New Dynamic Programming Algorithms for the Resource Constrained Elementary Shortest Path Problem

Giovanni Righini and Matteo Salani
Dipartimento di Tecnologie dell'Informazione, Università degli Studi di Milano, Via Bramante 65, 26013 Crema, Italy

The resource constrained elementary shortest path problem (RCESPP) arises as a pricing subproblem in branch-and-price algorithms for vehicle-routing problems with additional constraints. We address the optimization of the RCESPP and we present and compare three methods. The first method is a well-known exact dynamic-programming algorithm improved by new ideas, such as bidirectional search with resource-based bounding. The second method consists in a branch-and-bound algorithm, where lower bounds are computed by dynamic-programming with state-space relaxation; we show how bounded bidirectional search can be adapted to 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 favorable to decrement 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. © 2007 Wiley Periodicals, Inc. NETWORKS, Vol. 51(3), 155–170 2008

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

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. [21]). 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 recommend the reader to refer Desrosiers et al. [12], Bramel and Simchi-Levi [5], and Cordeau et al. [7].

If the underlying graph may have negative cost cycles (this 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 [14]. The most commonly used technique to solve the RCESPP to optimality is dynamic-programming, relying upon the seminal work by Desrochers and Soumis [9] for the resource constrained shortest path problem (RCSPP) in which the solution is not required to be elementary. Methods based on Lagrangean relaxation were proposed by Handler and Zang [18] and Beasley and Christofides [3], and were recently examined by Dumitrescu and Boland [15], who proposed 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. For a recent survey on models and algorithms for the RCSPP and the RCESPP, we recommend the reader to refer Irnich and Desaulniers [19].

In this paper, we consider the problem of computing optimal solutions to the RCESPP, as in Feillet et al. [16], 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. [16] and Righini and Salani [23]. In the second case, a dynamic-programming algorithm with

Received January 2005; accepted May 2007
Correspondence to: M. Salani; e-mail: matteo.salani@epfl.ch
Contract grant sponsor: ACSU
DOI 10.1002/net.20212
Published online 14 December 2007 in Wiley InterScience (www.interscience.wiley.com).

state-space relaxation is used to optimize the RCSPP, where cycles are allowed. This gives lower bounds corresponding to nonelementary 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 bidirectional search and resource-based bounding, as shown by Righini and Salani [23]. In this paper, we review them and we show that they can be also applied 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 gives as follows: 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 bidirectional 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 [23], which have been recalled here to make this paper self-contained.

The idea of decremental state-space relaxation presented in Section 7 was also independently developed by Boland et al. [4] with the name of "Space Augmenting Algorithm." They made computational tests on randomly generated instances of a generic resource constrained shortest path problem with one resource and mainly on noncomplete graphs. In Section 8, we present computational results on three different types of resource constrained shortest path problems, coming from VRP applications, with different resources, using instances taken from the VRP literature.

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 $i \in \mathcal{N}$, and 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 associated 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.

2.1. Capacity

In the CVRP (see Toth and Vigo [24]), a positive 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.

2.2. Distribution and Collection

In the VRPDC (see Dell'Amico et al. [8]), each vertex i has two positive integer quantities p_i and d_i associated with it, representing the amount of load to be collected and to be delivered at that vertex respectively. Each vehicle has a positive integer capacity Q, and 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.

2.3. Capacity and Time Windows

In the CVRPTW (see Cordeau et al. [7]), a positive integer demand d_i, 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 positive 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 label-setting algorithm of Desrochers and Soumis [9] for the RCSPP, that is, an extension of the well-known shortest path algorithm of Ford and Bellman (see [2]). The algorithm of Desrochers and Soumis 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 corresponding vertex is visited. The consumption of the N dummy resources is indicated by a vector S initialized at 0. Note that the vector 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.

3.1. Cost

The cost is initialized 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 $j = t$.

3.2. 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.

3.3. 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 $q \leq Q$.

3.4. 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, respectively. The first resource at vertex i is the amount of load that the vehicle can pick-up after visiting vertex i; the second resource at vertex i indicates the amount of load that the vehicle can deliver after visiting vertex i. The consumption of the first resource, indicated by π, increases after every pick-up operation, because visiting vertex i the vehicle consumes p_i units of capacity. Hence π is initialized at 0 and is not allowed to exceed the capacity Q. The maximum amount the vehicle can deliver after visiting vertex i is equal to the minimum residual capacity that the vehicle has had since its departure from the depot s up to vertex i. Hence its consumption δ is equal to the maximum amount of load that has been on board up to the visit at vertex i. For this reason, the maximum amount the vehicle can deliver after visiting vertex i cannot be greater than the maximum amount it can pick-up after visiting vertex i. Hence both consumptions 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.

3.5. Capacity and Time Windows

In this case, the time elapsed is a consumed resource and the consumption monotonically increases 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 nondominated 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. [16]. 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 unreachable 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. [16] 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. We consider label-correcting algorithms like those of Desrosiers et al. [13] and Feillet et al. [16]. 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 been proposed (see, for instance, Desrochers and Soumis [9]), but they require an hypothesis stronger than resource consumption monotonicity: in particular, there must exist 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. [12].

4. BOUNDED BIDIRECTIONAL DYNAMIC-PROGRAMMING

We recall here the main concepts of bounded bidirectional dynamic-programming that has been recently introduced by Righini and Salani [23] to improve the exact dynamic-programming algorithm described earlier. In bidirectional dynamic-programming, states are extended both forward from vertex s to its successors and backward from vertex t to its predecessors (see for instance [22] for an application of this idea to the constrained TSP). With each vertex $i \in \mathcal{V}$, we associate forward states indicated by $(S^{\mathrm{fw}}, R^{\mathrm{fw}}, C^{\mathrm{fw}}, i)$ and backward states indicated by $(S^{\mathrm{bw}}, R^{\mathrm{bw}}, C^{\mathrm{bw}}, i)$. A path from s to t is detected each time, and a forward state $(S^{\mathrm{fw}}, R^{\mathrm{fw}}, C^{\mathrm{fw}}, i)$ and a backward state $(S^{\mathrm{bw}}, R^{\mathrm{bw}}, C^{\mathrm{bw}}, j)$ 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 $(S^{\mathrm{bw}}, R^{\mathrm{bw}}, C^{\mathrm{bw}}, j)$ is extended to a backward state $(S'^{\mathrm{bw}}, R'^{\mathrm{bw}}, C'^{\mathrm{bw}}, i)$ the cost is updated according to the formula:

