S. This fact allows the filtering to be performed much faster.

An upper bound on the complexity of the algorithm is $O(2^{|S|})$, where S is the cardinality of the largest minimally complete set. The size of S is bounded by n, but it is generally much less in practice (in our computational tests, the largest size of S was 14). It would be possible to remove values from the variable domains instead of only producing a fail. However, this will increase the computational complexity, and the overall efficiency of CP algorithm will not be improved.

5.2. Partial Filtering Algorithm for the Ride Time Constraint

The ride time constraint in the DARP states that for every request, the difference between the time at which the service at its associated delivery vertex begins and the time at which the service at its associated pickup vertex ends cannot be greater than a given value. In our constraint programming model of the DARP, routes are constructed by assignments of the successor variables s, without any particular order. As a consequence, it is generally the case that the domains of the variables t (i.e., the time at which each pickup or delivery vertex is served) are not reduced until the final stages of the route construction. Unfortunately, this means that inconsistencies for the ride time constraint are only detected late in the search tree, even if a partial route originated at the beginning of the search tree was already infeasible. To overcome this difficulty we propose a filtering algorithm for the ride time constraint.

We now consider the *basic DARP with ride time constraint* relaxation of the DARP, which, as it was stated, has only constraints (*i*) to (*v*) (basic constraints) and constraint (*x*) (ride time constraint).

The partial route extension problem for the *basic DARP with ride time constraint* was proved to be NP-complete in Berbeglia, Pesant, and Rousseau (2011) as it is equivalent to the *uncapacitated pickup and delivery problem with fixed partial routes and ride times*. We now present a partial filtering algorithm for this relaxation which runs in $O((n+m)^2)$ time. For the clarity, we assume in this section that the service times are equal to zero. The modification of the filtering algorithms for the cases of nonzero service times is straightforward.

At each node of the search tree, the ride time filtering algorithm keeps track of the following information:

—*Begin*[i]: specifies the starting vertex of the partial route of vertex i.

—*Ends*[i]: specifies the ending vertex of the partial route of vertex i.

—*TravelTime*[i]: the time elapsed between the service at vertex i is finished and the time at which the end vertex of the partial route of i begins to be served.

—*TravelTimeBack*[i]: the time elapsed between the beginning of service at the first vertex of the partial route of i and the beginning of service at vertex i.

—*RequestBind*[i]: a binary variable that states if the pickup and delivery vertices of request i belong to the same partial route.

Before presenting the ride time forward and backward filtering algorithms, we define some notation similar to that of Pesant et al. (1998). We denote by *vIndex* the vertex at which $s[vIndex]$ became fixed and *vNext* its value. We call P_{first} the partial route that finishes at vertex *vIndex*. Finally, we note by P_{last} the partial route that starts at vertex *vNext*. We have added a new CP variable array into the model, called *Prevs*, that represents the possible predecessors of each vertex. Thus, $j \in Prevs[i]$ if and only if $i \in s[j]$ for every $i, j \in V$. Given a partial route that a vertex j just after serving vertex i, we define the $MinIdleTime(i, j)$ as the minimum idle time between the end of service at vertex i and the beginning of service at vertex j. Formally, it is defined as follows:

$$MinIdleTime(i, j) = \max\{0, \min(t[j]) - \max(t[i]) - T_{i,j}\}.$$

5.2.1. Ride Time Forward Filtering. We denote by *End* the last vertex of the partial route of *vNext* (i.e., the partial path P_{last}). Now consider any pickup vertex P belonging to P_{first} such that its associated delivery vertex D is neither in P_{first} nor in P_{last}. Let j be a vertex belonging to the domain of $s[End]$ (see Figure 3). First, note that if j belongs to the partial path P_{first} or P_{last}, then there is no feasible solution having the arc (End, j). If j is the first vertex of the partial path of r^-, whenever the total time taken from r^+ to r^- using the arc (End, j) is greater than the ride time of the request r, we can eliminate j from the domain of $s[End]$. Because the triangular inequality holds, using any other arc of the type (End, x) with $x \in s[End]$ the ride time of the request r will be longer, and therefore we can produce a fail. Finally, if $j \neq Begin[r^-]$, we can eliminate j from $s[End]$ whenever the total ride time of r will exceed its limit. The pseudo-code of the filtering algorithm is presented in Algorithm 3.

