Available online at www.sciencedirect.com
SCIENCE DIRECT®
Discrete Optimization 3 (2006) 255–273
DISCRETE OPTIMIZATION
www.elsevier.com/locate/disopt

Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints

Giovanni Righini*, Matteo Salani

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

Received 30 September 2004; received in revised form 1 November 2005; accepted 1 May 2006
Available online 12 July 2006

Abstract

When vehicle routing problems with additional constraints, such as capacity or time windows, are solved via column generation and branch-and-price, it is common that the pricing subproblem requires the computation of a minimum cost constrained path on a graph with costs on the arcs and prizes on the vertices. A common solution technique for this problem is dynamic programming. In this paper we illustrate how the basic dynamic programming algorithm can be improved by bounded bi-directional search and we experimentally evaluate the effectiveness of the enhancement proposed. We consider as benchmark problems the elementary shortest path problems arising as pricing subproblems in branch-and-price algorithms for the capacitated vehicle routing problem, the vehicle routing problem with distribution and collection and the capacitated vehicle routing problem with time windows.

Keywords: Shortest path; Vehicle routing; Dynamic programming; Column generation

1. Introduction

Vehicle routing problems require us to compute a set of tours for a fleet of vehicles that must provide a certain kind of service to a given set of customers. Each vehicle starts from a given depot and goes back to it after visiting a subset of customers. The objective is to minimize the total distance traveled. The structure of vehicle routing problems suggests to reformulate them as set covering problems and to apply column generation, because a solution is made by a set of sub-tours, one for each vehicle of the fleet, which can be computed independently provided that they cover the set of customers to be visited. A comprehensive treatment of column generation approaches to vehicle routing problems can be found in [7] and in [5]. In a column generation approach the master problem is a set covering problem as follows:

$$\text{minimize} \sum_{f \in \mathcal{F}} c_f z_f$$

$$\text{subject to} \sum_{f \in \mathcal{F}} x_{if} z_f \geq 1 \quad \forall i \in \mathcal{N} \tag{1}$$

* Corresponding author. Tel.: +39 0373898060; fax: +39 0373898010.
E-mail addresses: righini@dti.unimi.it (G. Righini), salani@dti.unimi.it (M. Salani).

1572-5286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.disopt.2006.05.007

$$-\sum_{f\in\mathcal{F}} z_f \geq -V \tag{2}$$

$$z_f \in \{0,1\} \quad \forall f \in \mathcal{F} \tag{3}$$

where $\mathcal{N}$ is the set of customers, $\mathcal{F}$ is the set of feasible vehicle routes, V is the number of available vehicles, c_f is the cost of route $f \in \mathcal{F}$ and x_{if} is the number of times route $f \in \mathcal{F}$ visits customer $i \in \mathcal{N}$. The linear relaxation of this set covering reformulation usually yields very tight lower bounds (see for instance Bramel and Simchi-Levi [4] and the references therein). However since in general $\mathcal{F}$ contains an exponential number of columns, only a subset $\mathcal{F}'$ is kept in a restricted linear master problem and further feasible routes must be generated on the fly, by the iterated solution of a pricing problem. The pricing problem consists in finding routes with negative reduced cost or proving that none exists. The reduced cost of route $f \in \mathcal{F}$ is:

$$\bar{c}_f = c_f - \sum_{i\in\mathcal{N}} x_{if}\lambda_i + \lambda_0$$

where (λ, λ_0) is the vector of non-negative dual variables corresponding to constraints (1) and (2) in the restricted linear master problem. It is common that the routes of the vehicles must satisfy some additional constraints, due for instance to capacity, precedence constraints or time windows. Such constraints do not modify the structure of the master problem, but rather they are taken into account in the pricing problem, that is they restrict the set $\mathcal{F}$ of feasible routes.

The kind of pricing problem arising in this context is therefore a shortest path problem with some special characteristics: first, it is formulated on a graph with costs on the arcs and prizes on the vertices. This is equivalent to formulating it on a graph with no prizes but with negative cost arcs and possibly negative cost cycles. Therefore the requisite that the path must be elementary does not come for free from cost minimization but it must be explicitly enforced. Second, the pricing problem may be subject to a number of additional restrictions, as mentioned above. These constraints are usually represented as *resource constraints*, since distances, costs, time, and capacities can all be interpreted as resources that are consumed every time a vehicle travels along an arc or visits a customer. Therefore the pricing problem turns out to be a resource constrained elementary shortest path problem (RCESPP). If the underlying graph may have negative cost cycles, the resource constrained elementary shortest path problem is strongly NP-hard [9].

