Annals of Operations Research 130, 199–216, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Solving VRPTWs with Constraint Programming Based Column Generation

LOUIS-MARTIN ROUSSEAU, MICHEL GENDREAU and GILLES PESANT
{louism,michelg,pesant}@crt.umontreal.ca
Centre de recherche sur les transports, Université de Montréal, CP 6128 Succ Centre-Ville, Montréal, Canada H3C 3J7

FILIPPO FOCACCI ffocacci@ilog.fr
ILOG SA 9, rue de Verdun, BP 85, 94253 Gentilly Cedex, France

Abstract. Constraint programming based column generation is a hybrid optimization framework recently proposed (Junker et al., 1999) that uses constraint programming to solve column generation subproblems. In the past, this framework has been used to solve scheduling problems where the associated graph is naturally acyclic and has done so very efficiently. This paper attempts to solve problems whose graph is cyclic by nature, such as routing problems, by solving the elementary shortest path problem with constraint programming. We also introduce new redundant constraints which can be useful in the general framework. The experimental results are comparable to those of the similar method in the literature (Desrochers, Desrosiers, and Solomon, 1992) but the proposed method yields a much more flexible approach.

Keywords: column generation, constraint programming, hybrid method, vehicle routing with time windows, optimization constraints

Introduction

Vehicle Routing Problems (VRP) are widely present in today's industries, ranging from distribution problems to fleet management. They account for a significant portion of the operational cost of many companies. The VRP can be described as follows: given a set of customers C, a set of vehicles V, and a depot d, find a set of routes of minimal length, starting and ending at d, such that each customer in C is visited by exactly one vehicle in V. Each customer having a specific demand, there are usually capacity constraints on the load that can be carried by a vehicle. In addition, there is a maximum amount of time that can be spent on the road. The time window variant of the problem (VRPTW) imposes the additional constraint that each customer c must be visited after time a_c and before time b_c. One can wait in case of early arrival, but late arrival is not permitted.

Column generation was introduced by Dantzig and Wolfe (1960) to solve linear programs with decomposable structures, it has been applied to many problems with success and has become a leading optimization technique to solve Crew Scheduling Problems (Desrosiers et al., 1995). In the first application to the field of Vehicle Routing Problems with Time Windows, presented by Desrochers, Desrosiers, and Solomon

(1992), the basic idea was to decompose the problem into sets of customers visited by the same vehicle (routes) and to select the optimal set of routes between all possible ones. Letting r be a feasible route in the original graph (which contains N customers); R be the set of all possible routes r, c_r be the cost of visiting all the customers in r; $A = (a_{ir})$ be a Boolean matrix expressing the presence of a particular customer (denoted by index $i \in [1..N]$) in route r; and x_r a Boolean variable specifying whether the route r is chosen ($x_r = 1$) or not ($x_r = 0$), the Set Partitioning Problem is defined as (S):

$$\begin{aligned} \min \; & \sum_{r \in R} c_r x_r \\ \text{s.t.} \; & \sum_{r \in R} a_{ir} x_r = 1 \quad \forall i \in [1..N], \\ & x \in \{0, 1\}^N. \end{aligned}$$

This formulation, however, poses some problems. Firstly, since it is impractical to construct and to store the set R because of its very large size, it is usual to work with a partial set R' that is enriched iteratively by solving a subproblem. Secondly, the Set Partitioning formulation is difficult to solve when R' is small and it allows negative dual values which can be problematic for the subproblem (a negative dual means a negative marginal cost to service a node). That is why, in general, the following relaxed Set Covering formulation is used instead as a Master Problem (M):

$$\begin{aligned} \min \; & \sum_{r \in R'} c_r x_r \\ \text{s.t.} \; & \sum_{r \in R'} a_{ir} x_r \geqslant 1 \quad \forall i \in [1..N], \\ & x \in [0, 1]^N. \end{aligned}$$

To enrich R', it is necessary to find new routes which offer a better way to visit the customers they contain, that is, routes which present a negative reduced cost. The reduced cost of a route is calculated by replacing the cost of an arc (the distances between two customers) d_{ij} by the reduced cost of that arc $c_{ij} = d_{ij} - \lambda_i$ where λ_i is the dual value associated with customer i. The dual value associated with a customer can be interpreted as the marginal cost of visiting that customer in the current optimal solution (given for R'). The objective of the subproblem is then the identification of a negative reduced cost path, that is, a path for which the sum of the travelled distance is inferior to the sum of the marginal costs (dual values). Such a path represents a novel and better way to visit the customers it serves.

The optimal solution of (M) has been identified when there exists no more negative reduced cost path. This solution can, however, be fractional, since (M) is a relaxation of (S), and thus does not represent the optimal solution to (S) but rather a lower bound on it. If this is the case, it is necessary to start a branching scheme in order to identify an integer solution.

Most column generation methods make use of dynamic programming to solve the shortest path subproblem where the elementary constraint (i.e., the constraint that the path does not go through the same node more than once) has been relaxed (Desrosiers, Solomon, and Soumis, 1993). This method is very efficient. But since the problem allows negative weight on the arcs, the path produced may contain cycles (since negative cost cycle decrease the objective function). However, applications of column generation in Crew Scheduling generally present an acyclic subproblem graph (one dimension of the graph being *time*) which eliminates this problem. Since routing problems are cyclic by nature, most of the methods that address the cyclic cases do so by first rendering the associated graph acyclic. Unfortunately, this transformation requires the different resources (time windows, capacity, etc.) to be discrete and the size of the resulting graph is usually quite impressive (pseudo-polynomial in the resources width).