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

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

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

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

4.1.1. 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 $(S^{\mathrm{bw}}, q^{\mathrm{bw}}, C^{\mathrm{bw}}, j)$ to a state $(S'^{\mathrm{bw}}, q'^{\mathrm{bw}}, C'^{\mathrm{bw}}, i)$, the extension rule is

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

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

4.1.2. 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 $(S^{\mathrm{bw}}, \pi^{\mathrm{bw}}, \delta^{\mathrm{bw}}, C^{\mathrm{bw}}, j)$ to a state $(S'^{\mathrm{bw}}, \pi'^{\mathrm{bw}}, \delta'^{\mathrm{bw}}, C'^{\mathrm{bw}}, i)$, the extension rule is

$$\pi'^{\mathrm{bw}} = \max\{\delta^{\mathrm{bw}} + d_i, \pi^{\mathrm{bw}} + p_i\}$$
$$\delta'^{\mathrm{bw}} = \delta^{\mathrm{bw}} + d_i.$$

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

4.1.3. 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 $[a_i^{\mathrm{fw}}, b_i^{\mathrm{fw}}]$ and $[a_i^{\mathrm{bw}}, b_i^{\mathrm{bw}}]$ as follows:

$$a_i^{\mathrm{fw}} = a_i$$
$$b_i^{\mathrm{fw}} = b_i$$
$$a_i^{\mathrm{bw}} = a_i + \theta_i$$
$$b_i^{\mathrm{bw}} = b_i + \theta_i.$$

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, $T = \max_{i \in \mathcal{N} \cup \{s\}}\{b_i^{\mathrm{fw}} + \theta_i + v_{it}\}$. 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 not later than at time T. When a feasible backward path is extended along arc (i,j) from a state $(S^{\mathrm{bw}}, q^{\mathrm{bw}}, \tau^{\mathrm{bw}}, C^{\mathrm{bw}}, j)$ to a state $(S'^{\mathrm{bw}}, q'^{\mathrm{bw}}, \tau'^{\mathrm{bw}}, C'^{\mathrm{bw}}, i)$, the extension rules are:

$$q'^{\mathrm{bw}} = q^{\mathrm{bw}} + d_i$$
$$\tau'^{\mathrm{bw}} = \max\left\{\tau^{\mathrm{bw}} + \theta_j + v_{ij}, T - b_i^{\mathrm{bw}}\right\}.$$

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

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 $(S^{\mathrm{fw}}, q^{\mathrm{fw}}, C^{\mathrm{fw}}, i)$ is joined with a backward path $(S^{\mathrm{bw}}, q^{\mathrm{bw}}, C^{\mathrm{bw}}, j)$ the cost of the resulting *s-t* path is equal to

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

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 cannot be visited by both paths.

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

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.

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

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

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

$$\pi^{\mathrm{fw}} + \pi^{\mathrm{bw}} \leq Q$$
$$\delta^{\mathrm{fw}} + \delta^{\mathrm{bw}} \leq Q.$$

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

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

4.3. Resource-Based Bounding

Since all forward and backward states generated by the bidirectional search algorithm are tentatively joined, it is crucial to reduce their number as much as possible. To this purpose, we select a *critical resource* $(\hat{r})$, 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 $(\hat{R})$ 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.

4.3.1. 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 $Q/2$.

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

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

The combination of bidirectional search with resource-based bounding allows to solve larger instances (or the same instances in less time) than monodirectional dynamic-programming; detailed experimental results are reported in [23]. In the next section, we show how bidirectional search and resource-based bounding can be incorporated into dynamic-programming algorithms based on state-space relaxation.

Algorithm 1. RCESPP—Bidirectional dynamic programming

// Initialization //
$\Gamma_s^{\text{fw}} \leftarrow \{(\mathbf{0}, \mathbf{0}, 0, s)\}$
$\Gamma_t^{\text{bw}} \leftarrow \{(\mathbf{0}, \mathbf{0}, 0, t)\}$
for all $i \in \mathcal{V} \setminus \{s\}$ **do** $\Gamma_i^{\text{fw}} \leftarrow \emptyset$
for all $i \in \mathcal{V} \setminus \{t\}$ **do** $\Gamma_i^{\text{bw}} \leftarrow \emptyset$
$E \leftarrow \{s, t\}$
// Search //
repeat
  // Vertex selection //
  Select $i \in E$
  // Forward extension //
  for all $l_i = (S^i, R^i, C^i, i) \in \Gamma_i^{\text{fw}}$ **do**
    if $\hat{r}^i < \hat{R}/2$ **then**
      for all $j \in \Delta_i^+$ such that $S_j^i = 0$ **do**
        $l_j \leftarrow Extend^{\text{fw}}(l_i, j)$
        $\Gamma_j^{\text{fw}} \leftarrow EFF(\Gamma_j^{\text{fw}}, l_j)$
        if Γ_j^{fw} has changed **then** $E \leftarrow E \cup \{j\}$
  // Backward extension //
  for all $l_i = (S^i, R^i, C^i, i) \in \Gamma_i^{\text{bw}}$ **do**
    if $\hat{r}^i < \hat{R}/2$ **then**
      for all $k \in \Delta_i^-$ such that $S_k^i = 0$ **do**
        $l_k \leftarrow Extend^{\text{bw}}(l_i, k)$
        $\Gamma_k^{\text{bw}} \leftarrow EFF(\Gamma_k^{\text{bw}}, l_k)$
        if Γ_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 $l_i = (S^i, R^i, C^i, i) \in \Gamma_i^{\text{fw}}$
    for all $j \in \mathcal{V}$
      for all $l_j = (S^j, R^j, C^j, j) \in \Gamma_j^{\text{bw}}$
        if $Feasible(l_i, l_j)$ **then** $Save(l_i, l_j)$

In Algorithm 1, we report the pseudocode of the bounded bidirectional algorithm where 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; $Extend^{\text{fw}}(l, k)$ and $Extend^{\text{bw}}(l, k)$ are the forward and backward extension procedures, respectively they extend the state specified as a first argument to a vertex specified as a second argument; these procedures check the resource constraints and produce only feasible states; finally $EFF(\Gamma, l)$ is the procedure that inserts state l into set Γ applying the domination rules; $Feasible(l_i, l_j)$ checks the resource compatibility of states l_i and l_j according to problem-dependent rules; $Save(l_i, l_j)$ saves the solution obtained from the two states l_i and l_j.

5. STATE SPACE RELAXATION

State space relaxation was introduced by Christofides et al. [6] 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. [10]): 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). In our algorithm, we also count unreachable vertices as if they were already visited, exploiting the technique of Feillet et al. [16]. 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 pseudopolynomial 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 at 0 and 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.