The shortest path problem with resource constraints has been addressed with methods based on the Lagrangian relaxation of the resource constraints; recent results along this research stream are those of Melhorn and Ziegelmann [13] and Dumitrescu and Boland [10]. Their methods are effective when the Lagrangian subproblem is a polynomially solvable shortest path problem, that is when arc costs are non-negative. Dumitrescu and Boland [10] also presented preprocessing and bounding techniques for the RCSPP; although they can be applied to the RCESPP as well, they are especially effective on graphs without negative cost cycles. The first attempt to solve the RCESPP on graphs with negative cost cycles via dynamic programming is due to Beasley and Christofides [2]; their idea was further developed by other authors: see for instance the recent papers by Feillet et al. [11] and Boland, et al. [3]. For a survey on models and algorithms for the RCSPP and the RCESPP we also refer the reader to Irnich and Desaulniers [12].

In this paper we consider dynamic programming algorithms for the resource constrained *elementary* shortest path problem (RCESPP), following the same approach of Feillet et al. [11] and we suggest and evaluate some ideas to improve their performance. In particular we consider *bi-directional search* and its combination with suitable *bounds* to reduce the computing time. We present two different uses of bounds, namely for fathoming unpromising states and for stopping the extension of the non-dominated states. We compare two different ways to achieve this second goal, namely arc bounding and resource bounding. We also describe how to avoid duplicate columns when the RCESPP is solved for pricing purposes in branch-and-price algorithms. Computational results show the effectiveness of bi-directional bounded dynamic programming with respect to the classical mono-directional implementation.

The paper is organized as follows. In Section 2 we provide the definition of the RCESPP and we survey the basic dynamic programming algorithm for its solution. In particular we consider three variants of the RCESPP arising from the set covering reformulation of the capacitated vehicle routing problem (CVRP), the vehicle routing problem with distribution and collection (VRPDC) and the capacitated vehicle routing problem with time windows (CVRPTW). In Section 3 we illustrate our main ideas to improve the dynamic programming algorithm. In Section 4 we provide

the results of our computational experiments on instances derived from Solomon's data-set with up to 100 nodes. Conclusions are outlined in Section 5.

2. Dynamic programming algorithms for the RCESPP

2.1. Problem definition

The RCESPP is the problem of finding the minimum cost elementary path from a node s to a node t of a given graph such that the overall amounts of resources consumed do not exceed some given limits; resources are consumed when visiting nodes or traversing arcs. A graph $\mathcal{G}(\mathcal{V}, \mathcal{A})$ is given: its vertex set $\mathcal{V}$ is made by N vertices representing customers and two special vertices s and t representing the depot. Let $\mathcal{N}$ indicate the set of the customers; hence we have $\mathcal{V} = \mathcal{N} \cup \{s, t\}$. A non-negative integer cost c_{ij} is associated with each arc $(i, j) \in \mathcal{A}$ and these costs 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.

These definitions of the problem are common to all RCESPP versions arising from the different routing problems we consider. Additional constraints must be taken into account, depending on the kind of vehicle routing problem at hand. All these additional constraints are modeled as resource constraints and they will be specified in the remainder.

2.2. Dynamic programming

The basic dynamic programming approach to the RCESPP is based on the algorithm devised by Desrochers et al. [8] for the RCSPP. It is an extension of the Bellman-Ford algorithm with the addition of resource constraints. The algorithm assigns *states* to each vertex: each state of vertex i represents a path from s to i. Each state has an associated resource consumption vector R and each component of R represents the consumption of a different resource along the path. Each state has an associated cost C and the optimal solution is given by the minimum cost state associated with t. Different states associated with the same vertex i correspond to different feasible paths reaching i. Hence states are represented by a *label* of the form (R, C, i). The dynamic programming algorithm repeatedly extends states to generate other states. When a state (R, C, i) is extended to generate another feasible state (R', C', j), the cost and the resource consumption vector of the new state must be computed and those states for which one or more components of R' exceed the available capacity are fathomed. The cost is initialized at 0 at vertex s and it is updated according to the formula

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

where $\lambda_i = -\lambda_0$ if $i = s$ and $\lambda_j = -\lambda_0$ if $j = t$. The resource vector R is initialized and updated according to the specific problem at hand. In addition dominance rules are applied in order to delete dominated states.

2.3. Resource constraints

Hereafter we consider three different specializations of the resource constraints arising from the CVRP, the VRPDC and the CVRPTW. We chose these three problems to validate our approach, because they offer a significant mix of different characteristics of resource types. In the first case there is only one resource, whose consumption depends on the vertices visited. In the second case 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 third case there are two resources, one associated with the vertices visited and the other associated with the arcs traversed. 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 different local constraints at each vertex.