Algorithm 3 (Ride time forward filtering algorithm).

```
Awakening condition: variable $s[vIndex]$ becomes
    fixed to $vNext$.
for each request $r$ such that $r^+ \in P_{first}$ and
    RequestBind[$r$] = $False$ do
for each vertex $j \in s[End]$ do
if $j \in P_{first}$ or $j \in P_{last}$ then
removeValue($s[End]$, $j$)
else
if $j$ = Begin[$r^-$] then
if (TravelTime[$r^+$] + $T_{End,j}$ + MinIdleTime($End$, $j$) +
    TravelTimeBack[$r^-$] > RideTime($r$)) then fail();
```

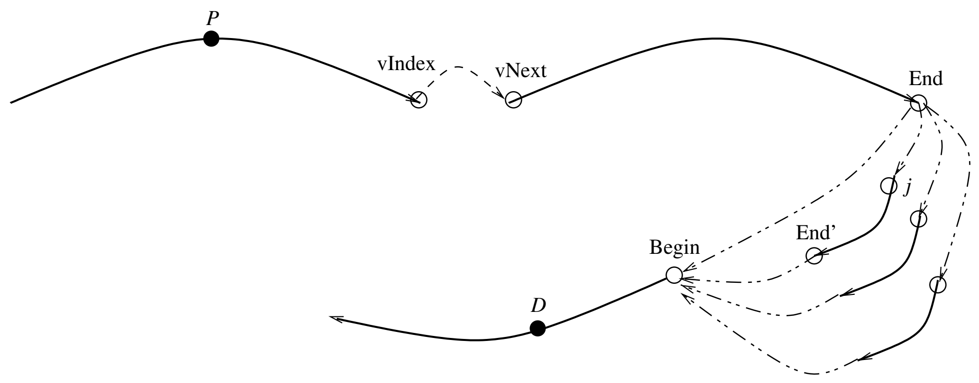

Figure 3 Notation for the Ride Time Forward Filtering Algorithm

```
else
if (TravelTime[$r^+$] + $T_{End,j}$ + MinIdleTime($End$, $j$) +
   TravelTime($j$) + $T_{Ends[j],Begin[r^-]}$ + MinIdleTime
   (Ends[$j$], Begin[$r^-$]) + TravelTimeBack[$r^-$] >
   RideTime($r$)) then
removeValue($s$[$End$], $j$);
end if
end if
end if
end for
end for
```

5.2.2. Ride Time Backward Filtering. The ride time backward filtering scheme resembles the ride time forward filtering. Here we denote by *Begin* the starting vertex of the partial route of *vIndex*. Consider any delivery vertex D belonging to P_{last} such that its pickup vertex, P, does not belong to P_{last} or to P_{first}. Also let j be a vertex belonging to *Prevs*[*Begin*] (see Figure 4). We first observe that if $j \in P_{first}$ or $j \in P_{last}$, then there is no feasible solution with the arc (j, *Begin*). Otherwise, we can produce a fail or remove j from *Prevs*[*Begin*], applying a reasoning similar to the one used at the forward filtering algorithm. The pseudo-code of the filtering algorithm is presented in Algorithm 4.

Algorithm 4 (Ride time backward filtering algorithm).

```
Awakening condition: variable $s$[$vIndex$] becomes
   fixed to $vNext$.
$Begin \leftarrow Begin[vIndex]$;
for each request $r$ such that $r^- \in P_{last}$ and
   RequestBind[$r$] = $False$ do
for each vertex $j \in Prevs[Begin]$ do
if $j \in P_{first}$ or $j \in P_{last}$ then
removeValue($Prevs$[$Begin$], $j$)
else
if $j$ = Ends[$r^+$] then
if (TravelTime[$r^+$] + $T_{j,Begin}$ + MinIdleTime($j$, $Begin$)
   + TravelTimeBack[$r^-$] > RideTime($r$)) then
fail( );
end if
else
if (TravelTime[$r^+$] + $T_{Ends[r^+],j}$ + MinIdleTime
   ($Ends$[($r^+$)], $Begin$[$j$]) + TravelTimeBack[$j$]
   +MinIdleTime($j$, $Begin$) + TravelTimeBack[$r^-$]
   > RideTime($r$)) then
removeValue(Prevs[$Begin$], $j$);
end if
end if
end if
end for
end for
```

6. Variable Selection and Value Selection Heuristics

We describe the variable selection and value selection heuristics used. A *variable selection heuristic* gives the order in which variables are fixed during the search process. A *value selection heuristic* gives the order in which the values of a variables are tried.