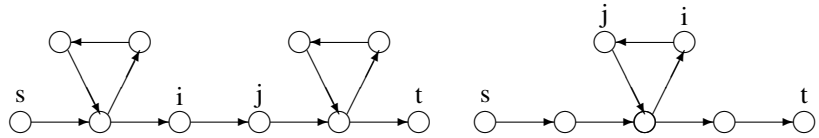

FIG. 1. On the left: an s-t path made of nonelementary paths s-i and j-t. On the right: a nonelementary s-t path made of elementary paths s-i and j-t.

Algorithm 2. RCSPP—Bidirectional state space relaxation

// Initialization //
$\Gamma_s^{\mathrm{fw}} \leftarrow \{(0, \mathbf{0}, 0, s)\}$
$\Gamma_t^{\mathrm{bw}} \leftarrow \{(0, \mathbf{0}, 0, t)\}$
for all $i \in \mathcal{V} \setminus \{s\}$ **do** $\Gamma_i^{\mathrm{fw}} \leftarrow \emptyset$
for all $i \in \mathcal{V} \setminus \{t\}$ **do** $\Gamma_i^{\mathrm{bw}} \leftarrow \emptyset$
$E \leftarrow \{s, t\}$
// Search //
repeat
  // Vertex selection //
  Select $i \in E$
  // Forward extension //
  for all $l_i = (\sigma^i, R^i, C^i, i) \in \Gamma_i^{\mathrm{fw}}$ **do**
    if $\hat{r}^i < \hat{R}/2$ **then**
      for all $j \in \Delta_i^+$ **do**
        $l_j \leftarrow Extend^{\mathrm{fw}}(l_i, j)$
        $\Gamma_j^{\mathrm{fw}} \leftarrow EFF(\Gamma_j^{\mathrm{fw}}, l_j)$
        if Γ_j^{fw} has changed **then** $E \leftarrow E \cup \{j\}$
  // Backward extension //
  for all $l_i = (\sigma^i, R^i, C^i, i) \in \Gamma_i^{\mathrm{bw}}$ **do**
    if $\hat{r}^i < \hat{R}/2$ **then**
      for all $k \in \Delta_i^-$ **do**
        $l_k \leftarrow Extend^{\mathrm{bw}}(l_i, k)$
        $\Gamma_k^{\mathrm{bw}} \leftarrow EFF(\Gamma_k^{\mathrm{bw}}, l_k)$
        if Γ_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 $l_i = (\sigma^i, R^i, C^i, i) \in \Gamma_i^{\mathrm{fw}}$
    for all $j \in \mathcal{V}$
      for all $l_j = (\sigma^j, R^j, C^j, j) \in \Gamma_j^{\mathrm{bw}}$
        if $Feasible(l_i, l_j)$ **and** $HalfWay(l_i, l_j)$ **then**
          $Save(l_i, l_j)$

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

$$\sigma' \leq \sigma''$$
$$R' \leq R''$$
$$C' \leq C''$$

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. [10]). Irnich and Villeneuve [20] 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 bidirectional 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 bidirectional search in the same way described earlier, that is, on the basis of the value of a critical resource.

When bounded bidirectional 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.

Algorithm 3. RCESPP—Decremental state space relaxation

// Initialization //
$\Psi = \emptyset$
$\Theta = \emptyset$
repeat
  $\Theta = \Theta \cup \Psi$
  $\Gamma_s^{\mathrm{fw}} \leftarrow \{(\mathbf{0}, \mathbf{0}, 0, s)\}$
  $\Gamma_t^{\mathrm{bw}} \leftarrow \{(\mathbf{0}, \mathbf{0}, 0, t)\}$
  for all $i \in \mathcal{V} \setminus \{s\}$ **do** $\Gamma_i^{\mathrm{fw}} \leftarrow \emptyset$
  for all $i \in \mathcal{V} \setminus \{t\}$ **do** $\Gamma_i^{\mathrm{bw}} \leftarrow \emptyset$
  $E \leftarrow \{s, t\}$
  // Search //
  repeat
    // Vertex selection //
    Select $i \in E$
    // Forward extension //
    for all $l_i = (S_\Theta^i, R^i, C^i, i) \in \Gamma_i^{\mathrm{fw}}$ **do**
      if $\hat{r}^i < \hat{R}/2$ **then**
        for all $j \in \Delta_i^+$ such that $j \notin \Theta$ **or** $S_j^i = 0$ **do**
          $l_j \leftarrow Extend^{\mathrm{fw}}(l_i, j)$
          $\Gamma_j^{\mathrm{fw}} \leftarrow EFF(\Gamma_j^{\mathrm{fw}}, l_j)$
          if Γ_j^{fw} has changed **then** $E \leftarrow E \cup \{j\}$
    // Backward extension //
    for all $l_i = (S_\Theta^i, R^i, C^i, i) \in \Gamma_i^{\mathrm{bw}}$ **do**
      if $\hat{r}^i < \hat{R}/2$ **then**
        for all $k \in \Delta_i^-$ such that $k \notin \Theta$ **or** $S_k^i = 0$ **do**
          $l_k \leftarrow Extend^{\mathrm{bw}}(l_i, k)$
          $\Gamma_k^{\mathrm{bw}} \leftarrow EFF(\Gamma_k^{\mathrm{bw}}, l_k)$
          if Γ_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 $l_i = (S_\Theta^i, R^i, C^i, i) \in \Gamma_i^{\mathrm{fw}}$
      for all $j \in \mathcal{V}$
        for all $l_j = (S_\Theta^j, R^j, C^j, j) \in \Gamma_j^{\mathrm{bw}}$
          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 //
  $\Psi = MultipleVisits()$
until $\Psi = \emptyset$

TABLE 1. RCESPP with capacity—50 vertices.