This paper considers a routing application domain and thus concentrates on Cyclic Resource Constrained Shortest Path Problems. These problems are also referred to as Resource Constrained Profitable Tour Problems (PTP) in the literature. The objective is to construct a tour that minimizes the sum of the distance travelled and maximizes the total amount of prize (here, dual values) collected. These objectives are in conflict since more prize collected implies more distance travelled. The combined objective is thus to minimize the total length of the routes minus the sum of all the dual values collected.

There have been few attempts to combine column generation and constraint programming. Junker et al. (2000) presented a framework that uses constraint programming search goals to guide the column generation process. Junker et al. (1999) have proposed a framework they call *constraint programming based column generation* which uses CP to solve constrained shortest path subproblems in column generation. Fahle and Sellmann (2002) later enriched the CP based column generation framework with a *Knapsack* constraint for problems which present knapsack subproblems. The general framework, which is detailed in the next section, however, requires that the underlying graph be acyclic.

The purpose of this paper is to show that constraint programming methods can identify elementary negative reduced cost paths by working on the smaller original cyclic graph. The use of CP also allows the addition of any form of constraints on the original problem (which is not the case with the dynamic programming approach). It is thus possible to deal with multiple time windows, precedence constraints amongst visits or any logical implication satisfying special customer demands. In fact, the case of multiple time windows TSP was addressed successfully in (Pesant, 1999)

Figure 1 gives an overview of the overall approach that is describe in this paper. The following sections presents the model chosen to describe the Profitable Tour Problem, introduces some redundant constraints and discusses the transformation of a PTP into an Asymmetric Travelling Salesman Problem (ATSP) in order to use known lower bounds. The results section evaluates the impact of the different components and compares the proposed method with two other exact algorithms that solve the same problem.

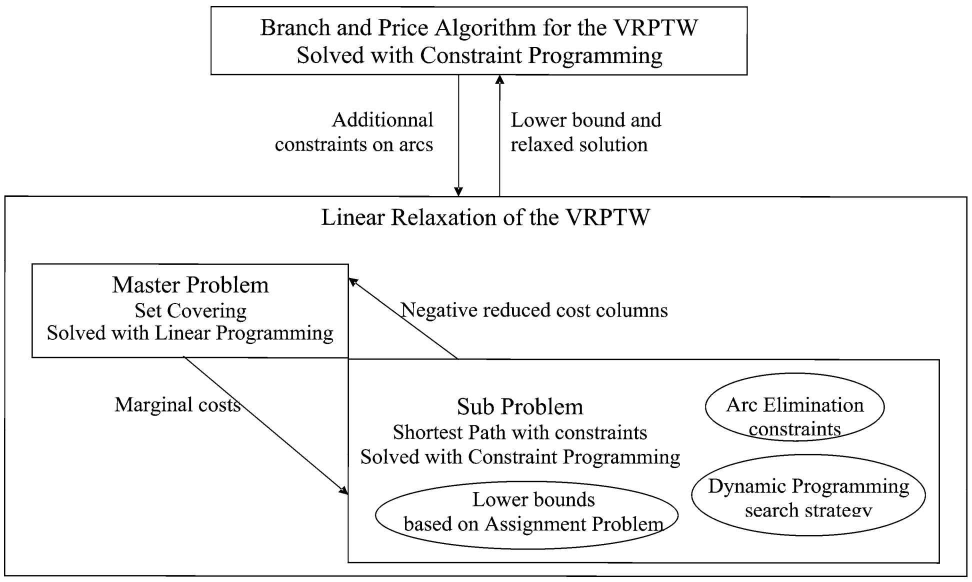

Branch and Price Algorithm for the VRPTW
Solved with Constraint Programming
Additionnal constraints on arcs
Lower bound and relaxed solution
Linear Relaxation of the VRPTW
Master Problem
Set Covering
Solved with Linear Programming
Negative reduced cost columns
Marginal costs
Sub Problem
Shortest Path with constraints
Solved with Constraint Programming
Arc Elimination constraints
Dynamic Programming search strategy
Lower bounds based on Assignment Problem

Figure 1. Overall method.

1. CP based column generation

The original motivation to use constraint programming based column generation (Junker et al., 1999) to solve airline crew assignment problems was that some problems were too complex to be modelled easily by pure Operational Research (OR) methods. Thus, the use of constraint programming to solve the subproblem in a column generation approach provided both the decomposition power of column generation and the modelling flexibility of constraint programming.

To model the constrained shortest path problem, the authors of (Junker et al., 1999) propose to use a single set variable Y (that is a variable which final value is a set) which contains the nodes to be included in the negative reduced cost path. Since the problem addressed in that framework is by nature acyclic (the underlying network is time directed), it is easy to compute in polynomial time the shortest path covering nodes in Y. A special constraint is also introduced to improve pruning and efficiency of the overall method. This constraint, which ensures that the nodes in Y are part of a feasible path, also enforces bound consistency by solving a shortest path problem on both the required and possible sets of Y. An incremental implementation of the Shortest Path algorithm ensures that the filtering is done efficiently.

2. Subproblem model

Since the problem considered, here, is cyclic, simple set variables cannot be used to record solutions (as proposed in (Junker et al., 1999)) because the construction of a complete solution from the set of included visits would require solving a TSP. Thus there is the need for a new model. Let $N = [0..n]$ be the set of all customers. The depot is copied into node $n+1$, so that a path starts and ends at a different depot. Let $N' = [1..n+1]$ be the set of all nodes except the initial depot.

2.1. Parameters

d_{ij}	distance from node i to node j.
t_{ij}	travel time from node i to node j.
a_i, b_i	bounds of node i's time window.
l_i	load to take at node i.
λ_i	dual value associated with node i.
C	capacity of the vehicle.
$c_{ij} = d_{ij} - \lambda_i$	reduced cost to go from node i to node j.

2.2. Variables

$S_i \in N' \; \forall i \in N$	direct successor of node i.
$T_i \in [a_i, b_i] \; \forall i \in N \cup \{n+1\}$	time of visit of node i.
$L_i \in [0, C] \; \forall i \in N \cup \{n+1\}$	truck load before visit of node i.

