New dynamic programming algorithms for the resource-constrained elementary shortest path problem

Giovanni Righini, Matteo Salani[*]

Dipartimento di Tecnologie dell'Informazione
Università degli Studi di Milano
Via Bramante 65, 26013 Crema, Italy

January 2005

Abstract

The resource-constrained elementary shortest path problem arises as a pricing subproblem in branch-and-price algorithms for vehicle routing problems with additional constraints. We address the optimization of the resource-constrained elementary shortest path problem and we present and compare three methods. The first method is a well-known exact dynamic programming algorithm improved by new ideas, such as bi-directional search with resource-based bounding. The second method consists of a branch-and-bound algorithm, where lower bounds are computed by dynamic programming with state space relaxation; we show how bounded bi-directional search can be combined with state space relaxation and we present different branching strategies and their hybridization. The third method, called decremental state space relaxation, is a new one; exact dynamic programming and state space relaxation are two special cases of this new method. The experimental comparison of the three methods is definitely favourable to decremental state space relaxation. Computational results are given for different kinds of resources, arising from the capacitated vehicle routing problem, the vehicle routing problem with distribution and collection and the vehicle routing problem with capacities and time windows.

Keywords: shortest path, vehicle routing, column generation, dynamic programming, branch-and-bound.

[*]Corresponding author: (salani@dti.unimi.it)

1 Introduction

Branch-and-price is one of the most effective techniques for the exact optimization of vehicle routing problems (VRP) with additional constraints. At each node of a branch-and-bound tree a relaxation of the set covering reformulation of the problem is solved via column generation. Algorithms based on this technique can solve constrained VRP instances with more than 100 vertices (see for instance Kohl et al. [20]). When vehicle routing problems with additional constraints are solved via column generation and branch-and-price, the pricing problem requires to find a resource-constrained elementary path of minimum cost between two given vertices of a weighted graph, with positive costs on the arcs and non-negative prizes on the vertices. The cost of the path is given by the sum of the costs of the arcs traversed minus the sum of the prizes collected at the vertices visited. Exact optimization is needed to find new columns with negative reduced cost or to prove that none of them exists. For a detailed exposition of branch-and-price methods for vehicle routing problems we refer the reader to Desrosiers et al. [11], Bramel and Simchi-Levi [4] and Cordeau et al. [6].

If the underlying graph may have negative cost cycles (which is the case when there are prizes on the vertices), the resource-constrained elementary shortest path problem (RCESPP) is strongly NP-hard: the proof is due to Dror [13]. The most commonly used technique to solve the RCESPP to optimality is dynamic programming, relying upon the seminal work by Desrochers [7] for the resource-constrained shortest path problem (RCSPP) in which the solution is not required to be elementary. Other methods, based on Lagrangean relaxation, were proposed by Handler and Zang [17] and Beasley and Christofides [3] and were recently examined by Dumitrescu and Boland [14], who devised improved preprocessing and bounding techniques. However these methods require a graph free from negative cost cycles, so that the Lagrangean subproblem is a polynomially solvable shortest path problem. The case with negative cost cycles was considered by Feillet et al. [15], who suggested some improvements to the basic Desrochers' algorithm [7]. For a recent survey on models and algorithms for the RCSPP and the RCESPP we refer the reader to Irnich and Desaulniers [18].

In this paper we consider the problem of computing optimal solutions to the RCESPP, as in Feillet et al. [15], and we present three different approaches: exact dynamic programming, branch-and-bound based on state space relaxation and decremental state space relaxation. All of them use dynamic programming but in different ways: in the first case an exact dynamic programming algorithm computes the optimal solution of the RCESPP; this approach was taken for instance by Feillet et al. [15] and Righini and Salani [22]. In the second case a dynamic programming algorithm with state space relaxation is used to optimize the RCSPP, where cycles are allowed. This gives lower bounds corresponding to non-elementary paths and these lower bounds are exploited in a branch-and-bound framework. The third method, decremental state space relaxation, is original and includes both exact dynamic programming and state space relaxation as special cases.

The performance of exact dynamic programming algorithms for the RCESPP can be significantly improved through bi-directional search and resource-based bounding, as shown by Righini and Salani [22]. In this paper we review them and we show that they can be applied also to dynamic programming with state space relaxation and decremental state space relaxation. We report on the outcome of experimental comparisons between these methods when solving RCESPPs with different kinds of resource constraints. In particular we consider three variations of the RCESPP arising from three well-known vehicle routing problems with additional constraints, namely the capacitated vehicle routing problem (CVRP), the vehicle routing problem with distribution and collection (VRPDC) and the vehicle routing problem with capacities and time windows (CVRPTW).

The outline of this paper is the following: in Section 2 we formally define the RCESPP; in Section 3 we review the dynamic programming algorithms for its exact optimization; in Section 4 we review bounded bi-directional dynamic programming; in Section 5 we analyze state space relaxation to compute lower bounds; in Section 6 we present the branch-and-bound algorithm; in Section 7 we introduce decremental state space relaxation; in Section 8 we report on computational results; in Section 9 we point out some conclusions. The content of Sections 3 and 4 is a review of concepts already described in [22], which have been recalled here in order to make this paper self-contained.

2 Problem definition

The RCESPP is defined as follows: a graph $\mathcal{G}(\mathcal{V}, \mathcal{A})$ is given, where the vertex set $\mathcal{V}$ is made by a set of vertices $\mathcal{N}$ representing N customers and two vertices s and t representing the depot. A non-negative cost c_{ij} is associated with each arc $(i, j) \in \mathcal{A}$; arc costs correspond to shortest paths and therefore they satisfy the triangle inequality. A non-negative prize λ_i is associated with each vertex $i \in \mathcal{N}$, a non-negative cost λ_0 is associated with the depot. A vehicle must go from s to t, visiting a subset of the other vertices; no cycles are allowed. The objective is to minimize the cost, given by the sum of the costs of the arcs traversed minus the sum of the prizes collected at the vertices visited. In a column generation framework this corresponds to generate columns of minimum reduced cost for the linear relaxation of the set covering reformulation of a VRP: λ_i is the dual multiplier associated with the covering constraint of vertex i and λ_0 is the dual multiplier asociated with the constraint on the maximum number of available vehicles.

These definitions of the problem are common to all RCESPP versions arising from the different routing problems we consider. Additional constraints, that depend on the kind of vehicle routing problem at hand, are modeled as resource constraints and they are specified hereafter.

Capacity. In the CVRP a non-negative integer demand d_i is associated with each vertex $i \in \mathcal{N}$ and a positive integer vehicle capacity Q is given. The sum of the demands of the nodes visited by the same vehicle cannot exceed Q.

Distribution and collection. In the VRPDC each vertex i has two non-negative integer quantities p_i and d_i associated with it, representing respectively the amount of load to be collected and to be delivered at that vertex. Each vehicle has a positive integer capacity Q, it leaves the depot carrying the total amount of load it must deliver and returns to the depot carrying the total amount of load it has collected. The capacity cannot be exceeded anywhere along the path.

Capacity and time windows. In the CVRPTW a non-negative integer service time θ_i and a time window $[a_i, b_i]$, defined by two non-negative integers, are associated with each vertex $i \in \mathcal{N}$ and the service at each visited vertex must start inside its time window. If the vehicle arrives at vertex i before time a_i, it waits until a_i. The traveling time from any vertex i to any vertex j is a non-negative integer datum v_{ij}.

We chose these three problems, because they offer a significant mix of different characteristics. In the CVRP there is only one resource, whose consumption depends on the vertices visited. In the VRPDC there are two resources associated with the vertices visited and they are interacting: the consumption of one of them also depends on the consumption of the other. In the CVRPTW there are two resources, one associated with the vertices visited and the other associated with the arcs traversed. In all cases resources are subject to a global constraint on their overall consumption along the s-t path, with the exception of the case with time windows, where a resource (time) is subject to local constraints, one for each vertex visited.

3 Exact dynamic programming

The starting point for our exposition is the reaching algorithm of Desrochers [7] for the RCSPP, that is an extension of the well-known shortest path algorithm of Ford and Bellman (see [2]). The algorithm assigns *states* to each vertex: each state associated with vertex i represents a path from s to i. Each state includes a resource consumption vector R whose component R_r represents the quantity of resource r used along the corresponding path. Each state has an associated cost C and the optimal solution corresponds to a minimum cost state associated with vertex t. The algorithm repeatedly extends each state to generate new states. The extension of a state corresponds to appending an additional arc (i, j) to a path from s to i, obtaining a path from s to j. This operation is repeated until all states have been extended in all feasible ways. This dynamic programming algorithm, devised for the RCSPP, can be adapted to solve the RCESPP on graphs with negative cost cycles. To this purpose Beasley and Christofides [3] proposed to add to the state an additional binary resource for each vertex $i \in \mathcal{N}$; there is only one unit available for each dummy resource and it is consumed when the correponding vertex is visited. The consumption of the N dummy resources is indicated by a vector S initialized at 0. Note that S does not keep any information about the order in which the vertices are visited. Hence in the basic exact dynamic programming algorithm for the RCESPP each state is represented by a label of the form (S, R, C, i).

When a label (S, R, C, i) associated with vertex i is extended to generate another feasible label (S', R', C', j) associated with vertex j, the resource consumption vectors and the cost are updated and the new state is checked for feasibility, as follows.

Cost. The cost is inizialized at 0 at vertex s and it is updated according to the formula

$$C' = C - \lambda_i/2 + c_{ij} - \lambda_j/2 \tag{1}$$

where $\lambda_i = -\lambda_0$ if $i = s$ and $\lambda_j = -\lambda_0$ if $j = t$.