Instance name	Exact D.P.		Resources + Arcs			Resource + Cycles			DSSR		
	Labels	Time	Nodes	Time	Gap	Nodes	Time	Gap	Iter	C.N.	Time
c_50_01	30	0.00	1	0.00	—	1	0.00	—	1	0	0.00
c_50_02	104	0.00	1	0.00	—	1	0.00	—	1	0	0.00
c_50_03	277	0.00	7	0.01	—	1	0.00	—	1	0	0.00
c_50_04	812	0.02	19	0.08	—	10	0.05	—	3	2	0.02
c_50_05	1978	0.03	11	0.05	—	22	0.09	—	2	2	0.01
c_50_06	8185	0.04	87	0.48	—	116	0.91	—	3	3	0.04
c_50_07	10694	0.78	35	0.24	—	52	0.33	—	2	3	0.02
c_50_08	43525	14.99	315	3.19	—	124	1.81	—	5	7	0.25
c_50_09	51467	19.66	673	5.80	—	224	1.78	—	4	8	0.18
c_50_10	211951	298.45	3919	44.56	—	3039	53.34	—	5	10	1.82
r_50_01	40	0.00	1	0.00	—	1	0.00	—	1	0	0.00
r_50_02	129	0.01	1	0.01	—	1	0.00	—	1	0	0.00
r_50_03	296	0.01	7	0.04	—	1	0.00	—	3	3	0.02
r_50_04	616	0.02	7	0.06	—	1	0.00	—	2	3	0.02
r_50_05	1224	0.05	591	6.64	—	7	0.09	—	6	7	0.17
r_50_06	2349	0.11	271	4.49	—	83	1.18	—	4	6	0.12
r_50_07	4269	0.24	6009	90.34	—	122	1.12	—	2	4	0.06
r_50_08	7731	0.32	3149	38.75	—	95	0.98	—	5	11	0.48
r_50_09	13638	0.97	191	6.12	—	17	0.53	—	4	9	0.40
r_50_10	22709	2.37	59	1.16	—	7	0.26	—	3	8	0.34
rc_50_01	21	0.00	1	0.00	—	1	0.00	—	1	0	0.00
rc_50_02	87	0.00	1	0.00	—	1	0.00	—	1	0	0.00
rc_50_03	136	0.00	11	0.02	—	1	0.01	—	4	3	0.00
rc_50_04	300	0.01	725	1.58	—	73	0.17	—	5	4	0.02
rc_50_05	421	0.01	1573	4.21	—	173	0.42	—	4	4	0.02
rc_50_06	865	0.03	73573	562.40	—	1774	8.29	—	7	8	0.09
rc_50_07	1006	0.04	3417	19.66	—	84	0.50	—	4	7	0.05
rc_50_08	1827	0.07	239467	—	6.25	10505	49.19	—	7	11	0.20
rc_50_09	2026	0.08	29021	348.97	—	960	7.68	—	6	11	0.20
rc_50_10	3721	0.18	254931	—	2.0	16679	156.19	—	6	11	0.27

5.1. Paths Join and Solutions Uniqueness

The bounded bidirectional 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 is unpleasant when we solve the RCESPP as the pricing problem in a branch-and-price framework, because in that context we search for many *different* columns with negative reduced cost. To overcome this drawback, we devised 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 $\hat{r}^{\mathrm{fw}}$ and $\hat{r}^{\mathrm{bw}}$ 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 $\phi = |\hat{r}^{\mathrm{fw}} - \hat{r}^{\mathrm{bw}}|$ 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 $\hat{r}^{\mathrm{fw}} < \hat{r}^{\mathrm{bw}}$ and with the previous position if $\hat{r}^{\mathrm{fw}} > \hat{r}^{\mathrm{bw}}$. In case of tie between two positions for which ϕ is minimum, we choose the one with $\hat{r}^{\mathrm{fw}} > \hat{r}^{\mathrm{bw}}$. This test guarantees that each *s-t* path is generated only once.

In Algorithm 2, we report the pseudocode of the state-space relaxation algorithm, using the same notation as in Algorithm 1. In the joining phase, the procedure *HalfWay*(l_i, l_j) checks whether the *s-t* path obtainable from the two states l_i and l_j satisfies the "half-way-point" conditions.

6. BRANCH-AND-BOUND

In this section, we describe a branch-and-bound algorithm that solves the RCESPP to optimality, exploiting the RCSPP lower bound given by the bounded bidirectional dynamic-programming algorithm with state-space relaxation. In Section 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.

TABLE 2. RCESPP with capacity—100 vertices.

Instance name	Exact D.P.		Resources + Arcs			Resource + Cycles			DSSR		
	Labels	Time	Nodes	Time	Gap	Nodes	Time	Gap	Iter	C.N.	Time
c_100_01	55	0.00	1	0.00	—	1	0.00	—	1	0	0.00
c_100_02	205	0.01	1	0.01	—	1	0.01	—	1	0	0.00
c_100_03	579	0.03	7	0.04	—	1	0.02	—	2	1	0.03
c_100_04	2093	0.18	19	0.26	—	10	0.15	—	3	2	0.06
c_100_05	4722	0.26	43	0.36	—	22	0.31	—	3	4	0.07
c_100_06	22505	4.42	193	5.19	—	699	16.31	—	4	5	0.21
c_100_07	30871	5.94	65	1.84	—	99	2.65	—	3	6	0.18
c_100_08	171703	178.04	1275	77.50	—	517	30.53	—	6	10	1.34
c_100_09	2176993	226.04	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	150	0.00	15	0.04	—	10	0.04	—	3	3	0.02
r_100_02	972	0.05	459	3.61	—	225	1.45	—	2	4	0.06
r_100_03	4285	0.41	5337	126.87	—	776	15.03	—	4	5	0.59
r_100_04	17054	3.14	7639	306.49	—	1819	69.68	—	4	7	2.80
r_100_05	72202	32.04	81677		7.3	4464	198.93	—	3	5	2.87
r_100_06	270466	371.13	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	6	11	1002.30
r_100_10	—	—	2551	—	67.4	6118	—	43.2	—	—	—
rc_100_01	21	0.00	1	0.00	—	1	0.00	—	1	0	0.00
rc_100_02	251	0.01	1	0.01	—	1	0.00	—	1	0	0.00
rc_100_03	699	0.03	31	0.33	—	10	0.13	—	1	0	0.00
rc_100_04	1823	0.14	7	0.20	—	10	0.27	—	2	1	0.07
rc_100_05	4527	0.38	1669	71.45	—	64	2.35	—	4	4	0.29
rc_100_06	11400	1.27	71	2.52	—	60	3.01	—	3	4	0.35
rc_100_07	24787	4.09	4403	376.85	—	752	59.22	—	4	5	0.92
rc_100_08	55665	15.51	5735	353.00	—	254	25.00	—	4	7	1.77
rc_100_09	110506	53.62	739	109.08	—	391	28.76	—	3	5	1.40
rc_100_10	230054	209.30	25055	—	3.7	7385	1191.96	—	5	10	7.33

6.0.1. 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.

6.0.2. 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 that the vertex has not been visited yet. At each vertex i, the algorithm chooses the next feasible vertex j such that $j \in argmin_k\{c_{ik} - \lambda_k\}$.

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.

