Solving small VRPTWs with Constraint Programming Based Column Generation

Louis-Martin Rousseau, Michel Gendreau, Gilles Pesant  
Center for Research on Transportation  
Université de Montréal,  
CP 6128 Succ Centre-Ville  
Montréal, Canada, H3C 3J7  
e-mail: {louism,michelg,pesant}@crt.umontreal.ca

Abstract  
Constraint programming based column generation is a hybrid optimization framework recently proposed [11] which uses constraint programming to solve column generation subproblems. In the past, this framework has been used to solve scheduling problems that are naturally acyclic and has done so very efficiently. This paper attempts to solve problems that are cyclic by nature, such as routing problems, by solving the elementary shortest path problem with constraint programming. We also introduce new implied constraints which can be useful in the general framework.

1 Introduction  
Vehicle Routing Problems (VRP) are omnipresent 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 $b_c$ and before time $e_c$. One can wait in case of early arrival, but late arrival is not permitted.

Column generation is a powerful method used to solve Set Partitioning problems. Introduced by Dantzig and Wolfe [2] 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 [5]. In the first application to the field of Vehicle Routing Problems with Time Windows, presented by Desrochers et al. [4], the basic idea was to decompose the problem into sets of customers visited by the
same vehicle (routes) and to select the optimal 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 \( r \), \( c_r \) 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 the route \( r \); and \( x_r \) a Boolean variable specifying whether the route \( r \) is chosen or not, the Set Partitioning Problem is defined as \((S)\):

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

This formulation however poses a few problems. Firstly since it is impractical to construct and 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. That is why, in general, the following relaxed Set Covering formulation is used instead as a Master Problem \((M)\):

\[
\begin{align*}
\min & \sum_{r \in R'} c_r x_r \\
\text{s.t} & \sum_{r \in R'} a_{ir} x_r \geq 1 & \quad \forall i \in \{1..N\} \\
& x \in (0, 1)^N
\end{align*}
\]

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 \( \lambda_i \) 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 cost (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 an integer 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 pass twice through the same node) has been relaxed [6]. This method is very efficient. But since the problem allows negative weight on the arcs, the path produced may contain cycles. 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 the routing domain of application and thus concentrates on the 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 total distance of the routes minus the sum of all the dual values collected.

There have been few attempts to combine column generation and constraint programming. Chabrier [1] presented a framework that uses constraint programming search goals to guide the column generation process. Junker et al. [11] have proposed a framework they call constraint programming based column generation which uses CP to solve constrained shortest path subproblems in column generation. This 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 problem by working on the smaller original cyclic graph. The use of CP also allows the use 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, ordering constraints amongst visits or any logical implication satisfying special customer demands.

This paper presents the model chosen to describe the Profitable Tour Problem, introduces some implied constraints and discusses the transformation of a PTP into an Asymmetric Travelling Salesman Problem (ATSP) in order to use known lower bounds. The result section evaluates the impact of the different components and compares the proposed method with two other exact algorithms that solve the same problem.

2 CP Based Column Generation

The original motivation to use of constraint programming based column generation ([11]) 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 [11] propose to use a single set variable $Y$ which contains the node 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 set of $Y$. An incremental implementation of SP algorithm ensures that the filtering is done efficiently.

Fahle and Sellmann [7] later enriched the CP based column generation framework with a Knapsack constraint for problems which present knapsack subproblems.

3 Model

Since the problem considered is cyclic, simple set variables cannot be used to record solutions (as proposed in [11]) since the construction of a complete solution from the set of included visits would require solving a TSP. Thus the need for a new model. Let $N$ be the set of all customers. The depot is copied $2V$ times, where $V$ is the number of vehicles, so that each route starts and ends at a different depot. Let then $I$ and $F$ be respectively the set of initial and final depots.

3.1 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 $i$.
- $\lambda_i$: Dual value associated with node $i$.
- $C$: Capacity of the vehicles.
- $c_{ij} = d_{ij} - \lambda_i$: Reduced cost to go from node $i$ to node $j$.

3.2 Variables

- $S_i \in N \cup F \quad \forall i \in N \cup I$: Successor of node $i$.
- $P_i \in N \cup I \quad \forall i \in N \cup F$: Predecessor of node $i$.
- $T_i \in [a_i, b_i] \quad \forall i \in N \cup I \cup F$: Time of visit of node $i$.
- $L_i \in [0, C] \quad \forall i \in N \cup I \cup F$: Truck load after visit of node $i$.

3.3 Constraints

- $S_i = j \Leftrightarrow P_j = i$: $S - P$ Consistency constraints.
- $AllDiff(S)$: Conservation of flow.
- $NoSubTour(S)$: SubTour elimination constraint.
- $S_i = j \Rightarrow T_i + t_{ij} \leq T_j$: Time window constraints.
- $S_i = j \Rightarrow L_i + l_j = L_j$: Capacity constraints.

The nodes which are left out of the chosen path have their $S_i$ and $P_i$ value fixed to the value $i$. The $NoSubTour$ constraint is taken from the work of Pesant et al. [14]. 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.
4 Additional Constraints

In order to improve solution time, we introduce implied constraints, which do not modify the solution set but allow improved pruning and filtering. Most of the constraints introduced in [14], which perform filtering base on time window and capacity narrowing, are included in the present method. We also propose two new implied constraints.