Dummy resources. The dummy resources vector S is initialized at 0 at vertex s and the update rule is:

$$S'_k = \begin{cases} S_k + 1 & k = j \\ S_k & k \neq j \end{cases}$$

A state (S, R, C, i) corresponds to an elementary path only if $S_k \leq 1 \ \forall k \in \mathcal{N}$.

According to the different kind of resources considered the extension rules and the feasibility test on R take different forms.

Capacity. The capacity constraint is modeled by a single resource, representing the amount of capacity still available along the path. Let q be the amount of resource consumed. When a vehicle leaves vertex s all the resource is available, that is $q = 0$. Every time a vertex is visited, q is increased by the demand of that vertex. Hence the extension rule is:

$$q' = q + d_j \tag{2}$$

A state (S, q, C, i) is feasible only if $q \leq Q$.

Distribution and collection. In this case the capacity constraint is taken into account by two additional resources, whose consumption is indicated by π and δ. The first resource at vertex i is the amount of load that the vehicle can pick-up after visiting i. Its consumption π increases after every pick-up operation, because when the vehicle visits vertex i, it consumes p_i units of this resource. The second resource at vertex i indicates the amount of load that the vehicle can deliver after visiting i. This quantity is equal to the difference between the capacity Q and the maximum amount of load that has been on board of the vehicle since its departure from s up to i. Initially Q units are available for the second resource and the available resource decreases each time a delivery operation is performed but it may decrease also after pick-up operations: the maximum amount the vehicle can deliver after visiting i cannot be greater than the maximum amount it can pick-up after visiting i. Hence both π and δ are initialized at 0 and the extension rule is:

$$\pi' = \pi + p_j$$
$$\delta' = \max\{\delta + d_j, \pi + p_j\}$$

A state (S, π, δ, C, i) is feasible only if $\pi \leq Q$ and $\delta \leq Q$. Note that for the definition of π and δ the latter condition implies the former.

Capacity and time windows. In this case the time elapsed is a consumed resource, monotonically increasing along the path. To represent the capacity constraint and the time window constraints, we need two resources, whose consumption is respectively indicated by q and τ: they are the capacity and the time consumed up to the beginning of service at each vertex. Both of them are initialized at 0 and the extension rules are:

$$q' = q + d_j$$
$$\tau' = \max\{\tau + \theta_i + v_{ij}, a_j\}$$

A state (S, q, τ, C, i) is feasible only if $q \leq Q$ and $\tau \leq b_i$.

The effectiveness of the dynamic programming algorithm heavily depends on the number of states generated. Hence it is essential to fathom feasible states which cannot lead to the optimal solution. To this purpose suitable dominance tests are always performed when states are extended, so that the algorithm records only non-dominated states. The dominance test is the following. Let (S', R', C', i) and (S'', R'', C'', i) be the labels of two states associated with vertex i; then (S', R', C', i) dominates (S'', R'', C'', i) only if

$$S' \leq S''$$
$$R' \leq R''$$
$$C' \leq C''$$

and at least one of the inequalities is strict.

When the consumption of some resource is non-negative and obeys the triangle inequality, the domination rule can be made stronger, as pointed out by Feillet et al. [15]. The idea is to identify vertices which cannot be visited in any feasible extension of a given state owing to the limits on the resources. These vertices are called *unreachable*. When a vertex is found to be unreacheable from a given state, the consumption of the corresponding dummy resource in that state can be set to 1, as if the vertex had already been visited. This allows the dynamic programming algorithm to identify a larger number of dominated states and to fathom them, thus reducing the computation time. We incorporated this method in all the algorithms we considered: the triangle inequality is certainly satisfied by the resources whose consumption occurs at the vertices, such as the RCESPP with capacity and the RCESPP with distribution and collection. The applicability to the case with time windows, where resource τ is consumed on the arcs, depends on whether the traveling times satisfy the triangle inequality or not; to apply the techniques of Feillet et al. to the time resource, we assumed $v_{ij} = c_{ij}\ \forall(i,j) \in \mathcal{A}$ in our tests.

The order in which the states are extended may be very important for the effectiveness of the overall algorithm. Here we consider label-correcting algorithms like those of Desrosiers et al. [12] and Feillet et al. [15]. States are explored according to the vertices they are associated with. All vertices are cyclically visited and for each vertex the algorithm extends all states that have not yet been extended. States associated with the same vertex can be sorted according to a secondary criterion, for instance according to the cost or the consumption of a certain resource. In the three cases we have considered states associated with the same vertex are sorted according to the values of q, π and τ respectively.

Label-setting algorithms have also been proposed (see for instance Desrochers and Soumis [9]) but they require an hypothesis stronger than resource consumption monotonicity: in particular there must exists a resource whose consumption is not less than a certain known amount β at each extension. In this case it is possible to define buckets of size β and to mark as permanent all those labels for which the resource consumption falls in the range of the first bucket not yet extended. For a more detailed exposition of label-setting algorithms we refer the reader to Desrosiers et al. [11].

4 Bounded bi-directional dynamic programming

Bounded bi-directional dynamic programming has been recently introduced by Righini and Salani [22] to improve the exact dynamic programming algorithm described above. In bi-directional dynamic programming states are extended both forward from vertex s to its successors and backward from vertex t to its predecessors (see for instance Mingozzi et al. [21] for an application of this idea to a constrained TSP). With each vertex $i \in \mathcal{V}$ we associate forward states indicated by $(S^{fw}, R^{fw}, C^{fw}, i)$ and backward states indicated by $(S^{bw}, R^{bw}, C^{bw}, i)$. A path from s to t is detected each time a forward state $(S^{fw}, R^{fw}, C^{fw}, i)$ and a backward state $(S^{bw}, R^{bw}, C^{bw}, j)$ can be feasibly joined. Hereafter we describe backward extension rules and feasibility tests for backward states. Dominance tests on backward states are identical to those for forward states. We also illustrate the operation of joining forward and backward states to produce s-t paths and we review the idea of resource-based bounding.

4.1 Backward extension and feasibility tests

The backward cost C^{bw} is initialized at 0 at vertex t and whenever a backward state $(S^{bw}, R^{bw}, C^{bw}, j)$ is extended to a backward state $(S'^{bw}, R'^{bw}, C'^{bw}, i)$ the cost is updated according to the formula:

$$C'^{bw} = C^{bw} - \lambda_i/2 + c_{ij} - \lambda_j/2$$

where $\lambda_i = -\lambda_0$ if $i = s$ and $\lambda_j = -\lambda_0$ if $j = t$.

The dummy resources vector S^{bw} is initialized at 0 at vertex t and the extension rule is:

$$S'^{bw}_k = \begin{cases} S^{bw}_k + 1 & k = j \\ S^{bw}_k & k \neq j \end{cases}$$

A backward path is feasible only if $S^{bw}_k \leq 1 \; \forall k \in \mathcal{N}$.

Capacity. Resource consumption q^{bw} in a backward state associated with vertex j represents the overall demand of customers visited from j (included) to t. The consumption q^{bw} is initialized at 0 at vertex t. When a feasible backward path is extended along arc (i, j) from a state $(S^{bw}, q^{bw}, C^{bw}, j)$ to a state $(S'^{bw}, q'^{bw}, C'^{bw}, i)$, the extension rule is:

$$q'^{bw} = q^{bw} + d_i \tag{3}$$

A backward path is feasible only if $q^{bw} \leq Q$.

Distribution and collection. Two resources, whose consumption is indicated by π^{bw} and δ^{bw}, are associated with each backward state. Their meaning, initialization and extension rules are symmetrical to those of forward labels: δ^{bw} indicates the amount of load delivered between j and t and π^{bw} indicates the maximum overall amount of load on board of the vehicle between j and t. When a backward path is extended along arc (i, j) from a state $(S^{bw}, \pi^{bw}, \delta^{bw}, C^{bw}, j)$ to a state $(S'^{bw}, \pi'^{bw}, \delta'^{bw}, C'^{bw}, i)$, the extension rule is:

$$\begin{aligned} \pi'^{bw} &= \max\{\delta^{bw} + d_i, \pi^{bw} + p_i\} \\ \delta'^{bw} &= \delta^{bw} + d_i \end{aligned}$$

A backward path is feasible only if $\pi^{bw} \leq Q$ and $\delta^{bw} \leq Q$ (the former condition implies the latter).

Capacity and time windows. In the case of time windows for the sake of symmetry it is useful to define forward and backward time windows $[a^{fw}_i, b^{fw}_i]$ and $[a^{bw}_i, b^{bw}_i]$ as follows:

$$\begin{aligned} a^{fw}_i &= a_i \\ b^{fw}_i &= b_i \\ a^{bw}_i &= a_i + \theta_i \\ b^{bw}_i &= b_i + \theta_i \end{aligned}$$

The forward time window represents the range of feasible arrival times at vertex i, while the backward time window represents the range of feasible departure times from vertex i. The overall resource availability T is equal to the maximum feasible arrival time at vertex t that is $T = \max_{i \in \mathcal{N} \cup \{s\}} \{b_i^{fw} + \theta_i + v_{it}\}$. The resource consumption τ^{bw} in a backward state associated with vertex j represents the minimum time which must be consumed since the departure from j up to the arrival at t. When a feasible backward path is extended along arc (i, j) from a state $(S^{bw}, q^{bw}, \tau^{bw}, C^{bw}, j)$ to a state $(S'^{bw}, q'^{bw}, \tau'^{bw}, C'^{bw}, i)$, the extension rules are:

$$\begin{aligned} q'^{bw} &= q^{bw} + d_i \\ \tau'^{bw} &= \max\{\tau^{bw} + \theta_j + v_{ij}, T - b_i^{bw}\} \end{aligned}$$

A backward label associated with vertex j is feasible only if $q^{bw} \leq Q$ and $\tau^{bw} \leq T - a_j^{bw}$.