6.1.1. 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 on vertices $v_1, v_2, \ldots, v_k$, that is, the number of arcs traversed between two visits to the same vertex. At child node $h = 0, \ldots, k-1$, the consecutive vertices $v_1, \ldots, v_h$ are merged into a macrovertex and the arc from v_h to v_{h+1} is forbidden. We experimentally observed that forbidding the cycles of length two, we often obtained cycles of length three.

6.1.2. Branching on Arcs. This binary branching scheme consists in 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.

6.1.3. 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. [17] for routing problems with time windows and it can be adapted to any problem with a critical resource whose consumption $\hat{r}$ is strictly monotone along the path. Given a branching vertex $\hat{\imath}$, let $\hat{r}'$ and $\hat{r}''$ be the two

TABLE 3. RCESPP with distribution and collection—50 vertices.

Instance name	Exact D.P.		Resources + Arcs			Resource + Cycles			DSSR		
	Labels	Time	Nodes	Time	Gap	Nodes	Time	Gap	Iter	C.N.	Time
c_50_01	25	0.00	1	0.00	—	1	0.00	—	1	0	0.00
c_50_02	85	0.00	1	0.00	—	1	0.00	—	1	0	0.00
c_50_03	188	0.01	1	0.02	—	1	0.01	—	2	1	0.00
c_50_04	632	0.02	7	0.02	—	10	0.04	—	2	1	0.01
c_50_05	1535	0.07	15	0.07	—	22	0.10	—	2	2	0.02
c_50_06	5507	0.27	27	0.17	—	122	0.94	—	3	3	0.05
c_50_07	10578	0.69	17	0.16	—	53	0.48	—	2	3	0.03
c_50_08	33588	7.23	83	0.90	—	91	1.44	—	5	7	0.47
c_50_09	56812	16.95	69	0.88	—	207	2.47	—	3	5	0.19
c_50_10	181699	140.59	119	3.56	—	129	3.66	—	5	10	2.20
r_50_01	51	0.00	1	0.00	—	1	0.00	—	1	0	0.00
r_50_02	116	0.00	5	0.02	—	7	0.02	—	2	1	0.01
r_50_03	298	0.01	1	0.01	—	1	0.01	—	3	3	0.01
r_50_04	585	0.02	1	0.03	—	1	0.03	—	2	3	0.02
r_50_05	1222	0.04	11	0.14	—	7	0.08	—	5	6	0.13
r_50_06	2345	0.09	17	0.28	—	45	0.62	—	3	5	0.07
r_50_07	4312	0.22	3	0.07	—	3	0.08	—	2	4	0.06
r_50_08	7772	0.50	71	0.97	—	122	1.13	—	4	7	0.19
r_50_09	13866	1.20	13	0.42	—	19	0.59	—	3	7	0.21
r_50_10	23788	3.03	5	0.21	—	5	0.21	—	3	7	0.25
rc_50_01	23	0.00	1	0.00	—	1	0.00	—	1	0	0.00
rc_50_02	58	0.00	1	0.00	—	1	0.00	—	1	0	0.01
rc_50_03	111	0.01	1	0.01	—	1	0.01	—	3	2	0.01
rc_50_04	247	0.01	55	0.12	—	73	0.14	—	5	4	0.02
rc_50_05	377	0.02	171	0.51	—	114	0.28	—	4	5	0.03
rc_50_06	639	0.02	3321	19.86	—	1156	5.01	—	6	7	0.09
rc_50_07	967	0.03	397	3.00	—	233	1.40	—	6	7	0.09
rc_50_08	1463	0.05	595	6.41	—	405	4.07	—	4	7	0.05
rc_50_09	2119	0.10	4363	63.94	—	666	8.13	—	7	9	0.22
rc_50_10	3201	0.15	12045	210.69	—	6551	79.95	—	6	11	0.24

values of resource consumption in two states associated with $\hat{\imath}$ with $\hat{r}' < \hat{r}''$. Then an integer value $\bar{r}$ is chosen such that $\hat{r}' < \bar{r} \leq \hat{r}''$. Two children nodes are generated imposing that the value of $\hat{r}$ at vertex $\hat{\imath}$ satisfies $\hat{r} \geq \bar{r}$ in one child node and $\hat{r} \leq \bar{r} - 1$ 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 $[a^{\hat{\imath}}, b^{\hat{\imath}}]$ of feasible values for the critical resource; when a path reaches that vertex with a critical resource consumption less than $a^{\hat{\imath}}$, the consumption is set to $a^{\hat{\imath}}$; when it reaches the vertex with a critical resource consumption greater than $b^{\hat{\imath}}$, 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.

6.1.4. 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 $\hat{r}'$ and $\hat{r}''$ as the resource consumptions at the first (resp. last) two visits of the branching vertex and we choose $\bar{r} = \left\lceil \frac{\hat{r}' + \hat{r}''}{2} \right\rceil$. 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 to the vertices, 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, and in the subsequent run, the dynamic-programming algorithm

TABLE 4. RCESPP with distribution and collection—100 vertices.

Instance name	Exact D.P.		Resources + Arcs			Resource + Cycles			DSSR		
	Labels	Time	Nodes	Time	Gap	Nodes	Time	Gap	Iter	C.N.	Time
c_100_01	47	0.00	1	0.00	—	1	0.00	—	1	0	0.00
c_100_02	166	0.00	1	0.00	—	1	0.00	—	1	0	0.02
c_100_03	381	0.02	1	0.00	—	1	0.00	—	2	1	0.02
c_100_04	1426	0.12	7	0.10	—	10	0.13	—	3	2	0.06
c_100_05	3689	0.22	15	0.26	—	22	0.40	—	3	4	0.10
c_100_06	14800	2.04	15	0.54	—	597	17.26	—	4	5	0.25
c_100_07	29977	5.10	35	1.24	—	101	3.67	—	3	6	0.27
c_100_08	123907	82.29	175	6.56	—	192	12.57	—	6	10	2.17
c_100_09	229386	218.82	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	153	0.00	1	0.00	—	1	0.00	—	1	0	0.00
r_100_02	994	0.06	801	7.47	—	222	1.52	—	2	4	0.06
r_100_03	4706	0.55	7275	186.84	—	831	16.90	—	4	5	0.64
r_100_04	18995	4.44	15297	790.61	—	1814	69.59	—	3	5	1.34
r_100_05	83158	47.10	40991	3503.40	—	5242	357.35	—	3	5	3.71
r_100_06	351405	686.55	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	6	11	1200.36
r_100_10	—	—	3559	—	62.8	4025	—	61.0	—	—	—
rc_100_01	47	0.00	1	0.00	—	1	0.00	—	1	0	0.00
rc_100_02	229	0.01	1	0.00	—	1	0.00	—	1	0	0.00
rc_100_03	642	0.04	7	0.10	—	10	0.13	—	2	1	0.03
rc_100_04	1776	0.15	7	0.19	—	10	0.26	—	2	1	0.07
rc_100_05	4331	0.49	27	1.16	—	61	2.25	—	4	4	0.30
rc_100_06	10794	1.54	23	1.18	—	54	2.98	—	3	4	0.33
rc_100_07	24657	5.83	801	74.78	—	736	58.96	—	4	5	0.97
rc_100_08	55131	31.83	361	44.60	—	242	24.13	—	3	5	1.11
rc_100_09	116239	87.50	235	23.30	—	389	29.55	—	3	5	1.55
rc_100_10	242383	688.90	5671	971.28	—	7162	1207.45	—	4	8	6.11