2.3. Objective

$$\text{Minimize} \sum_{i \in N} c_{iS_i} \quad \text{total travelled distance.}$$

2.4. Constraints

$$\mathit{AllDifferent}(S) \quad \text{conservation of flow;} \tag{1}$$

$$\mathit{NoSubTour}(S) \quad \text{SubTour elimination constraint;} \tag{2}$$

$$T_i + t_{iS_i} \leqslant T_{S_i} \; \forall i \in N \quad \text{time window constraints;} \tag{3}$$

$$L_i + l_i = L_{S_i} \; \forall i \in N \quad \text{capacity constraints.} \tag{4}$$

The S variable identify the *direct successor* of each node of the graph, but for the sake of brevity we will refer to these variables only as *successor* variables. The nodes which are left out of the chosen path are represented with self loops and thus have their S_i value fixed to the value i. Constraint sets (3) and (4) enforce the respect of the capacity and time windows by propagating the information about the time and load when a new arc becomes known.

The constraint *AllDifferent* (1) is used to express conservation of flow in the network. The nature of the decision variable already enforces that each node has exactly one outgoing arc, but we also need to make sure it also has exactly one ingoing arc. To do so it is necessary to insure that no two nodes have the same successor, which is the

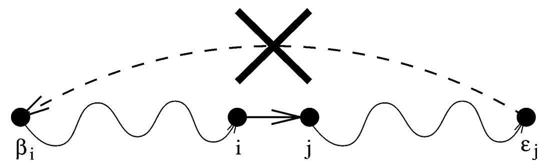


Figure 2. NoSubTour constraint.

role of the *AllDifferent* constraint. This property is obtained by solving a matching problem in a bipartite graph and by maintaining arc consistency in an incremental fashion as described in (Régin, 1994).

The constraint *NoSubTour* (2), illustrated in figure 2, is taken from the work of Pesant et al. (1998). For each chain of customers, we store the name of the first and last visit. When two chains are joined together (when a variable is fixed and a new arc is introduced), we take two actions. First, we update the information concerning the first and last visits of the new (larger) chain, and then, we remove the value of the first customer from the domain of the Successor variable of the last customer.

3. Subproblem additional constraints

In order to improve solution time, we introduce redundant constraints, which do not modify the solution set but allow improved pruning and filtering. The first redundant constraints are taken from previous work made on the Traveling Salesman Problem with Time Windows (TSPTW) while the second class of constraint are new and introduced in this paper.

4. TSPTW constraints

The constraints introduced in (Pesant et al., 1998), which perform filtering based on time window narrowing, are included in the present method. These constraints maintain for each node (say i) the latest possible departure time and the node (say j) associated with this time. When the domain of S_i is modified the constraint first verifies that j is still in the domain of S_i and if so performs no filtering. This implementation allows the computation of time feasibility to be very efficient since it prevents redundant checking of known feasible solution. For this project, we have implemented similar constraints dealing with the capacity dimension of the VRPTW. For the sake of brevity, we have chosen not to detailed these constraint here. The detailed explanation, including all propagation algorithms, can be found in the original paper (Pesant et al., 1998).

4.1. Arc elimination constraints

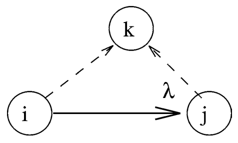
We introduce a new family of redundant constraints that can reduce the number of explored nodes of the search tree by reducing the number of arcs of the subproblem graph. The idea is to eliminate arcs which we know will not be present in the PTP optimal solution. Such a practice is known as cost-based filtering or optimization constraints (introduced in (Focacci, Lodi, and Milano, 1999)) since it filters out feasible solutions but not optimal ones.

The idea to eliminate arcs that cannot be present in the optimal solution has already been used by Mingozzi et al. (1999). However, the two routines they proposed cannot be applied to our method. The first one, which heavily relies on the dynamic programming approach used to identify negative reduced paths, is too expensive in terms of computation (pseudo-polynomial on the resources width) and the second is trivially enforced by constraint propagation.

We propose two arc eliminating constraints that can significantly reduce the size of the original graph based on the following idea: if the dual value associated with a customer is not sufficiently large, it may then not be worth the trip to visit this customer. Again, these constraints are valid only if the triangular inequality holds for the resources. Otherwise, the visit of an intermediate customer could yield savings in some resources and thus allow the visit of extra customers. Since this inequality does not hold when $j = i$ (in the following equations) the constraint was not defined over self-loop.

4.1.1. Arc elimination of type 1

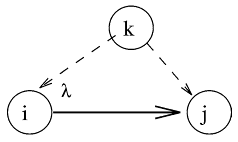
The *arc elimination constraint of type 1* is defined as follows: given an arc (i, j), if for all other customers k that are elements of the domain of the successor variable of j (S_j), it is always cheaper to go directly from i to k (d_{ik}) than to travel through j $(d_{ij} + d_{jk} - \lambda_j)$, then the arc (i, j) can be eliminated from the subproblem graph since it will never be part of an optimal solution:



$$\forall i \in N,\ \forall j \in S_i\colon\ j \neq i \text{ impose that}$$
$$\big(\forall k \in S_j\colon\ k \neq i \neq j\ (d_{ij} + d_{jk} - \lambda_j > d_{ik})\big) \ \Rightarrow\ S_i \neq j.$$

4.1.2. Arc elimination of type 2

The *arc elimination constraint of type 2* is defined as follows: given an arc (i, j), if for all other customers k whose successor variable (S_k) include i, it is always cheaper to go directly from k to j (d_{kj}) than to travel through i $(d_{ki} + d_{ij} - \lambda_i)$, then the arc (i, j) can be eliminated from the subproblem graph since it will never be part of an optimal solution:



$$\forall i \in N,\ \forall j \in S_i\colon\ j \neq i \text{ impose that}$$
$$\big(\forall k \in P_i\colon\ k \neq i \neq j\ (d_{ki} + d_{ij} - \lambda_i > d_{kj})\big) \Rightarrow\ S_i \neq j.$$

These constraints can be applied before the search to identify a negative reduced cost path is undertaken, and thus could be used in conjunction with any method addressing the PTP (even the dynamic programming approach which solves a relaxation of the PTP). However, since we are in the constraint programming paradigm we can obtain further pruning by applying these constraints during search.