Table 1
An example of a path with simultaneous delivery and collection

Node	s	1	2	3	4	5	t
p	0	2	4	3	1	1	0
d	0	3	3	2	2	1	0
π	0	2	6	9	10	11	11
δ	0	3	6	9	11	12	12
Load	11	10	11	12	11	11	11

Capacity

In the CVRP 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. This constraint is modeled by one resource, representing the amount of available capacity. Let q be the amount of resource consumed. When a vehicle leaves vertex s, it is empty, that is $q = 0$. Every time a vertex i is visited, q is increased by its demand. Each state is represented by a label (q, C, i), where q is the amount of demand satisfied from s to i (included). Each time a state is extended along arc (i, j) from a label (q, C, i) to a label (q', C', j), the resource consumption update rule is

$$q' = q + d_j.$$

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

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 picked-up and delivered at that vertex. We assume $p_i + d_i > 0$. 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 picked-up. The capacity cannot be exceeded anywhere along the path. In the corresponding RCESPP 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 node i indicates the amount of load that the vehicle can deliver after visiting i. Initially Q units are available for this resource and the available resource decreases each time a delivery operation is performed; however it may decrease also after pick-up operations, since 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 when a path is extended along arc (i, j) from a state (π, δ, C, i) to a state (π', δ', C', j), the update rule for the resource consumptions π and δ is:

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

A state (π, δ, C, i) is feasible if and only if $\pi \leq Q$ and $\delta \leq Q$; for the formulae above the latter condition implies the former. A small example is shown in Table 1.

Capacity and time windows

In the CVRPTW a non-negative integer service time θ_i and a time window $[a_i, b_i]$ are associated with each vertex $i \in \mathcal{N}$ and each visited vertex must be reached inside its time window. If the vehicle arrives at i before a_i, it waits until time a_i. The traveling time from i to j is indicated by a positive integer datum v_{ij}. The time elapsed is a consumed resource, monotonically increasing along the path. In the well-known Solomon's instances, which are commonly used as benchmarks for routing algorithms, a capacity constraint is also considered as in the CVRP. Hence in the corresponding RCESPP we need two resources, whose consumption is indicated by τ and q, that are respectively the time and the capacity consumed up to the beginning of service at each vertex. Both of them are initialized at 0 and each time a feasible path is extended along arc (i, j) from a state (τ, q, C, i) to a state (τ', q', C', j) the update rules

for τ and q are:

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

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

2.4. Elementary path constraints

The dynamic programming algorithm described above solves the RCSPP with pseudo-polynomial worst-case time complexity. The same algorithm can be used to solve the RCESPP, where feasible paths are not allowed to contain cycles. For this purpose Beasley and Christofides [2] 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. Hence we consider N resources, whose consumption is indicated by a vector S initialized at 0. When a feasible path is extended along arc (i, j) from a state (S, R, C, i) to a state (S', R', C', j), the update rule for S 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 if and only if $S_k \leq 1\ \forall k \in \mathcal{N}$. Note that S does not keep any information about the order in which the vertices are visited.

2.5. Dominance tests

The effectiveness of the dynamic programming algorithm outlined above heavily relies upon the possibility of fathoming feasible states that cannot lead to an optimal solution. For this purpose suitable dominance tests are always performed when states are extended, so that the algorithm only records non-dominated states. Each state is represented by a label, that is a tuple (S, R, C, i), where S is a vector indicating the vertices already visited, R is a vector indicating the consumption of resources, C is the cost and i is the last reached vertex. The dominance test between two states, or labels, is the following. Let (S_1, R_1, C_1, i) and (S_2, R_2, C_2, i) be the labels of two states associated with vertex i. Then the former dominates the latter if

$$S_1 \leq S_2$$
$$R_1 \leq R_2$$
$$C_1 \leq C_2$$

and at least one of the inequalities is strict.

Extended states are not deleted, because they can be useful to dominate other states not yet generated. This implies to keep all non-dominated states in memory, but allows one to recognize dominations earlier than they would be if extended states were canceled. From our experiments on this trade-off we concluded that keeping all non-dominated states yields better results in terms of computing time and memory occupation.

Dealing with the RCESPP arising as a pricing subproblem in branch-and-price algorithms for the VRPTW, Feillet et al. [11] observed that it is sometimes possible to identify vertices that cannot be visited in any feasible extension of a given state because of the resource limitations. These vertices are called *unreachable*. It is useful to set the consumption of the dummy resources corresponding to the unreachable vertices to 1, as if they had already been visited. This enhancement allows the dynamic programming algorithm to fathom a larger number of states and to reduce the computation time.

This method can be applied to all three versions of the RCESPP considered here. Demands are associated to the vertices and therefore they obviously satisfy the triangle inequality. In Solomon's instances we used for our experiments the cost of each arc is equal to the traveling time between the two endpoints, that is $v_{ij} = c_{ij}$. Hence capacity and time consumptions are positive and satisfy the triangle inequality.

In the case of multiple resources, as for the problem with distribution and collection and the problem with capacity and time windows, all of them are used to identify unreachable vertices.