As a variable selection heuristic, we have applied a randomized version of the *sparse* heuristic proposed by Pesant et al. (1998). The heuristic can be described as follows.

Step 1: Consider the set H of successor variables s that are not fixed yet and let $\tilde{s}$ be the size be the smallest domain among the variables in H.

Step 2: Consider the subset $S = \{S_1, \ldots, S_k\}$ of H that has all the variables of H with domain size of $\tilde{s}$.

Step 3: For each value v that belongs to the domain of some variable $S_i \in S$, compute $v^{\#}$, the number of times v appears in the domain of some variable in S.

Step 4: Choose randomly a variable S_i from those that maximize $\sum_{v \in domain(S_i)} v^{\#}$.

We now describe the value selection heuristic we have chosen. The heuristic basically favors first any delivery vertex of a request such that its pickup vertex is already in the partial route, then any pickup vertex, and then any delivery or depot vertex. Let v be the successor variable to which we have to select a value from its domain set *Domain*(v). The algorithm can be described as follows.

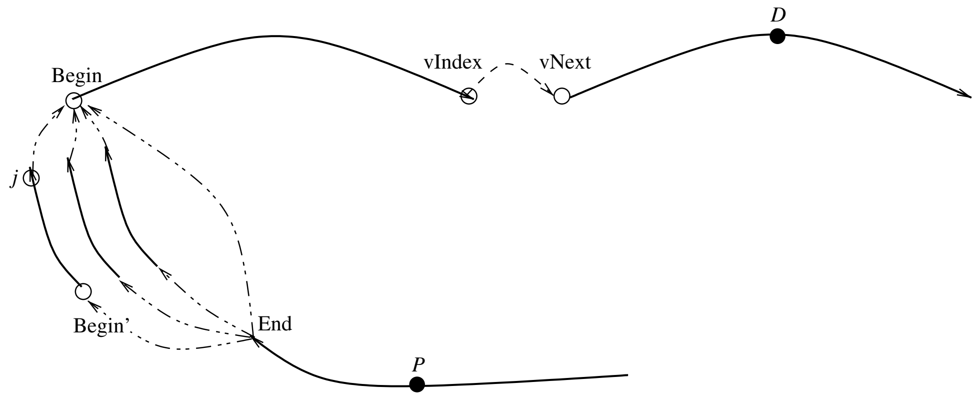

Figure 4 Notation for the Ride Time Backward Filtering Algorithm

Step 1: Let H^- be the set of deliveries whose pickup vertex is in the partial route of v. If $H^- \cap Domain(v) \neq \varnothing$, then select randomly a value from $H^- \cap Domain(v)$.

Step 2: Let H^+ be the set of pickups in $Domain(v)$. If $H^+ \neq \varnothing$ select randomly an element from H^+.

Step 3: Select randomly an element from $Domain(v)$.

More sophisticated methods for choosing the value based on the solution of the *assignment problem* over the unbounded vertices were implemented. Although they have produced good results, they were worse overall than the much simpler method stated above.

7. Reduction of the Search Space

We have reduced the search space by adding to the CSP model redundant constraints and by fixing some variables to break symmetries. We have used the technique proposed by Cordeau (2006) to identify pairs of requests that cannot be assigned to the same vehicle. Such pairs of requests are called *incompatible*. Two requests i and j are incompatible if none of the following six partial routes are feasible: (i^+, j^+, j^-, i^-), (i^+, j^+, i^-, j^-), (j^+, i^+, i^-, j^-), (j^+, i^+, j^-, i^-), (i^+, i^-, j^+, j^-), and (j^+, j^-, i^+, i^-). For every pair of requests, these six partial routes are analyzed to see if at least one of them respects the ride time constraint, the time window constraints, and the capacity constraint. We can then construct the following undirected graph $G' = (V', E')$, called the *incompatibility graph*. The vertex set V' consists of the requests R, and the edge $\{i, j\}$ belongs to E' if and only if i and j are incompatible requests.