To run efficiently these constraints must be implemented incrementally. We detail here the type 1 constraints while the other types have similar implementations. For each arc (i, j) the value of k, a successor of j, for which $d_{ij} + d_{jk} - \lambda_j \leqslant d_{ik}$ is kept. When the domain of S_j is modified, the constraint first validates two conditions before performing any filtering. Firstly, the arc (i, j) must still be a possible arc in the solution, that is value j must still be present in the domain of S_i. Secondly, if the value k is still in the domain of S_j then no filtering can occur and the filtering algorithm returns. Only when arc (i, j) is still present and k is no longer a possible successor of j does the constraint look for a new value of k; when none is found the arc is removed.

5. Search strategies for the subproblem

To construct the solution we need to define variable and value selection heuristics. Since the model used is very similar to the TSPTW model presented in (Pesant et al., 1998), the selection strategies presented in that paper could be used to guide the search in the present framework. This has been tried with limited success but, since most successful applications have been achieved through the use of dynamic programming, another search strategy is proposed.

The selection heuristics are based on what has been called Constraint Programming Based Dynamic Programming (Focacci and Milano, 2001) and it uses the notion of conditional constrained variables.

5.1. Conditional variables

Conditional variables have been introduced in (Focacci and Milano, 2001) to model dynamic programming methods (e.g., DP state space relaxation) in Constraint Programming. They are used to define variables that may or may not have sense in the definition of a problem.

A conditional variable is very similar to a traditional variable, except that it is associated with a definition constraint. When this constraint is satisfied, the conditional variable behaves just as a regular one. However, when its definition constraint becomes violated the conditional variable is emptied and all constraints that involved it are trivially satisfied. A more formal definition follows.

5.1.1. Definition

A *conditional variable* V^{c} is a variable depending on a constraint; it is defined by the pair $(D^{\mathrm{c}}, C^{\mathrm{c}})$ where D^{c} is a domain of possible values, and C^{c} is a constraint. The domain of such a variable must contain at least one value if and only if the definition constraint is respected and is emptied, otherwise. The following describes this behavior:

$$C^{\mathrm{c}} \iff D^{\mathrm{c}} \neq \emptyset.$$

A conditional variable $V^{\mathrm{c}} = (D^{\mathrm{c}}, C^{\mathrm{c}})$ is said to be *true* if its definition constraint C^{c} is true and false, otherwise. The constraint C^{c} of V^{c} is identified by $C^{\mathrm{c}}(V^{\mathrm{c}})$.

5.1.2. Operators

Several operators can be defined to involve conditional variables. Let us first consider the equality constraint defined between a variable and a conditional variable: let $V^{\mathrm{c}} = (D^{\mathrm{c}}, C^{\mathrm{c}})$ be a conditional variable, and V be a traditional constrained variable associated with domain D. The constraint $V^{\mathrm{c}} = V$ holds if C^{c} is false or if C^{c} is true and the two variables take on the same value. Again enforcing the fact that constraints treat conditional variables as regular variables when their definition constraint is true and are considered satisfied otherwise. The equality constraint could thus propagate as follows:

$$\begin{aligned} i \notin D &\implies V^{\mathrm{c}} \neq i, \quad \forall i \in D^{\mathrm{c}}, \\ C^{\mathrm{c}} &\implies \big(i \notin D^{\mathrm{c}} \Rightarrow V \neq i, \ \forall i \in D\big). \end{aligned}$$

Similarly, this equality constraint can be applied to two conditional variables. Let $V_1^{\mathrm{c}} = (D_1^{\mathrm{c}}, C_1^{\mathrm{c}})$, and $V_2^{\mathrm{c}} = (D_2^{\mathrm{c}}, C_2^{\mathrm{c}})$ be two such variables. The constraint $V_1^{\mathrm{c}} = V_2^{\mathrm{c}}$ holds if $C_1^{\mathrm{c}} \wedge C_2^{\mathrm{c}}$ is false or if $C_1^{\mathrm{c}} \wedge C_2^{\mathrm{c}}$ is true and the two variables assume the same value.

Conditional variables can also be combined via arithmetic operators: for instance, two conditional variables $V_1^{\mathrm{c}} = (D_1^{\mathrm{c}}, C_1^{\mathrm{c}})$, and $V_2^{\mathrm{c}} = (D_2^{\mathrm{c}}, C_2^{\mathrm{c}})$ can be combined as $V_3^{\mathrm{c}} = V_1^{\mathrm{c}} + V_2^{\mathrm{c}}$, where V_3^{c} is defined by $V_3^{\mathrm{c}} = (D_3^{\mathrm{c}}, C_1^{\mathrm{c}} \wedge C_2^{\mathrm{c}})$ and where D_3^{c} is defined by the usual propagation rule of the sum operator. The complete propagation mechanisms are described in detail in (Focacci and Milano, 2001).