4.2 Joining forward and backward states

Forward and backward paths must be joined together to produce complete s-t paths. When a forward path $(S^{fw}, q^{fw}, C^{fw}, i)$ is joined with a backward path $(S^{bw}, q^{bw}, C^{bw}, j)$ the cost of the resulting s-t path is equal to

$$C^{fw} - \lambda_i/2 + c_{ij} - \lambda_j/2 + C^{bw}$$

The join is subject to certain feasibility conditions on the resources. In particular the feasibility test on dummy resources S imposes that a same vertex can not be visited by both paths.

$$S_k^{fw} + S_k^{bw} \leq 1 \qquad \forall k \in \mathcal{N}$$

The feasibility test on problem-dependent resources R imposes that for each resource the consumption in the overall s-t path can not exceed the overall amount of available resource. Hereafter we define the feasibility tests for each specific kind of resource constraints considered.

Capacity. The feasibility test on the capacity for joining a forward path $(S^{fw}, q^{fw}, C^{fw}, i)$ with a backward path $(S^{bw}, q^{bw}, C^{bw}, j)$ is

$$q^{fw} + q^{bw} \leq Q$$

Distribution and collection. The feasibility conditions to join a forward path $(S^{fw}, \pi^{fw}, \delta^{fw}, C^{fw}, i)$ with a backward path $(S^{bw}, \pi^{bw}, \delta^{bw}, C^{bw}, j)$ are:

$$\begin{aligned} \pi^{fw} + \pi^{bw} &\leq Q \\ \delta^{fw} + \delta^{bw} &\leq Q \end{aligned}$$

Capacity and time windows. The feasibility conditions to join a forward path $(S^{fw}, q^{fw}, \tau^{fw}, C^{fw}, i)$ with a backward path $(S^{bw}, q^{bw}, \tau^{bw}, C^{bw}, j)$ are:

$$q^{fw} + q^{bw} \leq Q$$
$$\tau^{fw} + \theta_i + v_{ij} + \theta_j + \tau^{bw} \leq T$$

4.3 Resource-based bounding

Since all forward and backward states generated by the bi-directional search algorithm are tentatively joined, it is crucial to reduce their number as much as possible. To this purpose we select a *critical resource*, whose consumption is monotone along the paths, and we do not extend states in which at least half of the available amount of that resource has been consumed. This allows to greatly reduce the number of states generated still guaranteeing that the optimal solution will be found. Hereafter we describe how we have chosen the critical resource for each different kind of problem.

Capacity. The critical resource in this case is capacity. Forward and backward states are extended only if their associated resource consumption value, q^{fw} or q^{bw} respectively, is less than $Q/2$.

Distribution and collection. In this case there are two resources; we consider as a critical resource the sum of the resource consumptions $\rho^{fw} = \pi^{fw} + \delta^{fw}$ for forward states and $\rho^{bw} = \pi^{bw} + \delta^{bw}$ for backward states and in both directions we extend only those states for which $\rho < Q$.

Capacity and time windows. In this last case we consider time as the critical resource and we extend only forward states for which $\tau^{fw} < T/2$ and backward states for which $\tau^{bw} < T/2$.

The combination of bi-directional search with resource-based bounding allows to solve larger instances (or the same instances in less time) than mono-directional dynamic programming; detailed experimental results are reported in [22]. In the next section we show how bi-directional search and resource-based bounding can be incorporated into dynamic programming algorithms based on state space relaxation.

We report hereafter the pseudo-code of the bounded bi-directional algorithm. We use the following symbols: Γ_i^{fw} and Γ_i^{bw} are the lists of forward and backward states associated with vertex i; Δ_i^+ and Δ_i^- are the sets of successors and predecessors of vertex i; E is the set of vertices to be examined; $Extend^{fw}(l,k)$ and $Extend^{bw}(l,k)$ are respectively the forward and backward extension procedures, that extend state l to vertex k; they check the resource constraints and produce only feasible states; $EFF(\Gamma, l)$ is the procedure which inserts state l into set Γ applying the domination rules; $Feasible(l_i, l_j)$ checks the resource compatibility of forward state l_i and backward state l_j according to problem-dependent rules; $Save(l_i, l_j)$ saves the solution obtained joining the two states l_i and l_j;

Algorithm 1 RCESPP - Bi-directional dynamic programming

```
// Initialization //
$\Gamma_s^{fw} \leftarrow \{(\mathbf{0}, \mathbf{0}, 0, s)\}$
$\Gamma_t^{bw} \leftarrow \{(\mathbf{0}, \mathbf{0}, 0, t)\}$
for all $i \in \mathcal{V} \setminus \{s\}$ do $\Gamma_i^{fw} \leftarrow \emptyset$
for all $i \in \mathcal{V} \setminus \{t\}$ do $\Gamma_i^{bw} \leftarrow \emptyset$
$E \leftarrow \{s, t\}$
// Search //
repeat
    // Vertex selection //
    Select $i \in E$
    // Forward extension //
    for all $l_i = (S^i, R^i, C^i, i) \in \Gamma_i^{fw}$ do
        for all $j \in \Delta_i^+$ such that $S_j^i = 0$ do
            $l_j \leftarrow Extend^{fw}(l_i, j)$
            $\Gamma_j^{fw} \leftarrow EFF(\Gamma_j^{fw}, l_j)$
            if $\Gamma_j^{fw}$ has changed then $E \leftarrow E \cup \{j\}$
    // Backward extension //
    for all $l_i = (S^i, R^i, C^i, i) \in \Gamma_i^{bw}$ do
        for all $k \in \Delta_i^-$ such that $S_k^i = 0$ do
            $l_k \leftarrow Extend^{bw}(l_i, k)$
            $\Gamma_k^{bw} \leftarrow EFF(\Gamma_k^{bw}, l_k)$
            if $\Gamma_k^{bw}$ has changed then $E \leftarrow E \cup \{k\}$
    $E \leftarrow E \setminus \{i\}$
until $E = \emptyset$
// Join between forward and backward paths //
for all $i \in \mathcal{V}$
    for all $l_i = (S^i, T^i, C^i, i) \in \Gamma_i^{fw}$
        for all $j \in \mathcal{V}$
            for all $l_j = (S^j, T^j, C^j, j) \in \Gamma_j^{bw}$
                if $Feasible(l_i, l_j)$ then $Save(l_i, l_j)$
```

5 State space relaxation

State space relaxation was introduced by Christofides et al. [5] in 1981. The state space $\mathcal{S}$ explored by the dynamic programming algorithm is projected onto a lower dimensional space $\mathcal{T}$ so that each state in $\mathcal{T}$ retains the minimum cost among those of its corresponding states in $\mathcal{S}$ (assuming the objective function must be minimized). In this way the number of states to be explored is drastically reduced; the drawback is that some original state corresponding to an infeasible solution in $\mathcal{S}$ may be projected onto a state corresponding to a feasible solution in $\mathcal{T}$ and therefore the search in the relaxed state space does not guarantee to find an optimal solution but rather a lower bound.

State space relaxation has been used as a method alternative to exact optimization of the pricing problem in branch-and-price algorithms for the VRP with additional constraints (see for instance Desrochers et al. [8]): instead of the optimal value of the pricing problem, a lower bound is obtained. This allows faster convergence of the column generation algorithm at the expense of a weaker lower bound. Columns containing cycles must be eliminated through branching. Here on the contrary we focus on the use of state space relaxation for the exact optimization of the pricing problem by a branch-and-bound algorithm.

Our state space relaxation consists of mapping each state (S, R, C, i) onto a new state (σ, R, C, i), where $\sigma = \sum_{k=1}^{N} S_k$ represents the length of the path, that is the number of vertices visited (excluding s). Since each component of the resource consumption vector R may take on a finite number of values and σ can vary between 0 and N, a dynamic programming algorithm based on state space relaxation must explore only a pseudo-polynomial number of states. From the viewpoint of complexity and computing time this makes a big difference with respect to the exact dynamic programming algorithm in which vector S yields an exponential number of possible states. The surrogate resource consumption σ is initialized as 0 and it is increased by one unit each time a state is extended. Since the state does no longer keep information about the set of already visited vertices, cycles are no longer forbidden; therefore the path is guaranteed to be feasible with respect to the resource constraints but it is not guaranteed to be elementary.

In the state space relaxation algorithm the domination rule is modified as follows: a state (σ', R', C', i) dominates a state (σ'', R'', C'', i) only if

$$\begin{aligned} \sigma' &\leq \sigma'' \\ R' &\leq R'' \\ C' &\leq C'' \end{aligned}$$

and at least one of the inequalities is strict.

This state space relaxation of the RCESPP into the RCSPP can be tightened by eliminating all cycles of length two. This is easily accomplished by a duplication of the labels (see for instance Desrochers et al. [8]). Irnich and Villeneuve [19] proposed a method to eliminate cycles of length $k \geq 3$, but the computational complexity of their method dramatically increases with k. Hence we incorporated in our algorithms the technique to avoid cycles of length two.

The definitions above apply to both forward and backward states when bi-directional search is employed. In such case σ^{fw} and σ^{bw} represent respectively the number of forward extensions from s and the number of backward extensions from t. We bound bi-directional search in the same way described above, that is on the basis of the value of a critical resource.