It is worth noting that for checking the feasibility of each partial route, constraint propagation was applied to tighten the time windows of each vertex. As stated in Cordeau (2006), given a clique in G', it is clear that all the requests associated to the clique must be served with a different vehicle. It is therefore possible to find a maximum clique and then fix the vehicle that will serve each of the requests that belong to the clique. This gives us a way to break some symmetries of the problem.

Observe that the size of any clique gives a lower bound on the number of vehicles required. A better lower bound on the number of vehicles required can be obtained by computing the chromatic number of the graph G'. This is because given that it is impossible to color the graph G' with less than $\chi(G') = k$ colors, we would need at least k vehicles to serve all requests such that no two incompatible requests are served by the same vehicle. Once we compute the chromatic number of G', it is not possible to do the variable fixing in the same way as it was done after computing a clique. To see this, let c be a coloring of G' with $\chi(G')$ colors. It could happen that there exists a feasible solution of the DARP instance in which two requests which were given different colors are served by the same vehicle but no solution exists if both requests are served by different vehicles. It is nevertheless possible to fix variables to force any solution to have at least $\chi(G')$ nonempty vehicle routes. If the instance has less than $\chi(G')$ vehicles, we can prove it is infeasible. Because the chromatic number of a graph can be arbitrarily large with respect to the size of the maximum clique (Mycielski 1955), computing it can be useful to prove the infeasibility of some instances. This method could be particularly useful for determining the minimum number of vehicles required for a solution to exist on a given DARP instance by trying an increasing number of vehicles starting with the chromatic number of the incompatibility graph. The method could also be applied to other constrained vehicle routing problems.

8. Computational Results

We have conducted extensive computational tests to determine the efficiency of the filtering algorithms developed and the CP approach in general. We have implemented the CSP model for the DARP in C++ by using the ILOG Solver 6.0. We have also implemented the filtering algorithms proposed for each of the two relaxations presented. The program was run on a 2.5-GHz dual core AMD Opteron computer.

The instances were based on the set instances *a* and *b* used in Ropke, Cordeau, and Laporte (2007). In the instance set *a*, vertices are located in a 20×20 square, the distances are Euclidean and are measured in minutes, the time horizon is 12 hours, time windows have 15 minutes of length, and $Q = 3$. In the instance set *b*, is similar, except that the vehicle capacity (Q) is 6. We have only used the instances with at least 40 requests. The instance labels are of the form "*am-n*" or "*bm-n*." The letters *a* and *b* state whether the instance is from the set *a* or *b*, the number *m* corresponds to the number of vehicles, and the number *n* states the number of requests. More details of the instances can be found in Cordeau (2006).

The computational results are presented in §§8.1 and 8.2. First, in §8.1 we present experimental results to assess the utility of the space reduction techniques and the filtering algorithms. Finally, in §8.2, we compare the time taken to obtain a feasible solution with the CP algorithm and with the tabu search algorithm developed by Cordeau and Laporte (2003) under different situations.

In the case where neither a solution is found nor is infeasibility proved after five seconds, we restart the algorithm. The restart is repeated until three minutes of computing time are reached. The restart time sequence strategy (in seconds) begins with $s = 5, 5, 10, 5, 5, 10, 20, \ldots,$ which is the sequence proved to be optimal for Las Vegas algorithms by Luby, Sinclair, and Zuckerman (1993) when there is no knowledge about their running time distribution, using a unit time of five seconds. We have run the CP algorithm 10 times on each instance, and we give the average time taken to find a feasible solution. The label "inf. (t)" means that the CP algorithm has proved in t seconds that the instance is not feasible.