The concept of conditional variables has also been used in constraint programming to implement exclusive disjunctions, which means only one out of a set of conditional variables should be *true*. A global constraint can be defined as follows: $xunion(V^{\mathrm{c}}_{[1..k]}, y, x)$, where y and x are regular variables and $V^{\mathrm{c}}_{[1..k]}$ is an array of k conditional variables. This constraint thus ensures that exactly one out of the k conditional variables $V^{\mathrm{c}}_{[1..k]}$ will be *true*. Moreover, it imposes that this conditional variable will equal to the normal variable y and defines the variable x as the index at which is located in the array (i.e., $x \in [1..k]$). Formally, the constraint $xunion(V^{\mathrm{c}}_{[1..k]}, y, x)$

holds iff:

$XOR_{i=1..k}C^c(V_i^c)$	only one of the definition constraints is respected.
$x = i \mid C^c(V_i^c)$	x represents the index of *true* conditional variable.
$y = V_x^c$	y takes the value of the *true* conditional variable.

The following subsection will illustrate well how these variable can be useful.

5.2. *Subproblem model using conditional variables*

In addition to the model described in section 2, new domain variables are introduced to implement a state space relaxation graph of the original problem. The first variables P identify the *position* at which each node is visited. P_i thus indicates the position of node i in the solution and P_i is set to 0 when node i is not visited in the optimal path (except for the source node, which is said to be visited at position 0).

For each node i, new conditional variables $S_{ip}^c = (N', P_i = p)\ \forall p \in N$ are introduced and represent the node directly succeeding node i in the optimal path if i is visited at position p. Conditional variables are necessary here because S_{ip}^c does not have any sense if i is not visited at position p. Similar variables (T_{ip}^c, L_{ip}^c) are also introduced with respect to the time and capacity dimension of the problem. The original variables (S_i, T_i, L_i) are equal to the exclusive disjunction of the n conditional variables $S_{ip}^c, T_{ip}^c, L_{ip}^c$ since node i can only be visited at one position. These new variables and constraints, once introduced in the model presented in section 2, allow the search to proceed in a dynamic programming fashion.

5.2.1. *New variables*

$P_i \in N$	$\forall i \in N.$
$S_{ip}^c \in (N', P_i = p)$	$\forall i, p \in N.$
$T_{ip}^c \in ([a_i, b_i], P_i = p)$	$\forall i, p \in N.$
$L_{ip}^c \in ([0, C], P_i = p)$	$\forall i, p \in N.$

5.2.2. *New constraints*

$P_0 = 0.$	
$S_{i0}^c = i$	$\forall i \in N \setminus \{0\}.$
$S_{ip}^c \neq i$	$\forall i \in N, p \in N \setminus \{0\}.$
$xunion(S_{i[0..n]}^c, S_i, P_i)$	$\forall i \in N.$
$xunion(T_{i[0..n]}^c, T_i, P_i)$	$\forall i \in N.$
$xunion(L_{i[0..n]}^c, L_i, P_i)$	$\forall i \in N.$
$(P_j = p) \Rightarrow \exists i \neq j \mid S_{i(p-1)}^c = j$	$\forall i, p \in N.$
$T_{ip}^c + t_{iS_{ip}^c} \leqslant T_{S_{ip}^c p+1}^c$	$\forall i \in N.$
$L_{ip}^c + l_i = L_{S_{ip}^c p+1}^c$	$\forall i \in N.$

Once these constraints are added to original model, it is possible to propagate bound information on all the new variables (those indexed by the position value) in a dynamic programming fashion (as described in (Focacci and Milano, 2001)). This information is then used to guide the branching process on the original (S) variables. The variable selection strategy attempts to construct the shortest path from the source node to the sink node, which means it always selects the successor variable of the last node (say i) that has been added to the path. The value selection heuristic is simply to fix S_i to the most promising[1] value of the domain of S_{ip} where p is the position at which i was inserted.

Since finding the optimal negative reduced cost path is not important in a column generation framework, the problem is treated as a Constraint Satisfaction Problem (CSP) and only the first k solutions[2] are included into R'.

6. Lower bounds for the subproblem

In order to prune the search tree efficiently we must be able to compute lower bounds at each node. Unfortunately, even if the literature is prolific in terms of lower bounds for the TSP, none explicitly exists for the PTP. It is fairly simple, as shown in figure 3, to transform a PTP into an asymmetric TSP (see (Dell'Amico, Maffioli, and Varbrand, 1995)) by adding N nodes and $2N$ arcs (note that in this new graph $c_{ij} = d_{ij}$). This extra portion of the graph constitutes a dummy path that allows the visit of nodes left unvisited by the PTP solution. The cost of visiting a node through this dummy path is set to the cost of its associated dual value. The objective value of the resulting ATSP optimal solution will be superior to the value of the optimal solution to the PTP by a constant that equals the sum of all dual values ($\sum_{i\in\{1,\dots,N\}} \lambda_i$).

Well-known ATSP lower bounds can then be applied to the transformed graph once the resource constraints have been relaxed. An optimal solution to the Assignment Problem (AP) is a lower bound for the ATSP since it is obtained by relaxing the *NoSubTour* constraint. Efficient primal-dual methods (like the Hungarian Algorithm (Carpaneto, Martello, and Toth, 1994)) can be used to solve the AP and provide reduced costs, with which some further domain filtering can be achieved. As described in (Focacci, Lodi, and Milano, 1999), the reduced cost c' can be interpreted as the additional cost to incur if an unused arc is introduced in the solution. Arcs (i, j) whose reduced cost added to

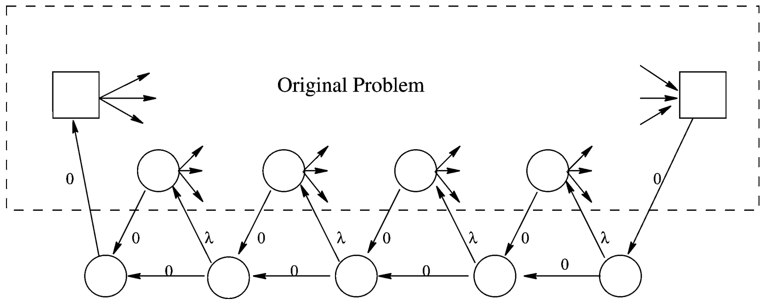


Figure 3. Transformation of a PTP to an ATSP.

lower bound exceed the current upper bound ($LB + c'_{ij} > UB$) can thus be eliminated dynamically during search.