When bounded bi-directional search is coupled with state space relaxation the join of forward and backward paths becomes critical: both the forward path and the backward path to be joined may contain cycles; moreover a cycle can be produced by the join, even if the two paths are elementary. These two cases are illustrated in figure 1. In addition there may be many different ways to join forward and backward paths providing the same solution. The former issue is addressed in the next section, where branching strategies are illustrated; the latter is addressed hereafter.

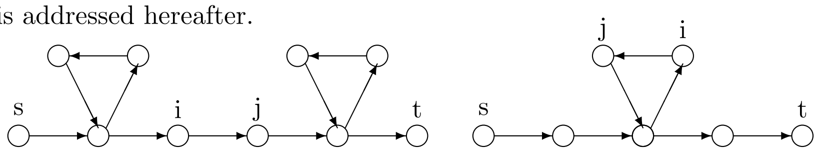

Figure 1: On the left: an *s*-*t* path made of non-elementary paths *s*-*i* and *j*-*t*. On the right: a non-elementary *s*-*t* path made of elementary paths *s*-*i* and *j*-*t*.

5.1 Paths join and solutions uniqueness

The bounded bi-directional dynamic programming algorithm can provide duplicate solutions: consider for instance an *s*-*t* path including vertices i, j and k in this order. If the constraint on the critical resource is not tight, it is possible that forward states for vertices i and j and backward states for vertices j and k are generated. Therefore the same *s*-*t* path can be obtained by joining a forward state of i with a backward state of j as well as joining a forward state of j with a backward state of k. This unpleasant phenomenon can be avoided with an additional test: we accept an *s*-*t* path only when it is produced by the join of a forward state and a backward state, for which the forward and backward consumptions of the critical resource are as close as possible to half the overall consumption *for that s-t path*, that is the two states are as close as possible to the "half way point" along the *s*-*t* path. Let r^{fw} and r^{bw} be the critical resource consumptions in forward and backward paths. Among all possible pairs of forward and backward states producing the same *s*-*t* path we choose the one for which $\phi = |r^{fw} - r^{bw}|$ is minimum. The test is done in constant time for each candidate pair of states, since the position closest to the "half-way point" is detected by direct comparison with the next position along the path if $r^{fw} < r^{bw}$ and with the previous position if $r^{fw} > r^{bw}$. In case of tie between two positions for which ϕ is minimum, we choose the one with $r^{fw} > r^{bw}$. This test guarantees that each *s*-*t* path is generated only once.

We report hereafter the pseudo-code of the state space relaxation algorithm where the terminology used is the same as before. In the joining step the procedure $HalfWay(l_i, l_j)$ checks if the *s*-*t* path obtainable joining the two states l_i and l_j satisfies the "half-way-point" conditions.

Algorithm 2 RCSPP - Bi-directional state space relaxation

```
// Initialization //
$\Gamma_s^{fw} \leftarrow \{(0, \mathbf{0}, 0, s)\}$
$\Gamma_t^{bw} \leftarrow \{(0, \mathbf{0}, 0, t)\}$
for all $i \in \mathcal{V} \setminus \{s\}$ do $\Gamma_i^{fw} \leftarrow \emptyset$
for all $i \in \mathcal{V} \setminus \{t\}$ do $\Gamma_i^{bw} \leftarrow \emptyset$
$E \leftarrow \{s, t\}$
// Search //
repeat
    // Vertex selection //
    Select $i \in E$
    // Forward extension //
    for all $l_i = (\sigma^i, R^i, C^i, i) \in \Gamma_i^{fw}$ do
        for all $j \in \Delta_i^+$ do
            $l_j \leftarrow Extend^{fw}(l_i, j)$
            $\Gamma_j^{fw} \leftarrow EFF(\Gamma_j^{fw}, l_j)$
            if $\Gamma_j^{fw}$ has changed then $E \leftarrow E \cup \{j\}$
    // Backward extension //
    for all $l_i = (\sigma^i, R^i, C^i, i) \in \Gamma_i^{bw}$ do
        for all $k \in \Delta_i^-$ do
            $l_k \leftarrow Extend^{bw}(l_i, k)$
            $\Gamma_k^{bw} \leftarrow EFF(\Gamma_k^{bw}, l_k)$
            if $\Gamma_k^{bw}$ has changed then $E \leftarrow E \cup \{k\}$
    $E \leftarrow E \setminus \{i\}$
until $E = \emptyset$
// Join between forward and backward paths //
for all $i \in \mathcal{V}$
    for all $l_i = (\sigma^i, T^i, C^i, i) \in \Gamma_i^{fw}$
        for all $j \in \mathcal{V}$
            for all $l_j = (\sigma^j, T^j, C^j, j) \in \Gamma_j^{bw}$
                if $Feasible(l_i, l_j)$ and $HalfWay(l_i, l_j)$
                  then $Save(l_i, l_j)$
```

6 Branch-and-bound

In this section we describe a branch-and-bound algorithm which solves the RCE-SPP to optimality, exploiting the RCSPP lower bound given by the bounded bi-directional dynamic programming algorithm with state space relaxation. In Subsection 6.1 we describe the branching policies needed to eliminate cycles: every time the optimal solution of the RCSPP is not elementary, the current node of the search tree is replaced by children nodes in which some additional constraints are added to the RCSPP.

Search policy. The search policy we use to explore the branch-and-bound tree is best-first, that is the open nodes of the tree are ranked according to the value of their associated lower bound and the most promising node is explored first.

Upper bounding. At each node of the branch-and-bound tree and at each iteration of the column generation algorithm a feasible solution is computed with a nearest neighbor heuristic. Starting from the depot s the most convenient vertex among the feasible ones is chosen until the path reaches t. For a vertex to be feasible we check that no resource constraint is exceeded and the vertex have not been visited yet. At each vertex i the algorithm chooses the next feasible vertex j such that $j = argmin_k\{c_{ik} - \lambda_k\}$.

6.1 Branching strategies

We present three different ways to perform branching, namely branching on cycles, branching on arcs and branching on resources. Our algorithm uses hybrid branching strategies in which all these techniques are exploited.

Branching on cycles. First we determine the minimum length cycle in the optimal RCSPP solution. Then k children nodes are generated, where k is the length of the cycle, that is the number of arcs traversed between two visits to the same vertex: at child node $h = 0, \ldots, k-1$ we fix the first h arcs of the cycle and we forbid the $h+1$-th arc. We experimentally observed that, forbidding cycles of length 2, k was very often equal to 3.

Branching on arcs. This binary branching scheme consists of selecting a vertex entered or left by more than one arc in the RCSPP solution. Let (i_1, j) and (i_2, j) be two arcs entering vertex j in the RCSPP solution. Then we partition the arcs entering j into two subsets I_1 and I_2 such that $i_1 \in I_1$ and $i_2 \in I_2$ and we forbid all arcs in I_1 in one child node and all arcs in I_2 in the other.

Branching on resources. When the optimal solution of the RCSPP has a cycle, there exists at least one vertex $\hat{\imath}$ that is visited more than once. The branching strategy consists of adding a constraint on the quantity of critical resource consumed up to the visit of vertex $\hat{\imath}$. This idea was proposed by Gélinas et al. [16] for routing problems with time windows and it can be adapted to any problem with a critical resource whose consumption r is strictly monotone along the path. Given a branching vertex $\hat{\imath}$, let r' and r'' the two values of resource consumption in two states associated with $\hat{\imath}$ with $r' < r''$. Then an integer value $\bar{r}$ is chosen such that $r' < \bar{r} \leq r''$. Two children nodes are generated imposing that the value of r at vertex $\hat{\imath}$ satisfies $r \geq \bar{r}$ in one child node and $r \leq \bar{r} - 1$ in the other.

It is remarkable that the dynamic programming algorithm that computes the lower bound can easily take into account the constraints imposed by all branching techniques. In particular when arc (i, j) is forbidden, it is simply deleted from the graph. The consequence of branching on the critical resource is that each vertex has an associated window $[a^r, b^r]$ of feasible values for the critical resource; when a path reaches that vertex with a critical resource consumption less than a^r, the consumption is set to a^r; when it reaches the vertex with a critical resource consumption greater than b^r, it is declared infeasible and it is discarded. This rule can be applied to both forward and backward states, with different resource windows for constraining forward and backward consumptions.

Hybrid branching. We obtained the best results when we employed hybrid branching strategies in our branch-and-bound algorithm. If either the forward path or the backward path forming the optimal RCSPP solution contain a cycle, we branch on the critical resource: we choose for branching the first vertex visited more than once which is encountered moving along the forward (resp. backward) path from s to t (resp. from t to s); we consider r' and r'' as the resource consumptions at the first (resp. last) two visits of the branching vertex and we choose $\bar{r} = \lceil \frac{r'+r''}{2} \rceil$. If the forward and the backward paths are both elementary but a cycle is generated by their join, we branch on arcs or cycles. When we branch on arcs, the branching vertex is the first vertex visited more than once which is encountered when moving along the path from the half way point forward.

We could not observe a clear domination between the hybrid branching strategies on resource/arcs and resource/cycles. In Section 8 we report on computational results obtained with each of them.