8.1. Impact of the Space Reduction and Filtering Algorithms

This first set of experiments was performed to assess the impact of the filtering algorithms and the space reduction techniques on the the time taken to obtain a feasible solution. In Table 1 we show for each tested instance of the set *a* and *b* the time taken in seconds to find a feasible solution when no space reduction and filtering algorithms are applied, when only space reduction is applied, and when space reduction and filtering algorithms are used. The maximum ride time was set to 30 minutes.

On average, the time taken to find a feasible solution using the space reduction techniques was reduced by about 25 percent when compared to the execution of the CP algorithm alone. In addition, when the space reduction and the filtering algorithms were applied, the time taken to find a feasible solution was reduced on average by about 80% compared to the execution of the CP alone. It is worth observing that there are four instances that could not be solved with the CP approach alone, but solutions were found when the filtering and space reduction methods were applied.

Table 1 **Comparison of the Time Needed in Seconds to Obtain a Feasible Solution with and Without the Space Reduction and Filtering Algorithms**

Instance	Basic	Space reduction	Space reduction and filtering
a4-40	1.3	0.3	0.5
a4-48	2.5	2.5	0.5
a5-40	2.2	0.5	0.3
a5-50	3.5	0.9	0.5
a5-60	9.8	9.2	1.0
a6-48	14.4	7.3	0.6
a6-60	46.1	9.5	5.6
a6-72	35.8	15.9	5.0
a7-56	14.8	—	1.8
a7-70	—	—	41.5
a7-84	72.9	44.1	3.4
a8-64	83.2	24.7	6.2
a8-80	171.2	25.1	8.5
a8-96	—	—	38.7
b4-40	0.4	0.4	0.3
b4-48	2.7	0.9	0.4
b5-40	1.7	1.1	0.4
b5-50	30.1	1.0	0.8
b5-60	37.2	10.2	3.1
b6-60	4.7	2.3	1.5
b6-72	24.3	9.3	2.4
b7-56	10.6	29.2	1.5
b7-70	47.5	154.6	18.4
b7-84	—	21.3	6.3
b8-64	140.4	74.6	1.9
b8-80	101.5	150.6	19.2
b8-96	—	—	34.8

Note. Runs exceeding 180 seconds are marked as "—."

8.2. Comparison with a Tabu Search Algorithm

We have conducted some experiments to compare the time taken to find a feasible solution for our constraint programming algorithm and a tabu search algorithm developed by Cordeau and Laporte (2003). We note that exact methods based on integer linear programming models, such as branch-and-cut, generally take much longer to find a feasible solution than does our CP algorithm. Solving the linear relaxation already takes a few minutes on instances with more than 50 requests. The results are shown in Table 2. Because the tabu procedure is randomized, we present the average result obtained from running each instance 10 times.

In the first two columns, we show the time (in seconds) taken to obtain a feasible solution with both methods on the instances in the original form, which is 30 minutes of ride time (RT) for set *a* and a RT of