prevents multiple visits to those vertices, allowing multiple visits to the others. This is easily accomplished by extending the state-space relaxation labels with a binary vector $\hat{S}$ playing the same role as S in exact dynamic-programming. The size of $\hat{S}$ is however restricted only to the critical vertices. When $\hat{S}$ contains all the vertices, the algorithm is equivalent to exact dynamic-programming; when $\hat{S}$ 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 Θ' be the set of critical vertices at the current iteration. Let Ψ be 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 increases at each iteration and eventually the algorithm provides the optimal solution to the RCESPP without having recourse to branching. In Algorithm 3, we report the pseudo-code of the decremental state-space relaxation algorithm, with the same notation as in Algorithm 1. In addition procedure, *MultipleVisits* returns the set of vertices visited more than once in the current optimal path and S_Θ indicates the binary vector S restricted to the components corresponding to the critical vertices.

8. EXPERIMENTAL RESULTS

For our experiments, we used the same instances as in Righini and Salani [23] and Feillet et al. [16]; they are derived from the well-known Solomon's CVRPTW 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 data sets are divided into random, clustered, and random-clustered categories, according to the displacement of the customers. Instances belonging to the same data-set 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 data-sets (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 10 RCESPP instances with 50 nodes and 10 RCESPP instances with 100 nodes, by choosing 10 different values for the vehicle capacity from 10 to 100.

TABLE 5. RCESPP with capacity and time windows—50 vertices.

Instance name	Exact D.P.		Resources + Arcs			Resource + Cycles			DSSR		
	Labels	Time	Nodes	Time	Gap	Nodes	Time	Gap	Iter	C.N.	Time
c101_50	500	0.02	1	0.00	—	1	0.00	—	1	0	0.00
c102_50	2747	0.22	15	0.19	—	12	0.19	—	2	2	0.12
c103_50	27656	13.73	2533	127.32	—	966	27.74	—	4	6	14.06
c104_50	—	—	35781	—	1.2	24785	1835.47	—	6	11	342.67
c105_50	603	0.02	1	0.02	—	1	0.02	—	1	0	0.01
c106_50	509	0.02	1	0.03	—	1	0.03	—	1	0	0.02
c107_50	661	0.03	1	0.04	—	1	0.04	—	1	0	0.02
c108_50	924	0.04	1	0.04	—	1	0.04	—	1	0	0.04
c109_50	2177	0.20	81	1.90	—	78	1.69	—	8	11	1.35
r101_50	189	0.00	1	0.00	—	1	0.00	—	1	0	0.00
r102_50	642	0.03	5	0.05	—	4	0.04	—	4	5	0.08
r103_50	1950	0.11	817	9.83	—	103	1.07	—	3	4	0.27
r104_50	10592	1.22	83	2.46	—	264	7.08	—	3	5	0.77
r105_50	368	0.01	1	0.01	—	1	0.01	—	1	0	0.00
r106_50	882	0.04	13	0.14	—	9	0.08	—	4	6	0.18
r107_50	2349	0.16	15435	248.92	—	268	3.10	—	4	5	0.47
r108_50	11253	1.33	5	0.15	—	5	0.15	—	2	4	0.48
r109_50	741	0.02	1	0.02	—	1	0.02	—	1	0	0.02
r110_50	1769	0.10	1	0.04	—	1	0.04	—	1	0	0.04
r111_50	2202	0.15	43	0.73	—	46	0.66	—	3	5	0.33
r112_50	3760	0.32	1	0.05	—	1	0.05	—	1	0	0.05
rc101_50	357	0.00	1	0.00	—	1	0.00	—	1	0	0.01
rc102_50	1020	0.03	17	0.13	—	17	0.13	—	3	3	0.09
rc103_50	3448	0.18	112	3.45	—	6168	90.88	—	5	11	0.95
rc104_50	8926	0.84	258	7.65	—	312	8.31	—	7	15	4.77
rc105_50	1000	0.03	63	0.48	—	60	0.32	—	3	4	0.11
rc106_50	999	0.03	71	0.52	—	676	4.64	—	4	7	0.16
rc107_50	3479	0.15	29567	653.53	—	13452	195.907	—	6	9	0.83
rc108_50	8671	0.53	34205	1143.65	—	58476	1474.19	—	6	13	2.12

For the RCESPP with distribution and collection, we kept the original delivery requests and we derived the pick-up requests as follows: $p_i = \lfloor 0.8d_i \rfloor$ if i is odd and $p_i = \lfloor 1.2d_i \rfloor$ if i is even. We generated 10 instances with 50 nodes and 10 instances with 100 nodes as before.

Finally, for the RCESPP with capacities and time windows, we considered the original instances of Solomon's data-set. In addition, we also defined another data-set 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. [16] 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 1.6 GHz processor with 512 MB RAM. The algorithms were coded in ANSI-C and compiled with gcc 3.0.4.

Tables 1–8 report on the experimental comparison between the elementary bidirectional algorithm with bounds described in [23], the state-space relaxation algorithm coupled with branch-and-bound and the decremental state-space relaxation algorithm. For the elementary bidirectional algorithm with bounds, named *Exact D.P.* in the tables, we report the total number of nondominated 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 decremental state-space algorithm, named DSSR, we report the number of iterations, that is, the number of times the dynamic-programming algorithm has been invoked, the number of critical nodes in the last iteration and the computing time. Empty cells mean that the optimal solution was not found within the time limit of one hour.

8.1. Capacities

Results reported in Tables 1 and 2 show that for 50 vertices instances the decremental state-space relaxation algorithm clearly outperformed all other algorithms on all classes of instances except for the rc-class, where exact bidirectional and bounded dynamic-programming was quite fast. However, these are very easy instances for all algorithms considered:

TABLE 6. RCESPP with capacity and time windows—100 vertices.