7 Decremental state space relaxation

The exact dynamic programming algorithm forbids multiple visits for each vertex, while the algorithm with state space relaxation does not. We pursued a compromise between these two extreme cases by the following idea: some vertices are identified as *critical*, according to the structure of the optimal RCSPP solution obtained with state space relaxation. Let Θ indicate the set of critical vertices at the current iteration. In the subsequent iteration the dynamic programming algorithm prevents multiple visits the vertices in Θ, still allowing multiple visits to the others. This is easily accomplished by extending the state space relaxation labels with a binary vector S_Θ playing the same role as S in exact dynamic programming. The size of S_Θ is however restricted only to the critical vertices. When S_Θ contains all the vertices the algorithm is equivalent to exact dynamic programming; when S_Θ is empty it is equivalent to the algorithm with state space relaxation. Therefore we indicate this algorithm by decremental state space relaxation (DSSR). The algorithm is run iteratively: every time it produces an optimal solution with cycles, the vertices visited more than once are marked as critical and the algorithm restarts. Let Ψ the set of vertices visited more than once in the optimal solution computed by the DSSR algorithm. If Ψ is not empty, then another iteration is performed with a set of critical vertices equal to $\Theta' = \Theta \cup \Psi$. Hence the set of critical vertices is enlarged at each iteration and eventually the algorithm provides the optimal solution to the RCESPP without having recourse to branching. We report hereafter the pseudo-code of the decremental state space relaxation algorithm. where S_Θ is the vector of dummy resources associated to the critical vertices; procedure $MultipleVisits$ returns the set Ψ of vertices visited more than once in the current optimal path.

8 Experimental results

For our experiments we used the same instances as in Feillet et al. [15] and Righini and Salani [22]; they are derived from the well-known Solomon's VRPTW benchmark. For each kind of RCESPP problem we tested our algorithms on two classes of instances obtained from Solomon's instances by considering the first 50 and 100 nodes. These datasets are divided into *random*, *clustered* and *random-clustered* categories, according to the displacement of the customers. Instances belonging to the same dataset have the customers located in the same way and with the same delivery requests; these instances differ only for the time windows.

When solving the RCESPP with capacities we considered one instance taken from each one of the three Solomon's testsets (namely c101, r101 and rc101); we kept the original customer locations and delivery requests and we neglected the time windows. Then we derived from each original instance ten RCESPP instances with 50 nodes and ten RCESPP instances with 100 nodes, by choosing ten different values for the vehicle capacity from 10 to 100.

For the RCESPP with distribution and collection we kept the original delivery requests and we derived the pickup requests as follows: $p_i = \lfloor 0.8 d_i \rfloor$ if i is odd and $p_i = \lfloor 1.2 d_i \rfloor$ if i is even. We generated ten instances with 50 nodes and ten instances with 100 nodes as before.

Finally, for the RCESPP with capacities and time windows we considered the original instances of Solomon's dataset. In addition we also defined another dataset built on the difficult Solomon's instance c_104; for each vertex i we kept the original starting time of the time window, a_i, and we set the end time as follows: $b_i = a_i + (1+\gamma)\theta_i$ for $\gamma = 0.25 * k$ and $k = 0, \ldots, 24$, where θ_i is the given service time at vertex i.

For each set of instances we generated the prizes λ_i as random integer variables uniformly distributed in $[0, \ldots, 20]$; we set $\lambda_0 = 0$. This data generation technique was devised by Feillet et al. [15] to have a reasonable number of negative cycles. We rounded up all the Euclidean distances between customers to integer values.

All tests were performed on a PC equipped with a PentiumIV 1.6GHz processor with 512Mb RAM. The algorithms were coded in ANSI-C and compiled with gcc 3.0.4.

Algorithm 3 RCESPP - Decremental state space relaxation

```
// Initialization //
$\Psi \leftarrow \emptyset$
$\Theta \leftarrow \emptyset$
repeat
    $\Theta \leftarrow \Theta \cup \Psi$
    $\Gamma_s^{fw} \leftarrow \{(\mathbf{0}, \mathbf{0}, 0, s)\}$
    $\Gamma_t^{bw} \leftarrow \{(\mathbf{0}, \mathbf{0}, 0, t)\}$
    for all $i \in \mathcal{V} \setminus \{s\}$ do $\Gamma_i^{fw} \leftarrow \emptyset$
    for all $i \in \mathcal{V} \setminus \{t\}$ do $\Gamma_i^{bw} \leftarrow \emptyset$
    $E \leftarrow \{s, t\}$
    // Search //
    repeat
        // Vertex selection //
        Select $i \in E$
        // Forward extension //
        for all $l_i = (S^i_\Theta, R^i, C^i, i) \in \Gamma_i^{fw}$ do
            for all $j \in \Delta_i^+$ such that $j \notin \Theta$ or $S^i_j = 0$ do
                $l_j \leftarrow Extend^{fw}(l_i, j)$
                $\Gamma_j^{fw} \leftarrow EFF(\Gamma_j^{fw}, l_j)$
                if $\Gamma_j^{fw}$ has changed then $E \leftarrow E \cup \{j\}$
        // Backward extension //
        for all $l_i = (S^i_\Theta, R^i, C^i, i) \in \Gamma_i^{bw}$ do
            for all $k \in \Delta_i^-$ such that $k \notin \Theta$ or $S^i_k = 0$ do
                $l_k \leftarrow Extend^{bw}(l_i, k)$
                $\Gamma_k^{bw} \leftarrow EFF(\Gamma_k^{bw}, l_k)$
                if $\Gamma_k^{bw}$ has changed then $E \leftarrow E \cup \{k\}$
        $E \leftarrow E \setminus \{i\}$
    until $E = \emptyset$
    // Join between forward and backward paths //
    for all $i \in \mathcal{V}$
        for all $l_i = (S^i, T^i, C^i, i) \in \Gamma_i^{fw}$
            for all $j \in \mathcal{V}$
                for all $l_j = (S^j, T^j, C^j, j) \in \Gamma_j^{bw}$
                    if $Feasible(l_i, l_j)$ and $HalfWay(l_i, l_j)$
                      then $Save(l_i, l_j)$
    // Search for vertices visited more than once //
    $\Psi \leftarrow MultipleVisits()$
until $\Psi = \emptyset$
```

Tables 1 to 8 report on the experimental comparison between the bi-directional dynamic programming algorithm with bounds [22] (which improves on the results reported in [15]), the state space relaxation algorithm coupled with branch-

and-bound and the decremental state space relaxation algorithm. For the bi-directional dynamic programming algorithm with bounds, named *Exact D.P.* in the tables, we report the total number of non-dominated labels and the computing time. For the branch-and-bound algorithm based on state space relaxation we report the total number of nodes of the search tree, the computing time and the percentage gap between the upper and the lower bounds; the reported results have been obtained with hybrid arcs/resources branching and hybrid cycles/resources branching. For the DSSR algorithm, we report the number of iterations (It), that is the number of times the bi-directional dynamic programming algorithm has been invoked, the number of critical nodes in the last iteration (CN) and the computing time. Empty cells mean that the optimal solution has not been found within the time limit of one hour.

Capacities. Results reported in Tables 1 and 2 show that for 50 vertices instances the DSSR algorithm clearly outperforms all other algorithms on all classes of instances except for the rc-class, where exact bi-directional and bounded dynamic programming is quite fast. However these are very easy instances for all algorithms considered: the computing times are all below one second. The branch-and-bound algorithms sometimes dominate exact dynamic programming but they also fail to terminate within a reasonable computing time or even within the time-out in some cases. For 100 vertices instances the exponential growth of the computing time required by exact dynamic programming becomes evident. DSSR dramatically reduces the computing time up to two orders of magnitude. The branch-and-bound algorithms have performances similar to those of exact dynamic programming and there is no clear domination between the two hybrid branching strategies.

Distribution and collection. When solving the RCESPP with distribution and collection we obtained results similar to those above: they are reported in Tables 3 and 4. The DSSR algorithm solved all instances in less than 340 seconds ouperforming the other algorithms and reducing the computing time by two orders of magnitude. The branch-and-bound is useful only for 100 vertices instances and the results are better for the hybrid branching on cycles and resources.

Capacities and time windows. All Solomon's instances with 50 and 100 nodes were solved by the DSSR algorithm. It should be pointed out that the most difficult instance, the c_104, has been solved within 350 seconds. For the other original Solomon's instances the branch-and-bound algorithms are not competitive, owing to the tightness and the displacement of the time windows, that often allow exact dynamic programming to go faster because the number of feasible solutions is relatively small.

Tightness of the constraints. The last two tables, 7 and 8, show that the difficulty of a RCESPP instance does not depend only on its size but it is strongly affected by the tightness of the constraints. When time windows become larger and larger, the number of non-dominated states increases and so does the computing time. The growth in the number of states and computing time is due to the local nature of the time windows constraints. In these experiments the superiority of algorithms based on state space relaxation is evident. Both branch-and-bound algorithms and the DSSR algorithm solved all instances in a few seconds, whereas exact dynamic programming showed a dramatic exponential growth in computing time. When constraints are very tight, DSSR and branch-and-bound have comparable computational performances.

In spite of its simplicity the idea of decremental state space relaxation is quite effective in practice: the number of critical nodes we could observe was never greater than 15. We remark that our current implementation of the DSSR algorithm does not exploit reoptimization: information computed in the previous run could be used to speed-up successive runs. In this way the computational performances of the algorithm could be further improved.

Last but not least, the implementation of DSSR is by far easier than that of branch-and-bound.

9 Conclusions

We have presented and compared three different methods for the solution of the RCESPP. The first method is exact dynamic programming: though being a well-known method that has been used for nearly two decades, since the seminal work of Desrosiers et al. [12], it can be improved by new ideas, such as bi-directional search with resource-based bounding. The second method is branch-and-bound, where the lower bound is computed by dynamic programming with state space relaxation. We have outlined how bounded bi-directional search can be combined with state space relaxation and we have presented different branching strategies and their hybridization, pointing out that the lower bounding algorithm can easily handle the additional restrictions introduced by branching operations at each node of the branch-and-bound tree. The third method is a new one: decremental state space relaxation. Both exact dynamic programming and state space relaxation are special cases of this new method.