In Algorithm 1 we report the dynamic programming algorithm of Feillet et al. [11]. The notation we use is the following: for each vertex $i \in \mathcal{V}$, we indicate with Γ_i the set of labels associated with the vertex, with $\overline{\Gamma}_i \subseteq \Gamma_i$ the subset of labels not extended so far and with Δ_i^+ the set of successors of i. E is the set of vertices to be examined; $Extend(l, k)$ is the extension procedure: it extends the state l specified as a first argument to a vertex k specified as a second argument; this procedure checks the resource constraints and produces only feasible states; it also recognizes and marks unreachable nodes as described by Feillet et al. [11]. Finally $EFF(\Gamma, l)$ is the procedure that inserts state l into set $\overline{\Gamma}$ applying the domination rules.

Algorithm 1 RCESPP — Mono-directional dynamic programming

```
// Initialization //
Γ_s ← {(0, 0, 0, s)}
for all i ∈ V \ {s} do
   Γ_i ← ∅
end for
E ← {s}
// Search //
repeat
   // Vertex selection //
   Select i ∈ E
   // Extension //
   for all l_i = (S^i, R^i, C^i, i) ∈ Γ̄_i do
      for all j ∈ Δ_i^+ such that S^i_j = 0 do
         l_j ← Extend(l_i, j)
         Γ_j ← EFF(Γ_j, l_j)
         if Γ̄_j ≠ ∅ then
            E ← E ∪ {j}
         end if
      end for
   end for
   E ← E \ {i}
until E = ∅
```

3. Bounded bi-directional dynamic programming

The dynamic programming algorithm outlined in the previous section generates a number of states rapidly increasing with the size of the problem instance at hand. Every time a label of vertex i is extended, it generates as many other labels as the number of possible successors of i. Therefore in the worst case the number of labels grows exponentially with the number of arcs in the path. States are fathomed only when they are dominated.

We propose here two ideas that work well together: bi-directional dynamic programming and bounding. Bi-directional dynamic programming has been sometimes considered as a useful technique to speed up Dijkstra's algorithm for the computation of an s–t shortest path on a digraph with non-negative arc weights [1]. In the RCESPP, when labels are propagated both forward from s to t and backward from t to s, the algorithm must examine two subsets of states, whose size grows exponentially with the number of arcs in the corresponding forward and backward paths. Due to the exponential dependence on the number of steps, it is intuitive that generating shorter paths may yield a significant advantage in terms of number of states considered, provided that duplicate solutions are avoided. This is precisely the effect of bounding, whose purpose is to limit the length of the paths corresponding to non-dominated states. Hereafter we formally define our bounded bi-directional dynamic programming algorithm.

3.1. Bi-directional search

In bi-directional search states are extended both forward from vertex s to its successors and backward from vertex t to its predecessors. States, recurrence equations and domination rules are symmetrical to those presented above.

We use Γ_i^{fw} and Γ_i^{bw} to indicate the sets of forward and backward labels associated with vertex i. A path from s to t is detected each time a forward state in Γ_i^{fw} and a backward state in Γ_j^{bw} can be feasibly joined through arc (i, j).

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

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

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

Forward and backward paths must be joined together to produce complete s–t paths. Let $(S_1, R_1, C_1, i) \in \Gamma_i^{fw}$ be a forward path and $(S_2, R_2, C_2, j) \in \Gamma_j^{bw}$ be a backward path. When they are joined, the cost of the resulting s–t path is $C_1 - \lambda_i/2 + c_{ij} - \lambda_j/2 + C_2$.

The two paths can be joined subject to certain feasibility conditions on the resources. A feasibility test on dummy resources S imposes that a same vertex cannot be visited by both paths, that is

$$S_1 + S_2 \leq 1 \quad \forall k \in \mathcal{N}.$$

In addition feasibility tests on problem-dependent resources represented by vector R impose that for each resource the consumption in the overall path does not exceed the overall amount of available resource. Hereafter we define the feasibility tests for each specific case considered.

Capacity

The resource consumption q^{bw} in a backward state associated with vertex j represents the amount of demand of customers visited from j (included) to t. Therefore a label $(S, q, C, j) \in \Gamma_j^{bw}$ corresponds to an elementary backward path of cost C, originating at j, terminating at t, visiting the vertices indicated by S and consuming q units of capacity. Initialization and extension of backward labels follow the same rules of forward labels.

The feasibility test on the capacity for joining a forward path (S_1, q_1, C_1, i) with a backward path (S_2, q_2, C_2, j) is

$$q_1 + q_2 \leq Q.$$

Distribution and collection

Two resources, whose consumption is indicated by π and δ, are associated with each backward state. Their meaning, initialization and extension rules are symmetrical to those of forward labels: δ indicates the amount of load delivered between j and t and π 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, π, δ, C, j) to a state $(S', \pi', \delta', C', i)$, the update rule is:

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

A backward path is feasible if and only if $\pi \leq Q$ and $\delta \leq Q$.