Table 2 **Comparison Between a Tabu Search Algorithm and the CP Algorithm**

	Original		$RT = 30$		$RT = 22$		75 % of vehicles	
Instance	TABU (sec)	CP (sec)	TABU (sec)	CP (sec)	TABU (sec)	CP (sec)	TABU (sec)	CP (sec)
a4-40	0.8	0.5	0.8	0.5	0.5	0.3	1.7	0.3
a4-48	1.0	0.5	1.0	0.5	1.3	0.4	78.5	0.6
a5-40	0.3	0.3	0.3	0.3	0.3	0.3	1.3	0.3
a5-50	0.7	0.5	0.7	0.5	0.6	0.6	3.9	1.3
a5-60	1.4	1.0	1.4	1.0	1.5	0.9	—	24.5
a6-48	0.4	0.6	0.4	0.6	0.4	0.7	1.1	0.5
a6-60	1.0	5.6	1.0	5.6	—	inf. (1.1)	11.6	6.2
a6-72	1.9	5.0	1.9	5.0	—	inf. (1.7)	5.7	2.0
a7-56	0.5	1.8	0.5	1.8	—	inf. (0.6)	0.9	1.5
a7-70	1.7	41.5	1.7	41.5	1.4	7.6	3.2	5.1
a7-84	2.7	3.4	2.7	3.4	3.0	4.1	7.5	3.5
a8-64	0.8	6.2	0.8	6.2	—	inf. (1.9)	1.1	2.5
a8-80	1.5	8.5	1.5	8.5	1.8	6.0	3.0	3.6
a8-96	3.5	37.7	3.5	38.7	—	inf. (3.7)	8.1	5.3
b4-40	0.6	0.4	0.4	0.3	0.4	0.3	—	inf. (0.1)
b4-48	1.3	0.4	1.6	0.4	—	inf. (0.1)	—	inf. (0.1)
b5-40	0.4	0.3	0.4	0.4	—	inf. (0.1)	—	inf. (0.1)
b5-50	1.2	6.8	0.9	0.8	1.1	0.9	—	inf. (0.1)
b5-60	1.5	109.5	1.8	3.1	1.6	1.2	—	inf. (0.1)
b6-60	0.9	5.1	1.5	1.5	1.0	1.4	5.3	3.8
b6-72	2.1	27.3	2.3	2.4	2.4	2.4	9.7	25.6
b7-56	0.5	13.3	0.6	1.5	0.5	1.5	2.1	3.9
b7-70	1.4	5.6	1.6	18.4	1.3	2.7	12.6	78.7
b7-84	3.0	25.8	2.9	6.3	—	inf. (0.1)	—	—
b8-64	0.7	12.7	0.8	1.9	0.8	1.9	1.8	4.8
b8-80	1.9	23.9	1.8	19.2	—	inf. (0.5)	4.0	13.9
b8-96	4.1	149.1	3.9	34.8	3.9	56.1	8.2	—

45 minutes for set *b*. In the third and forth columns, we have modified the maximum RT of the instances to 30 minutes, while in the fifth and sixth columns the RT was set to 22 minutes, which is the minimum time needed to take each request from the pickup point to its delivery point. Finally, in the last two columns, we have modified the instances by reducing the number of available vehicles to 75 percent of the original fleet size.

Observe that in the tests with the nonmodified instances, i.e., the first two columns of results, the tabu search procedure generally finds a feasible solution faster than the CP algorithm. When the maximum ride time is set to 30 minutes, the tabu still performs better, but the difference between them has been reduced. When the maximum ride time is modified to 22 minutes, we can observe that the time taken for both methods is similar. Observe that in general, feasible solutions are found faster when the maximum ride time is reduced. A very plausible reason for this is that the constraint programming algorithm detects, using the ride time constraint, that certain partial solutions cannot be extended into a feasible solution. When the ride time constraint is tighter, the filtering is done earlier, and therefore the algorithm can escape an infeasible branch higher up in the search tree. This behavior is not generally is present in heuristic algorithms like the tabu search (see Table 2), because they usually find a feasible solution faster in less constrained instances.

Finally, when the fleet of available vehicles is reduced to 75 percent, the time taken for both methods is also similar on average. As exceptions, we can point out the last instance that could not be solved in three minutes by the CP approach and the instance *a*5-60 that was not solved by the tabu but solved by the CP algorithm in 24.5 seconds on average.

As a final remark, observe that among these computational tests, 14 out of the 15 instances for which neither the tabu search nor the CP algorithm was able to find a feasible solution, CP proved that they are actually infeasible. This is a key characteristic of the CP algorithm that differentiates it from tabu search or any other incomplete method.

9. Conclusions

We have modeled the DARP as a constraint satisfaction problem, and we have developed an exact constraint programming algorithm to determine whether or not an instance is feasible. The algorithm consists of filtering algorithms that detect whenever a partial solution cannot be extended into a complete solution for some DARP relaxations and solution space reduction techniques, embedded into a constraint programming engine.

The algorithm was tested over two sets of DARP instances found in the literature and on some instances obtained by applying some modifications to these. Results have shown that the algorithm was generally able to find a feasible solution within a few seconds, that the proposed filtering algorithm is effective, and that the algorithm generally performs better on more constrained instances. On these instances, the algorithm has a performance comparable to that of tabu search, while being generally able to prove infeasibility rapidly. This is an important feature in a real-time environment in which a decision whether to accept or reject a request has to be done quickly.