7. Branch and price to solve the problem

The optimal solution to the master problem is obtained once we have proven that no negative reduced cost path exits. Unfortunately, this solution is not always integral and a branching scheme is thus needed to close the integrality gap. It is not useful to branch on the variables of the master problem because these variables cannot be forced to take the value 0. Even if we fixed x_r to 0 we could not efficiently prevent the CP algorithm from generating again the same route r and adding it to R'.

7.1. Branching strategies

We therefore choose as branching variables $\{B_i \in N \cup F \mid i \in N \cup I\}$, a set of successor variables similar to those used to describe the subproblem. In the following branching strategy, let f^r_{ij} be a Boolean value indicating whether j is the successor of i in route r and f_{ij} be the flow traversing arc (i, j).

1. *Node 0*. Iterate between the master problem and the subproblem until there exists no more negative reduced cost paths.
2. *Upper bounding*. If the current solution to the LP is an integer and its value is better than the best solution found, then update the upper bound, store the current solution and backtrack.
3. *Branching*. Once the optimal solution of the master problem has been found, identify the most fractional variable as the next branching (B) variable to be fixed. To do so, first calculate the flow that traverses each arc $f_{ij} = \sum_{r \in R'} f^r_{ij} x_r$. Then count for each customer i the number of positive flow outgoing arcs $o_i = \sum_{j \in [1..N]} (f_{ij} > 0)$. Finally, select for branching the B_i variable which is associated with the maximum value of o_i and branch on the value j which maximizes f_{ij}.
4. *LP probing*. In case of a tie in the value selection criteria, when for instance a variable has two outgoing arcs of flow 0.5, a tie breaking strategy must be employed. Since, in the present case, the master problem can be solved very efficiently (in about 0.01 seconds by LP), it can be used to estimate the impact of branching decisions. The results of selecting each value is thus temporarily imposed on the master problem which is then solved. The branching strategy then selects the value which generated the lowest LP value.
5. *Filtering*. Once a branching decision has been made, it is important to enforce it throughout the rest of the algorithm. The first measure to take is to prevent the selection of any columns that violate previously taken decisions. To do so, fix the following variable in the Set Covering Model:

$$x_r = 0 \quad \forall r \in R', i \in N, j \notin B_i\colon\ f^r_{ij} = 1.$$

It is also crucial to ensure that branching decisions are taken into account at the subproblem level. Therefore add the following constraints to the subproblem model:

$$j \notin B_i \;\Rightarrow\; j \notin S_i \quad \forall i, j \in [1..N].$$

6. *Lower bounding*. Just as in step one, iterate between the master problem and the subproblem until there exist no more negative reduced cost paths, while taking into consideration the new filtering constraints. If the lower bound obtained is higher than the upper bound then backtrack and cancel the previously taken branching decisions. Otherwise, go back to step two.

7.2. Initial bounds and columns

In order to accelerate the optimization process, we used known heuristic methods to rapidly obtain an upper bound on the original VRPTW. These heuristics are classic construction heuristics like *insertion*, *savings*, and *sweep* methods and are provided in the ILOG Dispatcher (ILOG S.A., 2002b) library used for this project. We then proceeded with a descent algorithm using the *2-opt, or-opt, cross, exchange* and *relocate* operators to obtain a good solution rapidly. We did not modify the construction heuristics in any way and we used as a descent algorithm the code provided in a Dispatcher example file (`vrp.cpp`). We also took advantage of this preliminary phase to improve the quality of the initial column set (R'). All routes identified during the descent phase are stored in R' and the best solution found is kept as an upper bound.

8. Experimental results

This section first evaluates the impact of the redundant constraints proposed for the negative cost shortest path problem and then reports results on well-known vehicle routing benchmarks. The results obtained are compared with two other approaches: a traditional dynamic programming based column generation framework and a pure constraint programming model.

8.1. Arc Elimination constraints

It is interesting to study the impact of the Arc Elimination constraints since they can be used to accelerate most constrained Profitable Tour Problem, even when they are solved with dynamic programming. Its behavior is compared on three different PTPs taken from the three problem classes introduced by Solomon. The chosen problems all have a good mixture of large and tight time windows. In table 1 we report the total number of backtracks and the CPU time needed to prove the optimality of the best found negative reduced cost path.

Table 1 thus reports the performance of the Arc Elimination constraints with respect to the level of propagation used. The constraint, when used, is either applied

Table 1
Arc elimination constraints propagation strategies: number of backtracks and time to solve one PTP.

Problem	Never		Preprocessing		During search	
class	Backtracks	Time	Backtracks	Time	Backtracks	Time
C102	6303	68.99	511	3.65	163	1.82
R102	833	10.54	490	4.83	100	1.81
RC102	5813	79.11	168	1.43	78	0.98

only once before the search starts reducing the domains (preprocessing) or incrementally when the domain of one of the involved variables is modified. Applying the constraints only once before the search starts already reduces significantly both the number of backtracks and the CPU time needed to find the optimal path. However, constraint programming methods can further utilize these constraints by incrementally maintaining consistency during search.

8.2. Benchmarking problems

We have tested the proposed method on the well-known Solomon problems. The geographical data are randomly generated in problem sets R1, clustered in problem sets C1, and a mix of random and clustered structures in problem sets RC1. The customer coordinates are identical for all problems within one type (i.e., R, C and RC). The problems differ with respect to the width of the time windows. Some have very tight time windows, while others have time windows which are hardly constraining. Each problem contains 100 customers but smaller problems are generated by considering only the 25 or 50 first customers. The proposed method was able to solve all of the small size problems and some of the medium and larger ones.