The feasibility conditions to join a forward path $(S_1, \pi_1, \delta_1, C_1, i)$ with a backward path $(S_2, \pi_2, \delta_2, C_2, j)$ are:

$$\pi_1 + \pi_2 \leq Q$$
$$\delta_1 + \delta_2 \leq Q.$$

Capacity and time windows

In the case of time windows it is useful to define forward and backward time windows $[a_i^{fw}, b_i^{fw}]$ and $[a_i^{bw}, b_i^{bw}]$ as follows:

$$a_i^{fw} = a_i$$
$$b_i^{fw} = b_i$$

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

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 $T = \max_{i \in \mathcal{V}}\{b_i^{fw} + \theta_i + v_{it}\}$.

The time resource consumption τ in a backward path associated with vertex j represents the time between the departure from j and the arrival at t. The capacity, whose consumption in backward states is indicated by q, follows the same rules as in the RCESPP arising from the CVRP.

When a feasible backward path is extended along arc (i, j) from a state (S, τ, q, C, j) to a state (S', τ', q', C', i), the update rules are:

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

A backward path $(S, \tau, q, C, j) \in \Gamma_j^{bw}$ is feasible if and only if $\tau \leq T - a_j^{bw}$ and $q \leq Q$.

The feasibility conditions to join a forward path $(S_1, \tau_1, q_1, C_1, i)$ with a backward path $(S_2, \tau_2, q_2, C_2, j)$ are:

$$\tau_1 + \theta_i + v_{ij} + \theta_j + \tau_2 \leq T$$
$$q_1 + q_2 \leq Q.$$

3.2. Search strategy

The set of states generated by the dynamic programming algorithm can be explored according to different search strategies, and the order in which the states are extended may be very important for the effectiveness of the overall algorithm. In label-correcting algorithms, like those of Desrosiers et al. [8] and Feillet et al. [11], 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.

Label-setting algorithms have been proposed (see for instance Desrochers and Soumis [6]) but they require a 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 [7].

In order to have a more significant comparison with the algorithm of Feillet et al. [11], we developed label-correcting algorithms, where for each vertex the states are ordered by non-decreasing resource consumption. 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. When examining a vertex, the bi-directional algorithm extends both forward and backward states associated with it.

In bi-directional algorithms we keep all non-dominated states in memory, in same way described in Section 2.5 for the mono-directional algorithm.

In Algorithm 2 we illustrate the bi-directional dynamic programming algorithm. The notation is analogous to the one used in Algorithm 1. The extension functions recognize and mark unreachable vertices as in the mono-directional case. Function *Join* is explained in the remainder.

3.3. Bounding

In our algorithms bounding is used for two different purposes: (i) to recognize and fathom the states that cannot produce optimal solutions and (ii) to stop the extension of forward and backward paths in order to reduce the number of states generated, while preserving the guarantee that the optimal solution will be found. Without this latter limitation the bi-directional algorithm would simply produce twice as many labels, compared to the mono-directional one. The

Algorithm 2 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**
  $\Gamma_i^{fw} \leftarrow \emptyset$
end for
for all $i \in \mathcal{V} \setminus \{t\}$ **do**
  $\Gamma_i^{bw} \leftarrow \emptyset$
end for
$E \leftarrow \{s, t\}$
// Search //
repeat
  // Vertex selection //
  Select $i \in E$
  // Forward extension //
  for all $l_i = (S^i, R^i, C^i, i) \in \overline{\Gamma}_i^{fw}$ **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 $\overline{\Gamma}_j^{fw} \neq \emptyset$ **then**
        $E \leftarrow E \cup \{j\}$
      end if
    end for
  end for
  // Backward extension //
  for all $l_i = (S^i, R^i, C^i, i) \in \overline{\Gamma}_i^{bw}$ **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 $\overline{\Gamma}_k^{bw} \neq \emptyset$ **then**
        $E \leftarrow E \cup \{k\}$
      end if
    end for
  end for
  $E \leftarrow E \setminus \{i\}$
until $E = \emptyset$
// Join between forward and backward paths //
Join

idea is that we can stop extending a path in one direction when we have the guarantee that the remaining part of the path will be generated in the other direction and therefore no optimal solution will be lost.

Hereafter we first present the bounding technique we use for fathoming unpromising states (Section 3.3.1) and then two different ways of bounding the paths to stop their extension (Sections 3.3.2 and 3.3.3): one is based on the number of arcs and another is based on the consumption of a selected resource.