Acknowledgments
The authors thank Gilbert Laporte and Jean-François Cordeau for their advice and comments that helped to improve the quality of this work. The first author was supported by the Canada Research Chair in Distribution Management and the Canada Research Chair in Logistics and Transportation. This support is gratefully acknowledged.

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

Beldiceanu, N., M. Carlsson, J.-X. Rampon. 2005. Global constraint catalog. Research Report T2005:08, Swedish Institute of Computer Science. (Dynamic on-line version at http://www.emn.fr/x-info/sdemasse/gccat/).

Berbeglia, G., G. Pesant, L.-M. Rousseau. 2011. Feasibility of the pickup and delivery problem with fixed partial paths: A complexity analysis. *Transportation Sci.* Forthcoming.

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.

Caseau, Y., F. C. Laburthe. 1997. Solving small TSPs with constraints. L. Naish, ed. *Proc. 14th Internat. Conf. Logic Programming*, MIT Press, Cambridge, MA, 316–330.

Cordeau, J.-F. 2006. A branch-and-cut algorithm for the dial-a-ride problem. *Oper. Res.* **54** 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** 579–594.

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

De Backer, B., V. Furnon, P. Shaw. 2000. Vehicle routing problems using constraint programming and metaheuristics. *J. Heuristics* **6** 501–523.

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** 301–325.

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

Junker, U., S. Karish, N. Kohl, B. Vaaben, T. Fahle, M. Sellmann. 1999. A framework for constraint programming based column generation. J. Jaffar, ed. *Principles and Practice of Constraint Programming, CP 99*, Vol. 1713, *Lecture Notes in Computer Science*. Springer, Berlin/Heidelberg, 261–274.

Laporte, G., I. H. Osman. 1995. Routing problems: A bibliography. *Ann. Oper. Res.* **61** 227–262.

Luby, M., A. Sinclair, D. Zuckerman. 1993. Optimal speedup of Las Vegas algorithms. *Inform. Processing Lett.* **47** 173–180.

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** 193–208.

Mycielski, J. 1955. Sur le coloriage des graphes. *Colloquium Mathematicum* **3** 161–162.

Pesant, G., M. Gendreau. 1996. A view of local search in constraint programming. E. C. Freuder, ed. *Principles and Practice of Constraint Programming, CP 96*, Vol. 1118, *Lecture Notes in Computer Science*. Springer, Berlin/Heidelberg, 353–366.

Pesant, G., M. Gendreau, J.-Y. Potvin, J.-M. Rousseau. 1998. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Sci.* **32** 12–29.

Pesant, G., M. Gendreau, J.-Y. Potvin, J.-M. Rousseau. 1999. On the flexibility of constraint programming models: From single to multiple time windows for the traveling salesman problem. *Eur. J. Oper. Res.* **117** 253–263.

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

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

Rossi, F., P. van Beek, T. Walsh, eds. 2006. *The Handbook of Constraint Programming*. Elsevier, Amsterdam.

Rousseau, L.-M., M. Gendreau, G. Pesant. 2002. Using constraint-based operators to solve the vehicle routing problem with time windows. *J. Heuristics* **8** 43–58.

Rousseau, L.-M., M. Gendreau, G. Pesant, F. Focacci. 2004. Solving VRPTWs with constraint programming based column generation. *Ann. Oper. Res.* **130** 199–216.

Savelsbergh, M. W. P. 1985. Local search in routing problems with time windows. *Ann. Oper. Res.* **4** 285–305.

Schrage, L., K. R. Baker. 1978. Dynamic programming solution of sequencing problems with precedence constraints. *Oper. Res.* **26** 444–449.

Shaw, P. 1998. Using constraint programming and local search methods to solve vehicle routing problems. G. Goos, J. Hartmanis, J. van Leeuwen, eds. *Principles and Practice of Constraint Programming CP 98*, Vol. 1520, *Lecture Notes in Computer Science*. Springer, Berlin/Heidelberg, 417–431.

Solomon, M. M. 1987. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Oper. Res.* **35** 254–265.

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

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.

Velasco-Rodríguez, N. 2006. *Problémes de Collectes et Livraisons: Application au Transport de Personnel Entre Plates-Formes Pétroliéres*. Université de Nantes, France.