Tables 2–4 provide the details of execution on all the problems that could be solved, "–" means that the computation of node 0 was completed but the proof of optimality could not be reached. The LP column gives the value of the possibly fractional solution obtained at the first node of the branch and price tree. Table 5 compares the success rates of a pure CP approach, the original Column Generation method and the hybrid proposed in this paper. There have been significant improvements in the area of exact methods solving the VRPTW lately, but all of them either use a different decomposition (i.e., Lagrangian) or have added features to the master problem. The proposed method is thus compared with the OR approach of (Desrochers, Desrosiers, and Solomon, 1992) which is a bit older but proposes the same decomposition (same master problem and sub-problem definition). It is also reasonable to think that most of the improvements which have been added to this method in the past years would also have been beneficial to our approach.

During this project the emphasis was put on increasing the flexibility of the method rather than reducing its computational time. The proposed method is much faster than the pure CP approach of (Gaudin, 1997) and slower than those of (Desrochers,

Table 2
Results on the Solomon class C problems.

Problem	Size	UB	LP	Nodes	Vehicles	Distance	Time
C101	25	191.81	191.81	0	3	191.81	3.99
C101	50	363.25	363.25	0	5	363.25	58.54
C101	100	891.38	828.94	0	10	828.94	305.13
C102	25	238.65	190.74	0	3	190.74	30.07
C102	50	363.25	362.17	0	5	362.17	127.63
C102	100	973.18	828.94	0	10	828.94	3407.61
C103	25	190.74	190.74	0	3	190.74	175.69
C103	50	434.25	362.17	0	5	362.171	745.26
C104	25	187.45	187.50	0	3	187.50	891.21
C104	50	423.96	358.89	0	5	358.883	1963.43
C105	25	191.81	191.81	0	3	191.81	4.95
C105	50	363.25	363.25	0	5	363.25	66.45
C105	100	896.14	828.94	0	10	828.94	333.57
C106	25	191.81	191.81	0	3	191.81	3.92
C106	50	363.25	363.25	0	5	363.25	67.37
C106	100	936.90	828.94	0	10	828.94	621.58
C107	25	191.81	191.81	0	3	191.81	5.63
C107	50	363.25	363.25	0	5	363.25	79.72
C107	100	854.31	828.94	0	10	828.94	426.70
C108	25	191.81	191.81	0	3	191.81	26.22
C108	50	390.39	363.25	0	5	363.25	150.63
C108	100	949.80	828.94	0	10	828.94	2476.52
C109	25	191.81	191.81	0	3	191.81	100.72
C109	50	363.25	363.25	0	5	363.25	948.75
C109	100	1042.20	828.94	0	10	828.94	2838.02

Desrosiers, and Solomon, 1992) when we consider the difference in computer[3] performance over the years.

The results, in terms of the number of solved problems, obtained by our method are comparable to those provided in the literature by similar OR algorithms, but the constraint programming paradigm yields a more flexible approach. For instance, precedence constraints or any kind of logical constraints on customers or vehicles can be easily supported.

It is also important to note that almost all OR column generation methods require the input data of the problem to be discrete. Since the complexity of most dynamic programming approaches is pseudo-polynomial on the resources (distances, travel times, capacities) width, most authors have truncated all distances to the first decimal point before multiplying them by ten.

This means that some arcs, which are not feasible when distances are computed with real numbers, become feasible. Of course one could argue that this precision is enough to ensure that all solutions found will be feasible with respect to real distances, but it is not case. We have found, among the 50 customers problem, that the solution reported for instance R105 is infeasible when true travel times are used. This incor-

Table 3
Results on the Solomon class R problems.

Problem	Size	UB	LP	Nodes	Vehicles	Distance	Time
R101	25	629.14	618.33	0	8	618.33	4.60
R101	50	1106.90	1046.70	0	12	1046.70	64.55
R101	100	1706.07	1636.39	18	20	1642.88	1668.20
R102	25	561.44	547.40	1	7	548.11	35.10
R102	50	929.29	911.44	0	11	911.44	329.42
R103	25	464.82	455.70	0	5	455.70	46.05
R103	50	808.72	775.65	0	9	775.65	447.41
R104	25	446.13	417.96	0	4	417.96	86.56
R105	25	538.40	531.54	0	6	531.54	7.74
R105	50	982.44	900.94	75	9	914.31	1422.80
R105	100	1473.63	1352.27	–	–	–	–
R106	25	483.13	458.28	6	5	466.48	111.93
R106	50	881.82	794.91	3	8	795.25	641.43
R107	25	457.42	425.27	0	4	425.27	70.58
R107	50	781.53	709.69	24	7	713.50	14394.20
R108	25	428.60	397.74	27	4	398.30	1076.94
R109	25	463.99	442.62	0	5	442.62	8.48
R109	50	888.93	777.82	–	–	–	–
R110	25	448.25	439.69	7	5	445.18	171.33
R110	50	784.36	697.52	–	–	–	–
R111	25	450.38	428.29	3	4	429.70	128.51
R111	50	786.54	698.91	–	–	–	–
R112	25	414.99	388.16	39	4	394.10	1124.26
R112	50	663.12	618.29	–	–	–	–

Table 4
Results on the Solomon class RC problems.