First we introduce the necessary notation. We have so far indicated with S and R the two vectors representing the vertices visited and the resource consumptions in the labels of the states. Here below we indicate with $\mathcal{S}$ the *set* of visited vertices corresponding to the binary vector S. Let $\mathcal{R}$ be the set of resources and $r \in \mathcal{R}$ be a generic resource, so that R_r indicates the amount of resource r consumed by a path reaching vertex i with label (S, R, C, i). Let us indicate with A_r the overall amount available for each resource $r \in \mathcal{R}$. We first consider forward paths and for a given resource

r we define $m_r(S, i, j)$ to be a lower bound to the consumption of resource r when vertex j is added to the forward path corresponding to (S, R, C, i) for each vertex $j \notin \mathcal{S}$. In particular we define $m_r(S, i, j) = \min_{k \notin \mathcal{S}\setminus\{i\}}\{w^r_{kj}\}$, where w^r_{kj} is the consumption of resource r when the vehicle traverses arc (k, j) and visits vertex j. We also define $m_r(S, i, 0) = \min_{k \notin \mathcal{S}\setminus\{i\}}\{w^r_{k0}\}$, where index 0 represents depot t. Finally we define u_j as an upper bound to the prize collected when visiting vertex $j \in \mathcal{N}$ along the path, that is

$$u_j = \lambda_j - \min_{k \notin \mathcal{S}\setminus\{i\}}\{c_{kj}\}$$

and

$$u_0 = \lambda_0/2 - \min_{k \notin \mathcal{S}\setminus\{i\}}\{c_{k0}\}$$

where 0 represents depot t.

Analogous definitions apply to backward paths: if (S, R, C, i) is the label of a backward path, we define $m_r(S, i, j) = \min_{k \notin \mathcal{S}\setminus\{i\}}\{w^r_{jk}\}$ and $m_r(S, i, 0) = \min_{k \notin \mathcal{S}\setminus\{i\}}\{w^r_{0k}\}$, where index 0 represents depot s. We also have

$$u_j = \lambda_j - \min_{k \notin \mathcal{S}\setminus\{i\}}\{c_{jk}\}$$

and

$$u_0 = \lambda_0/2 - \min_{k \notin \mathcal{S}\setminus\{i\}}\{c_{0k}\}$$

where 0 represents depot s.

3.3.1. Bounding for fathoming

For each newly generated non-dominated state (S, R, C, i), we compute an upper bound $\overline{P}$ to the following optimization subproblem, in which each variable y_j indicates whether node j is visited along the path.

$$\begin{aligned}
&\text{maximize} \sum_{j \in \mathcal{N}\setminus\mathcal{S}} u_j y_j + u_0 \\
&\text{subject to } R_r + \sum_{j \in \mathcal{N}\setminus\mathcal{S}} m_r(S, i, j) y_j + m_r(S, i, 0) \le A_r \quad \forall r \in \mathcal{R} \\
&y_j \in \{0, 1\} \qquad \forall j \in \mathcal{N} \setminus \mathcal{S}.
\end{aligned}$$

In particular we consider one constraint (resource) at a time, we solve the linear relaxation of the resulting binary knapsack problem instance and we keep as $\overline{P}$ the minimum of the $|\mathcal{R}|$ upper bounds obtained in this way. The value $\overline{P}$ is an upper bound to the maximum gain (prizes collected minus costs paid) that the vehicle can achieve by completing the path. Hence the state under examination can be fathomed if $C - \overline{P} \ge UB$, where UB is the incumbent upper bound, that is the value of a known feasible solution.

3.3.2. Arc bounding

For each state (S, R, C, i) we can compute an upper bound on the number of arcs that can be added to the corresponding path without exceeding the resource constraints. This is achieved by solving the following multi-knapsack problem:

$$\begin{aligned}
&\text{maximize} \sum_{j \in \mathcal{N}\setminus\mathcal{S}} y_j + 1 \\
&\text{subject to } R_r + \sum_{j \in \mathcal{N}\setminus\mathcal{S}} m_r(S, i, j) y_j + m_r(S, i, 0) \le A_r \quad \forall r \in \mathcal{R} \\
&y_j \in \{0, 1\} \quad \forall j \in \mathcal{N} \setminus \mathcal{S}.
\end{aligned}$$

As before we consider one resource at a time, we optimize the resulting knapsack problem instance (this is done in polynomial time owing to the particular objective function) and we keep the minimum upper bound obtained. This gives an upper bound on the maximum number of vertices that can be visited along the path after the last reached

Table 2
Bi-directional construction of the solution $(s, 1, 2, 3, 4, 5, t)$

Node	s	1	2	3	4	5	t
p	0	2	4	3	1	1	0
d	0	3	3	2	2	1	0
π^{fw}	0	2	6				
δ^{fw}	0	3	6				
ρ^{fw}	0	5	12				
π^{bw}			10	6	3	1	0
δ^{bw}			8	5	3	1	0
ρ^{bw}			18	11	6	2	0