4.1 CanBeConnected Constraint

The nature of our problem led us to develop a more complex implied and bounding constraint, which we call the CanBeConnected (CBC) constraint. The problem addressed by this constraint arises from the fact that a typical first fail solution strategy leads to the construction of small, disjointed, pieces of solution. Due to resource constraint, it is possible that those pieces of solution are incompatible and that no solution exists. Since it is clearly advantageous to discover such inconsistencies as early as possible, we devised the CBC constraint.

4.1.1 General CBC algorithm

1. Reduction of the graph: Since we propagate the CBC constraint at each node of the search tree, only partial information is available for pruning. A first step is thus taken to filter out from the graph all nodes $i$ for which $i \in S_i$. This can be safely done under the condition that we have the triangle inequality\(^1\) on all of the resources. Once this is done we are left with the $S$ variables associated with the last node of each partial chain; each of those variables’ domain contains only nodes that are the start of those same chains.

2. Finding a feasible solution: Since the graph obtained from step 1 is considerably reduced in size, it is possible to find a solution (instantiate all the $S$ and $P$ remaining variables) or prove infeasibility very quickly. This problem corresponds to a small resource constrained TSP which we can solve very efficiently. Here, it is feasibility that interests us so we only perform the search until a first solution is identified. We however need to relax the upper bound on the objective, since the nodes we have discarded could have permitted us to lower the objective value. If no solution is found, then we have detected an inconsistency and the global search procedure acts accordingly.

---

\(^1\)The $a-b-c$ trip will always consume more of every resource than the $a-c$ trip, for every customer $a,b,c$. 
3. **Bounding**: Since a solution to this reduced problem is also a solution to the general problem where all removed nodes are simply not visited, we can use this constraint to calculate upper bounds. If the value of the solution found in the CBC constraint is lower than the upper bound on the objective value, we simply store that solution and update the upper bound.

This constraint can be used in most of constraint programming based routing models since it is valid not only in the context of PTPs but also TSPs and VRPs. Although the bounding portion of the algorithm is specialized to the PTP context we believe that other routing problems could benefit from the use of the CBC constraint filtering.

4.2 **Arc Elimination Constraints**

We also introduce a new family of implied 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 by [8]) 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. in [13]. 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 the $S - P$ consistency constraint.

If the dual value associated with a customer is not sufficiently large, it may then not be worth the trip to visit this customer. We use this concept to propose three arc eliminating constraints that can significantly reduce the size of the original graph. Again, these constraints are valid only if the triangular inequality is respected for the resources. Otherwise the visit of an intermediate customer could yield savings in some resources and thus allow the visit of extra customers.

4.2.1 **Arc Elimination of Type 1**

The Arc Elimination constraint of type one 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.

\[
\forall i \in \{0..N\}, \forall j \in S_i : j \neq i \text{ impose that } (\forall k \in S_j : k \neq i \neq j \ (d_{ij} + d_{jk} - \lambda_j > d_{ik})) \Rightarrow S_i \neq j
\]
4.2.2 Arc Elimination of Type 2

The Arc Elimination constraint of type two is defined as follows: given an arc \((i, j)\), if for all other customers \(k\) that are elements of the domain of the predecessor variable of \(i\) \((P_i)\), 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.

\[
\forall i \in \{1..N\}, \forall j \in S_i : j \neq i \text{ impose that } \\
(\forall k \in P_i : k \neq i \neq j \ (d_{ki} + d_{ij} - \lambda_i > d_{kj})) \Rightarrow S_i \neq j
\]

4.2.3 Arc Elimination of Type 3

The Arc Elimination constraint of type three is defined as follows: given an arc \((i, j)\), if for all other customers \(k \in P_i\) and \(m \in S_j\) it is always cheaper to go directly from \(k\) to \(m\) \((d_{km})\) than to travel through \(i - j\) \((d_{ki} + d_{ij} + d_{jm} - \lambda_i - \lambda_j)\), then the arc \(i - j\) can be eliminated from the subproblem graph since it will never be part of an optimal solution.

\[
\forall i \in \{1..N\}, \forall j \in S_i : j \neq i \text{ impose that } \\
(\forall k \in P_i, \forall m \in S_j : k \neq i \neq j \neq m \ (d_{ki} + d_{ij} + d_{jm} - \lambda_i - \lambda_j > d_{km})) \Rightarrow S_i \neq j
\]

The complexity of these constraints is in \(O(n^3)\) for the type 1 and 2 and in \(O(n^4)\) for type 3, since we have to perform three (or four) nested loops on the number of customers. However, in practice, since we break out of the two (or three) innermost loops as soon as we identify a \(k\) (or \(m\)) not respecting the condition, we have observed that the algorithm usually consumes time in \(O(n^2)\) (or \(O(n^3)\)).

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 at every node of the search tree before the next branch is selected.