The experimental comparison of the three methods is definitely favourable to decremental state space relaxation, while no clear dominance has been observed between the other methods and not even between different hybrid branching strategies within the branch-and-bound framework. Exact dynamic programming is less robust to the constraints tightness: when the number of non-dominated states grows, the computing time tends to explode very quickly.

Further improvements to the basic DSSR algorithm presented here are possible in at least two directions: first by incorporating re-optimization techniques like those of Desrochers and Soumis [10], so that each iteration of the algorithm does not restart from scratch but can re-use part of the information coming from the previous iteration; second, by guessing a clever initial subset Θ of critical nodes, instead of starting with $\Theta = \emptyset$.

The main motivation of this study is that the RCESPP arises as a pricing subproblem in branch-and-price algorithms for the vehicle routing problem with additional constraints. A natural extension of this research is the comparison between solving the pricing problem to optimality and solving it with state space relaxation or other methods for relaxed pricing. The strategy of solving a relaxation of the pricing subproblem was adopted for instance by Agarwal et al. [1] for solving the CVRP and by Desrochers et al. [8] for solving the VRPTW, while recently Feillet et al. [15] suggested the use of exact pricing for solving the CVRPTW, by proving that tighter lower bounds (and sometimes integer optimal solutions) can be achieved at the root node by column generation with no dramatic increase in computing time. Hence the trade-off between saving computing time and improving the lower bound tightness definitely deserves further investigation and it will be subject of future research. Preliminary experiments on the CVRPTW show that a column generation algorithm in which exact pricing is done via DSSR gives the same lower bounds of Feillet et al. at the root node in only a fraction of the time, in particular for Solomon's instances of classes "c" and "rc". Although we cannot claim that the comparison analyzed in this paper can be directly transferred to the choice between exact pricing and relaxed pricing, we conjecture that the experiments reported here can give useful suggestions about the trade-off between the quality of the lower bound and the computing time required to compute it, depending on the kind of resource constraints and their tightness.

Acknowledgments

We are grateful to Dominique Feillet for providing us his code. We also acknowledge the support of ACSU - Associazione Cremasca Studi Universitari to the Operations Research Laboratory of our department, where this research was done.

References

[1] Agarwal Y., Mathur K., Salkin H.M., *A set-partitioning-based exact algorithm for the vehicle routing problem*, Networks 19 (1989) 731-749

[2] Ahuja R.K., Magnanti T.L., Orlin J.B., *Network flows*, Prentice Hall 1993

[3] Beasley J.E., Christofides N., *An algorithm for the resource constrained shortest path problem*, Networks 19 (1989) 379-394

[4] Bramel J., Simchi-Levi D., *Set-covering-based algorithms for the capacitated VRP*, in *The vehicle routing problem*, Toth P., Vigo D. eds., SIAM Monographs on Discrete Mathematics and Applications, 2002

[5] Christofides N., Mingozzi A., Toth P., *State-space relaxation procedures for the computation of bounds to routing problems* Networks 11, 145-164 (1981).

[6] Cordeau J.F., Desaulniers G., Desrosiers J., Solomon M.M., Soumis F., *VRP with time windows*, in *The vehicle routing problem*, Toth P., Vigo D. eds., SIAM Monographs on Discrete Mathematics and Applications, 2002

[7] Desrochers M., *An algorithm for the shortest path problem with resource constraints*, Cahiers du GERAD G-88-27, University of Montreal (1988)

[8] Desrochers M., Desrosiers J., Solomon M., *A new optimization algorithm for the vehicle routing problem with time windows*, Operations Research 40 (1992) 342-354

[9] Desrochers M., Soumis F., *A generalized permanent labelling algorithm for the shortest path problem with time windows*, INFOR 26, 191-212 (1988)

[10] Desrochers M., Soumis F., *A reoptimization algorithm for the shortest path problem with time windows*, European Journal of Operational Research 35 (1988) 242-254

[11] Desrosiers J., Dumas Y., Solomon M., Soumis F., *Time constrained routing and scheduling* in *Network Routing*, Ball M.O. et al. eds., Handbooks in Operations Research and Management Science, Elsevier Science 1995

[12] Desrosiers J., Pelletier P., Soumis F., *Plus court chemin avec contraintes d'horaires*, RAIRO 17, 357-377 (1983)

[13] Dror M., *Note on the complexity of the shortest path models for column generation in VRPTW*, Operations Research 42 (1994) 977-978

[14] Dumitresu I., Boland N., *Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem*, Networks 42, 135-153 (2003)

[15] Feillet D., Dejax P., Gendreau M., Gueguen C., *An exact algorithm for the elementary shortest path problem with resource constraints: application to some vehicle routing problems*, Networks 44, 216-229 (2004)

[16] Gélinas S., Desrochers M., Desrosiers J., Solomon M.M., *A new branching strategy for time constrained routing problems with application to backhauling* Cahiers du GERAD G-92-13, HEC Montréal, 1992.

[17] Handler G.Y., Zang I., *A dual algorithm for the constrained shortest path problem*, Networks 10 (1980) 293-310

[18] Irnich S., Desaulniers G., *Shortest path problems with resource constraints*, Cahier du GERAD G-2004-11, Université de Montréal, 2004

[19] Irnich S., Villeneuve D., *The shortest path problem wiht resource constraints and k-cycle elimination for* $k \geq 3$, Cahiers du GERAD G-2003-55, HEC Montréal, 2003

[20] Kohl N., Desrosiers J., Madsen O.B.G., Solomon M.M., Soumis F., *2-path cuts for the vehicle routing problem with time windows* Transportation Science 33 (1999) 101-116

[21] Mingozzi A., Bianco L., Ricciardelli S., *Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints*, Operations Research 45 (1997) 365-377

[22] Righini G., Salani M., *Symmetry helps: bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints*, Note del Polo - Ricerca 66, DTI - Università degli Studi di Milano, 2004 (submitted for publication)

Table 1: RCESPP with capacity - 50 vertices

Instance	Exact D.P.		B&B Res+Arcs			B&B Res+Cycles			DSSR		
Name	Labels	Time	Nodes	Time	(%)	Nodes	Time	(%)	It	CN	Time
c_50_01	56	0.00	1	0.00	-	1	0.00	-	1	0	0.00
c_50_02	268	0.00	1	0.00	-	1	0.00	-	1	0	0.00
c_50_03	692	0.00	7	0.01	-	1	0.02	-	1	0	0.00
c_50_04	2574	0.03	19	0.08	-	10	0.05	-	3	2	0.02
c_50_05	4692	0.07	11	0.05	-	22	0.09	-	2	2	0.01
c_50_06	15236	0.91	87	0.48	-	116	0.91	-	3	3	0.04
c_50_07	23394	1.75	35	0.24	-	52	0.33	-	2	3	0.02
c_50_08	75026	20.35	315	3.19	-	124	1.81	-	5	7	0.25
c_50_09	101128	33.24	673	5.80	-	224	1.78	-	4	8	0.18
c_50_10	331402	394.97	3919	44.56	-	3039	53.34	-	5	10	1.82
r_50_01	62	0.00	1	0.00	-	1	0.00	-	1	0	0.00
r_50_02	210	0.01	1	0.01	-	1	0.00	-	1	0	0.00
r_50_03	525	0.01	7	0.04	-	1	0.00	-	3	3	0.02
r_50_04	1250	0.02	7	0.06	-	1	0.00	-	2	3	0.02
r_50_05	2418	0.05	591	6.64	-	7	0.09	-	6	7	0.17
r_50_06	4570	0.11	271	4.49	-	83	1.18	-	4	6	0.12
r_50_07	7874	0.24	6009	90.34	-	122	1.12	-	2	4	0.06
r_50_08	13590	0.60	3149	38.75	-	95	0.98	-	5	11	0.48
r_50_09	22800	1.49	191	6.12	-	17	0.53	-	4	9	0.40
r_50_10	36838	3.87	59	1.16	-	7	0.26	-	3	8	0.34
rc_50_01	44	0.00	1	0.00	-	1	0.00	-	1	0	0.00
rc_50_02	124	0.00	1	0.00	-	1	0.00	-	1	0	0.00
rc_50_03	268	0.00	11	0.02	-	1	0.01	-	4	3	0.00
rc_50_04	560	0.01	725	1.58	-	73	0.17	-	5	4	0.02
rc_50_05	800	0.01	1573	4.21	-	173	0.42	-	4	4	0.02
rc_50_06	1551	0.03	73573	562.40	-	1774	8.29	-	7	8	0.09
rc_50_07	1774	0.04	3417	19.66	-	84	0.50	-	4	7	0.05
rc_50_08	3217	0.08	239467	-	6.25	10505	49.19	-	7	11	0.20
rc_50_09	3322	0.09	29021	348.97	-	960	7.68	-	6	11	0.20
rc_50_10	5864	0.19	254931	-	2.0	16679	156.19	-	6	11	0.27