Problem	Size	UB	LP	Nodes	Vehicles	Distance	Time
RC101	25	529.83	409.24	231	4	462.16	669.14
RC101	50	948.58	763.44	–	–	–	–
RC101	100	1778.89	1588.97	–	–	–	–
RC102	25	451.54	352.74	0	3	352.74	50.33
RC102	50	907.44	724.11	–	–	–	–
RC103	25	388.17	333.92	0	3	333.92	145.16
RC103	50	851.10	647.37	–	–	–	–
RC104	25	362.36	307.14	0	3	307.14	161.43
RC104	50	557.04	546.51	0	5	546.51	3025.67
RC105	25	519.98	412.38	0	4	412.38	85.30
RC105	50	953.28	763.44	–	–	–	–
RC106	25	449.94	346.50	0	3	346.50	30.03
RC106	50	917.14	668.17	–	–	–	–
RC107	25	363.94	298.95	0	3	298.95	18.10
RC107	50	776.97	604.48	–	–	–	–
RC108	25	295.44	294.99	0	3	294.99	72.47

Table 5
Percentage of problem solved according to size.

Problem size	Pure CP[a]	Column generation[b]	Hybrid CG–CP
25 customers	20%	100%	100%
50 customers	7%	48%	55%
100 customers	0%	24%	28%

[a] (Gaudin, 1997).
[b] (Desrochers, Desrosiers, and Solomon, 1992).

rect solution could be computed in only 25% of the normal CPU time by truncating all distances to the first decimal point in our method.

Finally, all Constraint Programming models were implemented and solved using ILOG Solver 5.2 (ILOG S.A., 2002c) while the Set Covering model was implemented in ILOG Cplex 7.5. (ILOG S.A., 2002a). The use of fast CP solver is primordial since a lot of time is spent generating the columns at each iterations. On the other hand, the Set Covering model generally runs in less 0.1 seconds and could thus be solved by a less efficient linear code.

9. Conclusion

We have presented a Constraint Programming Based Column Generation method that addresses vehicle routing problems. The proposed method is flexible since it can handle not only resource based constraints but almost any structure of constraints, while still providing acceptable performance on known benchmark problems.

We also introduced two Arc Elimination algorithms useful in solving any Negative Reduced Cost Shortest Path Problem either in a Column Generation framework or in a Lagrangian Decomposition method (Kohl and Madsen, 1997).

We think that those components have enriched the framework of constraint programming Based Column Generation by enabling the solution of cyclic problems and by proposing tools that will accelerate the execution time of existing methods.

Acknowledgments

This research as been partially funded by the Natural Science and Engineering Research Council of Canada and Le Fonds Québécois de la Recherche sur la Nature et les Technologies. Thanks are also due to anonymous referees for useful comments.

Notes

1. The value j which minimizes d_{ij}.
2. k is given as a parameter to the method.
3. All experimentations where performed on a SUN computer running at 400 MHz.

References

Carpaneto, G., S. Martello, and P. Toth. (1994). "Algorithm and Codes for the Assignment Problems." *Annals of Operations Research* 78, 146–161.

Chabrier, A. (2000). "Using Constraint Programming Search Goals to Define Column Generation Search Procedures." In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*, pp. 19–27.

Dantzig, G.B. and P. Wolfe. (1960). "Decomposition Principles for Linear Programs." *Operations Research* 8, 101–111.

Dell'Amico, M., F. Maffioli, and P. Varbrand. (1995). "On Prize-Collecting Tours and the Asymmetric Travelling Salesman Problem." *International Transactions in Operational Research* 2(3), 297–308.

Desrochers, M., J. Desrosiers, and M.M. Solomon. (1992). "A New Optimisation Algorithm for the Vehicle Routing Problem with Time Windows." *Operations Research* 40, 342–354.

Desrosiers, J., Y. Dumas, M.M. Solomon, and F. Soumis. (1995). "Time Constrained Routing and Scheduling." In M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser (eds.), *Network Routing, Handbooks in Operations Research and Management Science*, Vol. 8, pp. 35–139. Amsterdam: North-Holland.

Desrosiers, J., M.M. Solomon, and F. Soumis. (1993). "Time Constrained Routing and Scheduling." In *Handbooks of Operations Research and Management Science*, Vol. 8, pp. 35–139.

Fahle, T. and M. Sellmann. (2002). "Constraint Programming Based Column Generation with Knapsack Subproblems." *Annals of Operations Research* 115, 73–94.

Focacci, F., A. Lodi, and M. Milano. (1999). "Solving TSP through the Integration of OR and CP Techniques." *Electronic Notes in Discrete Mathematics* 1.

Focacci, F. and M. Milano. (April 2001). "Connections and Integrations of Dynamic Programming and Constraint Programming." In *Proceedings of the 3rd Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Ashford, Kent, UK, pp. 125–138.

Gaudin, E. (1997). "Contribution de la programmation par contraintes au transport: définition et résolution d'un modèle complexe de gestion de flotte." Ph.D. Thesis, Université Paris 7.

ILOG S.A. (2002a). ILOG Cplex.

ILOG S.A. (2002b). ILOG Dispatcher.

ILOG S.A. (2002c). ILOG Solver.

Junker, U., S.E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. (1999). "A Framework for Constraint Programming Based Column Generation." In *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, Vol. 1713, pp. 261–274. Berlin: Springer.

Kohl, N. and O.B.G. Madsen. (1997). "An Optimization Algorithm for the Vehicle Routing Problem with Time Windows Based on Lagrangian Relaxation." *Operations Research* 45(3), 395–407.

Mingozzi, A., M.A. Boschetti, S. Ricciardelli, and L. Bianco. (1999). "A Set Partioning Approach to the Crew Scheduling Problem." *Operations Research* 47(6), 873–888.

Pesant, G., M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau. (1998). "An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows." *Transportation Science* 32, 12–29.

Pesant, G., M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau. (1999). "On the Flexibility of Constraint Programming Models: From Single to Multiple Time Windows for the Traveling Salesman Problem." *European Journal of Operational Research* 117, 253–263.

Régin, J.-C. (1994). "A Filtering Algorithm for Constraints of Difference in CSPs." In *Proc. of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pp. 362–367.