Table 3
RCESPP with capacity — 50 vertices

Instance	Mono-directional		Arc bounding		Resource bounding	
	Labels	Time	Labels	Time	Labels	Time
c_50_01	30	0.00	30	0.00	30	0.00
c_50_02	104	0.00	121	0.00	104	0.00
c_50_03	311	0.01	433	0.01	277	0.00
c_50_04	885	0.05	1 012	0.06	812	0.03
c_50_05	2 593	0.28	2 864	0.32	1 978	0.04
c_50_06	8 707	2.47	8 304	0.94	8 185	0.57
c_50_07	30 973	26.30	20 512	5.21	10 694	0.78
c_50_08	111 814	287.50	84 142	33.97	43 525	14.99
c_50_09	393 680	3240.86	148 116	96.62	51 467	19.66
c_50_10			504 944	981.46	211 951	298.45
r_50_01	40	0.00	40	0.00	40	0.00
r_50_02	135	0.00	154	0.00	129	0.01
r_50_03	312	0.00	409	0.01	296	0.01
r_50_04	652	0.04	922	0.04	616	0.02
r_50_05	1 345	0.09	1 394	0.09	1 224	0.04
r_50_06	2 868	0.24	2 785	0.20	2 349	0.08
r_50_07	6 296	0.77	5 614	0.64	4 269	0.13
r_50_08	14 226	2.91	11 308	2.14	7 731	0.32
r_50_09	32 561	12.25	25 444	7.85	13 638	0.97
r_50_10	73 456	52.40	58 948	17.79	22 709	2.37
rc_50_01	21	0.00	21	0.00	21	0.00
rc_50_02	87	0.00	76	0.00	87	0.00
rc_50_03	164	0.00	228	0.01	136	0.00
rc_50_04	302	0.01	381	0.01	300	0.01
rc_50_05	511	0.02	666	0.03	421	0.01
rc_50_06	876	0.05	1 011	0.05	865	0.03
rc_50_07	1 331	0.10	1 202	0.08	1 006	0.03
rc_50_08	2 038	0.18	2 009	0.10	1 827	0.08
rc_50_09	3 115	0.35	2 950	0.15	2 026	0.09
rc_50_10	4 846	0.67	4 184	0.22	3 721	0.18

vertex i. If this number is less than $|\mathcal{S}|$, then the path has reached its *half-way point* and the extension is stopped. The remaining part of the path will be generated in all possible ways as a set of paths in the other direction, owing to the bi-directional dynamic programming algorithm.

3.3.3. Resource bounding

Another way to stop the extension of paths is to select a *critical resource*, whose consumption is monotone along the paths, and to stop the extension of the states in which at least half of the available amount of that resource has

Table 4
RCESPP with capacity — 100 vertices

Instance	Mono-directional		Arc bounding		Resource bounding	
	Labels	Time	Labels	Time	Labels	Time
c_100_01	55	0.00	106	0.00	55	0.00
c_100_02	205	0.01	237	0.01	205	0.01
c_100_03	640	0.09	720	0.09	579	0.04
c_100_04	2 136	0.41	2 106	0.38	2 093	0.18
c_100_05	7 056	2.49	5 898	0.78	4 722	0.26
c_100_06	26 135	21.87	24 552	7.50	22 505	4.42
c_100_07	116 247	327.42	60 082	28.44	30 871	5.94
c_100_08			282 184	502.66	171 703	178.04
c_100_09					217 699	226.04
c_100_10						
r_100_01	163	0.00	96	0.00	150	0.00
r_100_02	1 076	0.09	1 088	0.05	972	0.05
r_100_03	5 106	1.24	4 962	0.60	4 285	0.41
r_100_04	25 613	19.59	20 356	5.42	17 054	3.14
r_100_05	133 007	417.56	106 610	106.44	72 202	32.04
r_100_06			732 786	2332.08	270 466	371.13
r_100_07						
r_100_08						
r_100_09						
r_100_10						
rc_100_01	21	0.00	64	0.00	21	0.00
rc_100_02	257	0.01	196	0.01	251	0.01
rc_100_03	705	0.06	856	0.09	699	0.03
rc_100_04	1 857	0.28	2 506	0.32	1 823	0.14
rc_100_05	5 024	1.20	7 748	1.31	4 527	0.38
rc_100_06	14 260	5.86	23 662	4.86	11 400	1.27
rc_100_07	40 375	31.40	68 422	29.35	24 787	4.09
rc_100_08	111 591	181.25	166 649	137.11	55 665	15.51
rc_100_09	299 056	1086.05	227 468	690.88	110 506	53.62
rc_100_10					230 054	209.30

been consumed. Our stopping criterion requires that a positive consumption of the critical resource is associated with every arc. Hereafter we describe how we have defined the critical resource for each different vehicle routing problem.