Instance name	Exact D.P.		Resources + Arcs			Resource + Cycles			DSSR		
	Labels	Time	Nodes	Time	Gap	Nodes	Time	Gap	Iter	C.N.	Time
c101_100	1039	0.14	1	0.02	—	1	0.02	—	1	0	0.02
c102_100	9759	3.97	15	0.89	—	12	0.67	—	2	2	0.63
c103_100	95138	148.50	2539	507.78	—	3075	485.68	—	5	9	40.15
c104_100	—	—	17553	—	17.8	23402	—	16.5	7	16	919.41
c105_100	1256	0.22	1	0.06	—	1	0.06	—	1	0	0.06
c106_100	1502	0.33	1	0.07	—	1	0.07	—	1	0	0.07
c107_100	1378	0.30	1	0.08	—	1	0.08	—	1	0	0.08
c108_100	2109	0.58	1	0.17	—	1	0.17	—	1	0	0.17
c109_100	4816	1.90	111	14.18	—	101	12.66	—	8	13	9.19
r101_100	765	0.05	1	0.00	—	1	0.00	—	1	0	0.00
r102_100	13021	4.51	4203	438.82	—	1003	96.25	—	3	6	21.69
r103_100	75599	105.77	377	128.71	—	252	61.10	—	4	7	159.74
r104_100	349866	1278.57	4531	—	0.9	1945	1013.04	—	3	5	78.32
r105_100	1679	0.17	1	0.03	—	1	0.03	—	1	0	0.03
r106_100	19411	11.03	26059	—	3.4	3344	467.29	—	4	7	71.60
r107_100	83422	141.20	1417	717.37	—	732	232.91	—	5	13	136.36
r108_100	312346	1094.81	1593	1098.05	—	1451	611.61	—	3	5	146.58
r109_100	4417	0.87	49	3.39	—	49	3.45	—	3	5	2.93
r110_100	22744	12.71	307	69.25	—	406	77.55	—	3	6	19.31
r111_100	44094	39.38	2669	866.34	—	361	129.24	—	3	7	53.16
r112_100	269888	1019.10	2167	2227.74	—	665	385.00	—	7	14	316.91
rc101_100	1038	0.08	1	0.02	—	1	0.02	—	1	0	0.02
rc102_100	5209	0.82	17	1.12	—	21	1.34	—	4	4	3.17
rc103_100	22618	9.87	861	110.78	—	6494	587.39	—	4	8	35.37
rc104_100	137013	202.25	607	194.68	—	1054	203.36	—	4	8	102.56
rc105_100	3288	0.41	7	0.43	—	13	0.67	—	4	6	1.14
rc106_100	3124	0.37	31	1.85	—	12	0.56	—	3	3	1.01
rc107_100	10651	2.17	87	8.50	—	27	2.79	—	4	6	3.65
rc108_100	39880	20.12	239	32.34	—	143	12.43	—	3	5	4.84

TABLE 7. RCESPP with capacity and time windows—c_104, 50 vertices.

Instance name	Exact D.P.		Resources + Arcs			Resource + Cycles			DSSR		
	Labels	Time	Nodes	Time	Gap	Nodes	Time	Gap	Iter	C.N.	Time
c104_50_01	100	0.00	1	0.01	—	1	0.01	—	1	0	0.01
c104_50_02	157	0.00	1	0.01	—	1	0.01	—	1	0	0.01
c104_50_03	229	0.01	1	0.01	—	1	0.01	—	1	0	0.01
c104_50_04	235	0.01	1	0.01	—	1	0.01	—	1	0	0.01
c104_50_05	244	0.01	1	0.01	—	1	0.01	—	1	0	0.01
c104_50_06	323	0.01	1	0.02	—	1	0.02	—	1	0	0.02
c104_50_07	532	0.01	1	0.02	—	1	0.02	—	1	0	0.02
c104_50_08	637	0.04	1	0.02	—	1	0.02	—	1	0	0.02
c104_50_09	772	0.05	1	0.02	—	1	0.02	—	1	0	0.02
c104_50_10	841	0.06	1	0.02	—	1	0.02	—	1	0	0.02
c104_50_11	1479	0.11	1	0.03	—	1	0.03	—	1	0	0.03
c104_50_12	2349	0.24	1	0.03	—	1	0.03	—	1	0	0.03
c104_50_13	3827	0.46	1	0.04	—	1	0.04	—	1	0	0.04
c104_50_14	4081	0.53	19	0.24	—	13	0.18	—	2	3	0.14
c104_50_15	5556	0.76	13	0.22	—	9	0.12	—	3	4	0.49
c104_50_16	9463	2.16	39	0.88	—	12	0.25	—	3	4	0.53
c104_50_17	18631	5.15	45	1.09	—	15	0.35	—	3	4	0.60
c104_50_18	21792	6.63	145	3.53	—	76	1.66	—	3	4	0.61
c104_50_19	25698	9.42	77	2.43	—	64	1.41	—	3	5	1.20
c104_50_20	36875	23.86	85	2.02	—	91	2.35	—	4	6	2.11
c104_50_21	82108	76.42	179	5.95	—	144	3.99	—	4	6	2.69
c104_50_22	106183	115.17	199	5.35	—	91	2.52	—	3	6	2.31
c104_50_23	127235	157.86	255	11.00	—	214	9.34	—	8	11	17.45
c104_50_24	159960	302.80	276	19.70	—	238	17.98	—	5	7	13.43
c104_50_25	335617	1166.70	321	18.36	—	381	19.83	—	4	7	13.31

TABLE 8. RCESPP with capacity and time windows—c_104, 100 vertices.