Table 2: RCESPP with capacity - 100 vertices

Instance	Exact D.P.		B&B Res+Arcs			B&B Res+Cycles			DSSR		
Name	Labels	Time	Nodes	Time	(%)	Nodes	Time	(%)	It	CN	Time
c_100_01	106	0.00	1	0.00	-	1	0.00	-	1	0	0.00
c_100_02	559	0.01	1	0.01	-	1	0.01	-	1	0	0.00
c_100_03	1456	0.03	7	0.04	-	1	0.02	-	2	1	0.03
c_100_04	5900	0.19	19	0.26	-	10	0.15	-	3	2	0.06
c_100_05	11546	0.44	43	0.36	-	22	0.31	-	3	4	0.07
c_100_06	45138	6.60	193	5.19	-	699	16.31	-	4	5	0.21
c_100_07	75698	13.78	65	1.84	-	99	2.65	-	3	6	0.18
c_100_08	310651	276.88	1275	77.50	-	517	30.53	-	6	10	1.34
c_100_09	4799333	520.82	8125	326.11	-	862	32.71	-	6	14	2.02
c_100_10	-	-	11465	627.08	-	12076	910.73	-	6	13	7.68
r_100_01	266	0.00	15	0.04	-	10	0.04	-	3	3	0.02
r_100_02	2120	0.06	459	3.61	-	225	1.45	-	2	4	0.06
r_100_03	11866	0.70	5337	126.87	-	776	15.03	-	4	5	0.59
r_100_04	53668	8.37	7639	306.49	-	1819	69.68	-	4	7	2.80
r_100_05	215976	104.41	81677	-	7.3	4464	198.93	-	3	5	2.87
r_100_06	764476	1300.33	51755	-	9.5	13209	1282.61	-	4	8	34.64
r_100_07	-	-	31121	-	11.2	29182	-	1.1	5	10	143.63
r_100_08	-	-	12548	-	25.4	18741	-	6.3	5	11	281.62
r_100_09	-	-	6912	-	51.1	12549	-	18.9	3	10	303.34
r_100_10	-	-	2551	-	67.4	6118	-	43.2	3	10	319.68
rc_100_01	90	0.00	1	0.00	-	1	0.00	-	1	0	0.00
rc_100_02	636	0.01	1	0.01	-	1	0.00	-	1	0	0.00
rc_100_03	1732	0.04	31	0.33	-	10	0.13	-	1	0	0.00
rc_100_04	5706	0.23	7	0.20	-	10	0.27	-	2	1	0.07
rc_100_05	12561	0.69	1669	71.45	-	64	2.35	-	4	4	0.29
rc_100_06	29786	2.80	71	2.52	-	60	3.01	-	3	4	0.35
rc_100_07	60499	9.74	4403	376.85	-	752	59.22	-	4	5	0.92
rc_100_08	124752	37.46	5735	353.00	-	254	25.00	-	4	7	1.77
rc_100_09	237652	130.20	739	109.08	-	391	28.76	-	3	5	1.40
rc_100_10	459269	470.24	25055	-	3.7	7385	1191.96	-	5	10	7.33

Table 3: RCESPP with distribution and collection - 50 vertices

Instance	Exact D.P.		B&B Res+Arcs			B&B Res+Cycles			DSSR		
Name	Labels	Time	Nodes	Time	(%)	Nodes	Time	(%)	It	CN	Time
c_50_01	26	0.00	1	0.00	-	1	0.00	-	1	0	0.00
c_50_02	159	0.00	1	0.00	-	1	0.00	-	1	0	0.00
c_50_03	554	0.01	1	0.02	-	1	0.01	-	2	1	0.00
c_50_04	1751	0.02	7	0.02	-	10	0.04	-	2	1	0.01
c_50_05	4675	0.07	15	0.07	-	22	0.10	-	2	2	0.02
c_50_06	11311	0.41	27	0.17	-	122	0.94	-	3	3	0.05
c_50_07	24006	1.72	17	0.16	-	53	0.48	-	2	3	0.03
c_50_08	51401	7.95	83	0.90	-	91	1.44	-	5	7	0.47
c_50_09	110354	35.46	69	0.88	-	207	2.47	-	3	5	0.19
c_50_10	233478	165.21	119	3.56	-	129	3.66	-	5	10	2.20
r_50_01	59	0.00	1	0.00	-	1	0.00	-	1	0	0.00
r_50_02	188	0.00	5	0.02	-	7	0.02	-	2	1	0.01
r_50_03	486	0.01	1	0.01	-	1	0.01	-	3	3	0.01
r_50_04	1113	0.02	1	0.03	-	1	0.03	-	2	3	0.02
r_50_05	2085	0.04	11	0.14	-	7	0.08	-	5	6	0.13
r_50_06	3882	0.10	17	0.28	-	45	0.62	-	3	5	0.07
r_50_07	6986	0.23	3	0.07	-	3	0.08	-	2	4	0.06
r_50_08	12138	0.51	71	0.97	-	122	1.13	-	4	7	0.19
r_50_09	20384	1.23	13	0.42	-	19	0.59	-	3	7	0.21
r_50_10	33107	3.15	5	0.21	-	5	0.21	-	3	7	0.25
rc_50_01	24	0.00	1	0.00	-	1	0.00	-	1	0	0.00
rc_50_02	83	0.00	1	0.00	-	1	0.00	-	1	0	0.01
rc_50_03	199	0.01	1	0.01	-	1	0.01	-	3	2	0.01
rc_50_04	397	0.01	55	0.12	-	73	0.14	-	5	4	0.02
rc_50_05	764	0.02	171	0.51	-	114	0.28	-	4	5	0.03
rc_50_06	1108	0.02	3321	19.86	-	1156	5.01	-	6	7	0.09
rc_50_07	1817	0.03	397	3.00	-	233	1.40	-	6	7	0.09
rc_50_08	2546	0.05	595	6.41	-	405	4.07	-	4	7	0.05
rc_50_09	3435	0.10	4363	63.94	-	666	8.13	-	7	9	0.22
rc_50_10	4998	0.15	12045	210.69	-	6551	79.95	-	6	11	0.24

Table 4: RCESPP with distribution and collection - 100 vertices

Instance	Exact D.P.		B&B Res+Arcs			B&B Res+Cycles			DSSR		
Name	Labels	Time	Nodes	Time	(%)	Nodes	Time	(%)	It	CN	Time
c_100_01	48	0.00	1	0.00	-	1	0.00	-	1	0	0.00
c_100_02	328	0.00	1	0.00	-	1	0.00	-	1	0	0.02
c_100_03	1179	0.02	1	0.00	-	1	0.00	-	2	1	0.02
c_100_04	3829	0.08	7	0.10	-	10	0.13	-	3	2	0.06
c_100_05	11112	0.41	15	0.26	-	22	0.40	-	3	4	0.10
c_100_06	30823	2.62	15	0.54	-	597	17.26	-	4	5	0.25
c_100_07	76548	12.72	35	1.24	-	101	3.67	-	3	6	0.27
c_100_08	197386	87.88	175	6.56	-	192	12.57	-	6	10	2.17
c_100_09	509042	516.42	239	15.83	-	884	58.01	-	5	11	2.34
c_100_10	-	-	737	89.37	-	952	11.27	-	7	15	20.64
r_100_01	253	0.00	1	0.00	-	1	0.00	-	1	0	0.00
r_100_02	1948	0.06	801	7.47	-	222	1.52	-	2	4	0.06
r_100_03	10874	0.68	7275	186.84	-	831	16.90	-	4	5	0.64
r_100_04	49258	7.34	15297	790.61	-	1814	69.59	-	3	5	1.34
r_100_05	189041	84.00	40991	3503.40	-	5242	357.35	-	3	5	3.71
r_100_06	676338	1040.25	42932	-	5.0	14627	1074.52	-	4	8	39.63
r_100_07	-	-	36124	-	8.9	24782	-	1.4	5	10	180.41
r_100_08	-	-	19733	-	22.3	30764	-	12.5	4	10	217.66
r_100_09	-	-	8153	-	45.5	11489	-	22.0	3	10	337.47
r_100_10	-	-	3559	-	62.8	4025	-	61.0	3	10	337.43
rc_100_01	67	0.00	1	0.00	-	1	0.00	-	1	0	0.00
rc_100_02	501	0.01	1	0.00	-	1	0.00	-	1	0	0.00
rc_100_03	1422	0.04	7	0.10	-	10	0.13	-	2	1	0.03
rc_100_04	4540	0.17	7	0.19	-	10	0.26	-	2	1	0.07
rc_100_05	10790	0.55	27	1.16	-	61	2.25	-	4	4	0.30
rc_100_06	25657	2.29	23	1.18	-	54	2.98	-	3	4	0.33
rc_100_07	52378	7.73	801	74.78	-	736	58.96	-	4	5	0.97
rc_100_08	107414	28.62	361	44.60	-	242	24.13	-	3	5	1.11
rc_100_09	207049	99.56	235	23.30	-	389	29.55	-	3	5	1.55
rc_100_10	-	-	5671	971.28	-	7162	1207.45	-	4	8	6.11