5 Lower Bounds

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 to transform a PTP into an asymmetric TSP (see [3]) by adding \(N\) nodes and \(2N\) arcs (note that in this new
Figure 2: Transformation of a PTP to an ATSP

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 ($\sum_{i \in \{1..N\}} \lambda_i$).

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. The Minimal Spanning Three (MST) is another lower bound obtained by relaxation of the AllDiff constraint. Both of these bounds can be calculated in polynomial time by specialized algorithms.

6 Upper Bound

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 [10] 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 an Dispatcher example file (vrp.cpp). We also took advantage of this preliminary phase to improve the quality of the initial column set ($R_0$). All routes identified during the descent phase are stored in $R'$ and the best solution found is kept as an upper bound.

7 Branching Strategy

The optimal solution to the master problem is obtained once we have proven that no reduced cost path exits. Unfortunately, this solution is not always integer and a branching scheme is thus needed to close the integrality gap. It is not possible 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 \( R' \).

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

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 \( f_{ij} = \sum_{r \in R'} f^*_{ij} x_r \). 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. **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:

   \[
   x_r = 0 \quad \forall r \in R', i \in N, j \notin B_i : f^*_{ij} = 1
   \]

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

   \[
   j \notin B_i \rightarrow j \notin S_i \quad \forall i, j \in \{1..N\}
   \]

5. **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.

8 Experimental Results

This section evaluates the impacts of the components presented on the overall performance of the method. Results are also given on the well-known Solomon problems [15].

8.1 Implied Constraints

To evaluate the role played by the implied constraints we have introduced, we chose to compare their behavior 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 the following tables we report the total number of backtracks and the CPU time needed to prove the optimality of the best found negative reduced cost path. It is important to note that the number of backtracks reported when the CBC constraint is in use also includes the number of backtracks required at step two of the CBC algorithm.

<table>
<thead>
<tr>
<th>Problem</th>
<th>When Domain</th>
<th>Once per Node</th>
<th>Preprocessing</th>
<th>No Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C102</td>
<td>80934</td>
<td>107.97</td>
<td>105383</td>
<td>776.05</td>
</tr>
<tr>
<td>R102</td>
<td>9754</td>
<td>15.52</td>
<td>12219</td>
<td>35.91</td>
</tr>
<tr>
<td>RC102</td>
<td>27339</td>
<td>31.41</td>
<td>34669</td>
<td>51.17</td>
</tr>
</tbody>
</table>

Table 1: Arc Elimination Constraints Propagation Strategies: Number of Backtracks and Time to Solve One PTP.

Table 1 reports the performances of the Arc Elimination constraints with respect to the level of propagation used. The constraint is either applied only once before the search is undertaken (preprocessing), at each node of the search tree (once per node), or when the domain of one of the involved variable is modified (when domain). We can easily conclude that applying the constraint only once per search node is the most efficient approach since the overhead associated with a greater level of propagation dominates the efficiency gain obtained from the reduction in the number of backtracks.

<table>
<thead>
<tr>
<th>Problem</th>
<th>No Constraints</th>
<th>CBC</th>
<th>Arc Elim.</th>
<th>AE + CBC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C102</td>
<td>1811172</td>
<td>517.75</td>
<td>1540813</td>
<td>755.10</td>
</tr>
<tr>
<td>R102</td>
<td>193622</td>
<td>55.53</td>
<td>124862</td>
<td>37.82</td>
</tr>
<tr>
<td>RC102</td>
<td>654465</td>
<td>206.84</td>
<td>506989</td>
<td>134.03</td>
</tr>
</tbody>
</table>

Table 2: Implied Constraint Effectiveness: Number of Backtracks and Time to Solve One PTP.

Table 2 shows the importance of both the CBC and the Arc Elimination constraints in finding rapidly the optimal PTP solution. We note that the Arc Elimination constraints (in a once per node application) seem to be more effective than the CBC constraint and that the combination of the two produces even better results.

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.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25 Customers</td>
<td>20 %</td>
<td>100 %</td>
<td>86 %</td>
</tr>
<tr>
<td>50 Customers</td>
<td>7 %</td>
<td>48 %</td>
<td>24 %</td>
</tr>
<tr>
<td>100 Customers</td>
<td>0%</td>
<td>24 %</td>
<td>7 %</td>
</tr>
</tbody>
</table>

Table 3: Percentage of Problem Solved According to Size

The proposed method was able to solve most of the small size problems and some of the medium and larger ones. Problems which exceeded a one hour time limit were declared unsolved by our approach. For all these instances the time limit was reached during the first call to the subproblem.

Table 3 compares the success rates of a pure CP approach, the original Column Generation method and the hybrid proposed in this paper. The time needed to solve most instances is comparable to those reported in [9], however they are significantly greater than those of [4]. The results obtained by our method are not as good as those provided in the literature by 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. The results obtained on the Solomon’s problems are good and the proposed method constitutes, to our knowledge, the best CP-based exact algorithm solving the VRPTW.

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.

The research of this method has led to the development of a new implied constraint for constraint programming based routing models, the CBC constraint, which could be applied in numerous applications solving Travelling Salesman Problems or Vehicle Routing Problems. We also introduced three 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 [12].

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.

References