Instance name	Exact D.P.		Resources + Arcs			Resource + Cycles			DSSR		
	Labels	Time	Nodes	Time	Gap	Nodes	Time	Gap	Iter	C.N.	Time
c104_100_01	205	0.01	1	0.04	—	1	0.04	—	1	0	0.04
c104_100_02	286	0.01	1	0.05	—	1	0.05	—	1	0	0.05
c104_100_03	415	0.03	1	0.06	—	1	0.06	—	1	0	0.06
c104_100_04	463	0.05	1	0.07	—	1	0.07	—	1	0	0.07
c104_100_05	477	0.06	1	0.07	—	1	0.07	—	1	0	0.07
c104_100_06	603	0.06	1	0.08	—	1	0.08	—	1	0	0.08
c104_100_07	975	0.12	1	0.08	—	1	0.08	—	1	0	0.08
c104_100_08	1232	0.22	1	0.09	—	1	0.09	—	1	0	0.09
c104_100_09	1475	0.34	1	0.09	—	1	0.09	—	1	0	0.09
c104_100_10	1645	0.42	1	0.09	—	1	0.09	—	1	0	0.09
c104_100_11	2738	0.65	1	0.11	—	1	0.11	—	1	0	0.11
c104_100_12	4595	1.36	1	0.12	—	1	0.12	—	1	0	0.12
c104_100_13	7437	2.63	1	0.11	—	1	0.11	—	1	0	0.11
c104_100_14	8579	3.58	33	1.76	—	11	0.74	—	3	3	1.49
c104_100_15	11053	4.86	65	3.24	—	21	1.05	—	2	3	1.26
c104_100_16	19267	10.42	249	15.12	—	23	1.62	—	2	3	1.56
c104_100_17	39260	24.91	211	20.02	—	21	1.60	—	2	3	1.65
c104_100_18	53823	41.34	75	8.64	—	30	3.54	—	2	3	1.65
c104_100_19	66373	58.19	45	5.85	—	30	2.99	—	4	6	6.32
c104_100_20	92042	112.48	59	4.66	—	42	2.04	—	4	5	7.84
c104_100_21	198464	350.88	67	7.29	—	48	4.26	—	4	5	8.79
c104_100_22	318067	740.42	89	10.78	—	59	5.94	—	3	6	7.54
c104_100_23	441254	1238.62	53	7.94	—	53	7.60	—	7	9	32.06
c104_100_24	554831	2087.82	141	31.02	—	143	33.46	—	5	7	29.87
c104_100_25	—	—	187	47.05	—	17	44.32	—	4	7	27.74

the computing times are all smaller than 1 s. The branch-and-bound algorithms sometimes dominated exact dynamic-programming but they also failed 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 is evident. Decremental state-space relaxation dramatically reduced the computing time up to two orders of magnitude. The branch-and-bound algorithms had performances similar to those of exact dynamic-programming and the two hybrid branching strategies did not dominate each other.

8.2. *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 decremental state-space relaxation algorithm solved all instances except instance r_100_10 in less than 1200 s, outperforming the other algorithms and reducing the computing time by two orders of magnitude in some cases.

8.3. *Capacities and Time Windows*

All Solomon's instances with 50 and 100 nodes were solved by the decremental state-space relaxation algorithm. It should be pointed out that the most difficult instance, namely instance c_104, was solved within 920 s. For the other original Solomon's instances, the branch-and-bound algorithms were not competitive, because of the tightness and the displacement of the time windows that often allowed exact dynamic-programming to go faster, because of the relatively small number of feasible solutions.

8.4. *Tightness of the Constraints*

The last two tables, namely 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 nondominated 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 decremental state-space relaxation algorithm solved all instances in a few seconds, whereas exact dynamic-programming showed a dramatic growth in computing time. When constraints are very tight decremental state-space relaxation and branch-and-bound have comparable computational performances.

The number of critical nodes in decremental state-space relaxation we observed was never greater than 16. We remark that our current implementation of the decremental state-space relaxation 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 can be further improved.

Last but not the least, we observed that the implementation of decremental state-space relaxation is by far easier than that of the competitor algorithms considered here.

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. [13], we have shown how it can be improved by new ideas, such as bidirectional 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 bidirectional search can be adapted to 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 favorable 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 nondominated 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 reoptimization techniques like those of Desrochers and Soumis [11], so that each iteration of the algorithm does not restart from scratch but can reuse 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. [10] for solving the CVRPTW, while recently Feillet et al. [16] 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 computing time and lower bound tightness definitely deserves further investigation and it will be subject of future research. 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 and to two anonymous referees for their helpful comments.

This paper was submitted to Networks in January 2005. During the review process (first revision received in March 2006) the idea of decremental state-space relaxation also appeared in a paper by Boland et al. [4] in Op. Res. Letters, January 2006. The two results were obtained independently.

REFERENCES

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

[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, Network flows: Theory, Algorithms and Applications, Prentice-Hall, Englewood Cliffs, N.J., 1993.

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

[4] N. Boland, J. Dethridge, and I. Dumitrescu, Accelerated label setting algorithms for the elementary resource constrained shortest path problem, Oper Res Lett 34 (2006), 58–68.

[5] J. Bramel and D. Simchi-Levi, Set-covering-based algorithms for the capacitated VRP, The vehicle-routing problem, SIAM monographs on discrete mathematics and applications, P. Toth and D. Vigo (Editors), 2002, pp. 85–106.

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

[7] J.F. Cordeau, G. Desaulniers, J. Desrosiers, M.M. Solomon, and F. Soumis, VRP with time windows, The vehicle-routing problem, SIAM monographs on discrete mathematics and applications, P. Toth and D. Vigo (Editors), 2002.

[8] M. Dell'Amico, G. Righini, and M. Salani, A branch-and-price approach to the vehicle-routing problem with simultaneous distribution and collection, Transport Sci 40(2), 2006, 235–247.

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

[10] M. Desrochers, J. Desrosiers, and M. Solomon, A new optimization algorithm for the vehicle-routing problem with time windows, Oper Res 40 (1992), 342–354.

[11] M. Desrochers and F. Soumis, A reoptimization algorithm for the shortest path problem with time windows, Eur J Oper Res 35 (1988), 242–254.

[12] J. Desrosiers, Y. Dumas, M. Solomon, and F. Soumis, Time constrained routing and scheduling in Network Routing, Handbooks in operations research and management science, M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser (Editors), Elsevier Science, 1995.

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

[14] M. Dror, Note on the complexity of the shortest path models for column generation in VRPTW, Oper Res 42 (1994), 977–978.

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

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

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

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

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

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

[21] N. Kohl, J. Desrosiers, O.B.G. Madsen, M.M. Solomon, and F. Soumis, 2-path cuts for the vehicle-routing problem with time windows, Transport Sci 33 (1999), 101–116.

[22] A. Mingozzi, L. Bianco, and S. Ricciardelli, Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints, Oper Res 45 (1997), 365–377.

[23] G. Righini and M. Salani, Symmetry helps: Bounded bidirectional dynamic-programming for the elementary shortest path problem with resource constraints, Discrete Optimization 3(3), 2006, 255–273.

[24] P. Toth and D. Vigo, Capacitated vehicle-routing problems in The vehicle-routing problem, SIAM Monographs on Discrete Mathematics and Applications, P. Toth and D. Vigo (Editors), 2002.