Table 5: RCESPP with capacity and time windows - 50 vertices

Instance	Exact D.P.		B&B Res+Arcs			B&B Res+Cycles			DSSR		
Name	Labels	Time	Nodes	Time	(%)	Nodes	Time	(%)	It	CN	Time
c101_50	323	0.01	1	0.00	-	1	0.00	-	1	0	0.00
c102_50	3026	0.14	15	0.19	-	12	0.19	-	2	2	0.12
c103_50	30736	10.25	2533	127.32	-	966	27.74	-	4	6	14.06
c104_50	-	-	35781	-	1.2	24785	1835.47	-	4	11	344.65
c105_50	435	0.02	1	0.02	-	1	0.02	-	1	0	0.01
c106_50	359	0.01	1	0.03	-	1	0.03	-	1	0	0.02
c107_50	504	0.02	1	0.04	-	1	0.04	-	1	0	0.02
c108_50	736	0.04	1	0.04	-	1	0.04	-	1	0	0.04
c109_50	2031	0.15	81	1.90	-	78	1.69	-	8	11	1.35
r101_50	121	0.00	1	0.00	-	1	0.00	-	1	0	0.00
r102_50	596	0.02	5	0.05	-	4	0.04	-	4	5	0.08
r103_50	2322	0.06	817	9.83	-	103	1.07	-	3	4	0.27
r104_50	15441	0.66	83	2.46	-	264	7.08	-	3	5	0.77
r105_50	238	0.01	1	0.01	-	1	0.01	-	1	0	0.0
r106_50	818	0.02	13	0.14	-	9	0.08	-	4	6	0.18
r107_50	2784	0.08	15435	248.92	-	268	3.10	-	4	5	0.47
r108_50	16457	0.75	5	0.15	-	5	0.15	-	2	4	0.48
r109_50	584	0.02	1	0.02	-	1	0.02	-	1	0	0.02
r110_50	1600	0.04	1	0.04	-	1	0.04	-	1	0	0.04
r111_50	2289	0.07	43	0.73	-	46	0.66	-	3	5	0.33
r112_50	3987	0.12	1	0.05	-	1	0.05	-	1	0	0.05
rc101_50	270	0.00	1	0.00	-	1	0.00	-	1	0	0.01
rc102_50	906	0.01	17	0.13	-	17	0.13	-	3	3	0.09
rc103_50	3509	0.07	112	3.45	-	6168	90.88	-	5	11	0.95
rc104_50	8801	0.28	258	7.65	-	312	8.31	-	7	15	4.77
rc105_50	937	0.01	63	0.48	-	60	0.32	-	3	4	0.11
rc106_50	889	0.01	71	0.52	-	676	4.64	-	4	7	0.16
rc107_50	3525	0.07	29567	653.53	-	13452	195.907	-	6	9	0.83
rc108_50	10166	0.21	34205	1143.65	-	58476	1474.19	-	6	13	2.12

Table 6: RCESPP with capacity and time windows - 100 vertices

Instance	Exact D.P.		B&B Res+Arcs			B&B Res+Cycles			DSSR		
Name	Labels	Time	Nodes	Time	(%)	Nodes	Time	(%)	It	CN	Time
c101_100	679	0.06	1	0.02	-	1	0.02	-	1	0	0.02
c102_100	11839	2.99	15	0.89	-	12	0.67	-	2	2	0.63
c103_100	123804	133.56	2539	507.78	-	3075	485.68	-	5	9	40.15
c104_100	-	-	17553	-	17.8	23402	-	16.5	4	11	311.44
c105_100	915	0.11	1	0.06	-	1	0.06	-	1	0	0.06
c106_100	1159	0.16	1	0.07	-	1	0.07	-	1	0	0.07
c107_100	1058	0.16	1	0.08	-	1	0.08	-	1	0	0.08
c108_100	1690	0.33	1	0.17	-	1	0.17	-	1	0	0.17
c109_100	4608	1.28	111	14.18	-	101	12.66	-	8	13	9.19
r101_100	452	0.01	1	0.00	-	1	0.00	-	1	0	0.00
r102_100	14792	2.21	4203	438.82	-	1003	96.25	-	3	6	21.69
r103_100	135575	95.73	377	128.71	-	252	61.10	-	4	7	159.74
r104_100	655858	1242.56	4531	-	0.9	1945	1013.04	-	3	5	78.32
r105_100	1161	0.06	1	0.03	-	1	0.03	-	1	0	0.03
r106_100	22970	5.52	26059	-	3.4	3344	467.29	-	4	7	71.60
r107_100	138027	100.83	1417	717.37	-	732	232.91	-	4	12	335.98
r108_100	570910	891.81	1593	1098.05	-	1451	611.61	-	3	5	146.58
r109_100	3504	0.37	49	3.39	-	49	3.45	-	3	5	2.93
r110_100	25063	4.89	307	69.25	-	406	77.55	-	3	6	19.31
r111_100	69890	25.86	2669	866.34	-	361	129.24	-	3	7	53.16
r112_100	394702	647.36	2167	2227.74	-	665	385.00	-	3	9	340.61
rc101_100	955	0.03	1	0.02	-	1	0.02	-	1	0	0.02
rc102_100	5384	0.30	17	1.12	-	21	1.34	-	4	4	3.17
rc103_100	38308	6.01	861	110.78	-	6494	587.39	-	4	8	35.37
rc104_100	232961	148.73	607	194.68	-	1054	203.36	-	4	8	102.56
rc105_100	2964	0.15	7	0.43	-	13	0.67	-	4	6	1.14
rc106_100	2574	0.12	31	1.85	-	12	0.56	-	3	3	1.01
rc107_100	10505	0.72	87	8.50	-	27	2.79	-	4	6	3.65
rc108_100	45430	6.75	239	32.34	-	143	12.43	-	3	5	4.84

Table 7: RCESPP with capacity and time windows - Instance c_104, 50 vertices

Instance	Exact D.P.		B&B Res+Arcs			B&B Res+Cycles			DSSR		
Name	Labels	Time	Nodes	Time	(%)	Nodes	Time	(%)	It	CN	Time
c104_50_01	59	0.00	1	0.01	-	1	0.01	-	1	0	0.01
c104_50_02	126	0.00	1	0.01	-	1	0.01	-	1	0	0.01
c104_50_03	200	0.01	1	0.01	-	1	0.01	-	1	0	0.01
c104_50_04	206	0.01	1	0.01	-	1	0.01	-	1	0	0.01
c104_50_05	208	0.01	1	0.01	-	1	0.01	-	1	0	0.01
c104_50_06	269	0.01	1	0.02	-	1	0.02	-	1	0	0.02
c104_50_07	475	0.01	1	0.02	-	1	0.02	-	1	0	0.02
c104_50_08	586	0.04	1	0.02	-	1	0.02	-	1	0	0.02
c104_50_09	730	0.05	1	0.02	-	1	0.02	-	1	0	0.02
c104_50_10	804	0.06	1	0.02	-	1	0.02	-	1	0	0.02
c104_50_11	1440	0.09	1	0.03	-	1	0.03	-	1	0	0.03
c104_50_12	2292	0.21	1	0.03	-	1	0.03	-	1	0	0.03
c104_50_13	3771	0.43	1	0.04	-	1	0.04	-	1	0	0.04
c104_50_14	4031	0.53	19	0.24	-	13	0.18	-	2	3	0.14
c104_50_15	5508	0.74	13	0.22	-	9	0.12	-	3	4	0.49
c104_50_16	9411	2.10	39	0.88	-	12	0.25	-	:3	4	0.53
c104_50_17	18579	5.02	45	1.09	-	15	0.35	-	3	4	0.60
c104_50_18	21738	6.54	145	3.53	-	76	1.66	-	3	4	0.61
c104_50_19	25638	9.25	77	2.43	-	64	1.41	-	3	5	1.20
c104_50_20	36762	23.16	85	2.02	-	91	2.35	-	4	6	2.11
c104_50_21	81804	73.19	179	5.95	-	144	3.99	-	4	6	2.69
c104_50_22	105756	110.25	199	5.35	-	91	2.52	-	3	6	2.31
c104_50_23	126645	149.81	255	11.00	-	214	9.34	-	8	11	17.45
c104_50_24	157915	275.93	276	19.7	-	238	17.98	-	5	7	13.43
c104_50_25	323641	1016.98	321	18.36	-	381	19.83	-	4	7	13.31

Table 8: RCESPP with capacity and time windows - Instance c_104, 100 vertices

Instance	Exact D.P.		B&B Res+Arcs			B&B Res+Cycles			DSSR		
Name	Labels	Time	Nodes	Time	(%)	Nodes	Time	(%)	It	CN	Time
c104_100_01	160	0.01	1	0.04	-	1	0.04	-	1	0	0.04
c104_100_02	227	0.01	1	0.05	-	1	0.05	-	1	0	0.05
c104_100_03	368	0.02	1	0.06	-	1	0.06	-	1	0	0.06
c104_100_04	415	0.04	1	0.07	-	1	0.07	-	1	0	0.07
c104_100_05	427	0.05	1	0.07	-	1	0.07	-	1	0	0.07
c104_100_06	523	0.06	1	0.08	-	1	0.08	-	1	0	0.08
c104_100_07	895	0.11	1	0.08	-	1	0.08	-	1	0	0.08
c104_100_08	1153	0.19	1	0.09	-	1	0.09	-	1	0	0.09
c104_100_09	1393	0.29	1	0.09	-	1	0.09	-	1	0	0.09
c104_100_10	1557	0.37	1	0.09	-	1	0.09	-	1	0	0.09
c104_100_11	2655	0.59	1	0.11	-	1	0.11	-	1	0	0.11
c104_100_12	4504	1.24	1	0.12	-	1	0.12	-	1	0	0.12
c104_100_13	7336	2.37	1	0.11	-	1	0.11	-	1	0	0.11
c104_100_14	8457	3.30	33	1.76	-	11	0.74	-	3	3	1.49
c104_100_15	10932	4.49	65	3.24	-	21	1.05	-	2	3	1.26
c104_100_16	19133	9.75	249	15.12	-	23	1.62	-	2	3	1.56
c104_100_17	39114	23.83	211	20.02	-	21	1.60	-	2	3	1.65
c104_100_18	53650	39.11	75	8.64	-	30	3.54	-	2	3	1.65
c104_100_19	66165	56.43	45	5.85	-	30	2.99	-	4	6	6.32
c104_100_20	91825	110.71	59	4.66	-	42	2.04	-	4	5	7.84
c104_100_21	197498	336.64	67	7.29	-	48	4.26	-	4	5	8.79
c104_100_22	315475	702.11	89	10.78	-	59	5.94	-	3	6	7.54
c104_100_23	437113	1169.71	53	7.94	-	53	7.60	-	7	9	32.06
c104_100_24	547902	1945.73	141	31.02	-	143	33.46	-	5	7	29.87
c104_100_25	-	-	187	47.05	-	17	44.32	-	4	7	27.74