Capacitated VRP

The critical resource in this case is capacity. Forward and backward states are extended only if their associated resource consumption value q is less than $Q/2$, where Q is the vehicle capacity.

VRP with distribution and collection

In this case there are two resources; we consider as a critical resource ρ the sum of the resource consumptions $\pi + \delta$ and we extend only those states for which $\pi + \delta < Q$.

Capacitated VRP with time windows

In this last case we consider time as the critical resource and we extend only states for which $\tau < T/2$.

Note that the arc bounding technique can be considered as a special case of resource bounding, in which the critical resource is the number of allowed visits and its available amount is recomputed in every state.

3.4. Solutions uniqueness

Another issue to be considered comes from the need of generating many different columns with negative reduced cost when we solve the RCESPP as a pricing problem in a branch-and-price framework. The bounded bi-directional

Table 5
RCESPP with distribution and collection — 50 vertices

Instance	Mono-directional		Arc bounding		Resource bounding	
	Labels	Time	Labels	Time	Labels	Time
c_50_01	25	0.00	26	0.00	25	0.00
c_50_02	191	0.00	168	0.04	85	0.00
c_50_03	1 127	0.01	562	0.06	188	0.01
c_50_04	4 788	0.19	1 852	0.09	632	0.02
c_50_05	21 420	4.30	4 876	1.28	1 535	0.07
c_50_06	88 706	79.75	14 381	4.41	5 507	0.27
c_50_07	346 218	1201.05	26 022	7.95	10 578	0.69
c_50_08			55 462	35.46	33 588	7.23
c_50_09			140 364	165.21	56 812	16.95
c_50_10			335 670	672.28	181 699	140.59
r_50_01	51	0.00	58	0.00	51	0.00
r_50_02	207	0.01	204	0.01	116	0.01
r_50_03	633	0.01	498	0.03	298	0.01
r_50_04	1 910	0.04	1 126	0.04	585	0.02
r_50_05	5 338	0.23	2 096	0.10	1 222	0.05
r_50_06	13 925	1.53	3 986	0.32	2 345	0.09
r_50_07	34 947	9.65	7 186	0.98	4 312	0.22
r_50_08	83 238	52.00	14 136	1.63	7 772	0.50
r_50_09	188 997	257.86	22 386	5.25	13 866	1.20
r_50_10	410 572	1695.94	38 107	8.19	23 788	3.03
rc_50_01	23	0.00	23	0.00	23	0.00
rc_50_02	96	0.00	92	0.00	58	0.00
rc_50_03	231	0.01	202	0.01	111	0.01
rc_50_04	511	0.01	416	0.02	247	0.01
rc_50_05	1 104	0.02	790	0.03	377	0.02
rc_50_06	2 080	0.07	1 306	0.06	639	0.02
rc_50_07	3 797	0.19	1 915	0.14	967	0.03
rc_50_08	6 807	0.63	2 641	0.19	1 463	0.05
rc_50_09	12 367	2.17	3 633	0.36	2 119	0.09
rc_50_10	22 823	7.55	5 018	0.60	3 201	0.14

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 resource constraints are 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 solution is obtainable 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. If only the optimal solution is sought, these duplicates are discarded with no additional computational effort, when they are evaluated, since they have the same cost. But if one needs to store in some data structure all columns with negative reduced cost, the duplicate columns cannot be discarded on the basis of their cost and their identification may be computationally expensive.

For this reason we have devised an additional test, represented by the function *HalfWay*. The meaning of this test is that 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 ρ^{fw} and ρ^{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 $\phi = |\rho^{fw} - \rho^{bw}|$ 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 $\rho^{fw} < \rho^{bw}$ and with the previous position if $\rho^{fw} > \rho^{bw}$. In the case of a tie between two positions for which ϕ is minimum, we choose the one with $\rho^{fw} > \rho^{bw}$. This test guarantees that each s–t path is generated only once.

Algorithm 3 refers to the procedure *Join* of the bi-directional bounded dynamic programming algorithm, when it is solved as a pricing problem and hence duplications must be avoided. We use the following terminology:

Algorithm 3 RCESPP — Bi-directional dynamic programming: *Join*

```
for all i ∈ V do
  if ψ_i^fw − λ_i/2 + min_{j∈N\{i}}{c_ij} − max_{j∈N}{λ_j/2} + ψ^bw < UB then
    for all l_i = (S^fw, R^fw, C^fw, i) ∈ Γ_i^fw do
      if C^fw − λ_i/2 + min_{j∈N\{i}}{c_ij} − max_{j∈N}{λ_j/2} + ψ^bw < UB then
        for all j ∈ V do
          if C^fw − λ_i/2 + c_ij − λ_j/2 + ψ_j^bw < UB then
            for all l_j = (S^bw, R^bw, C^bw, j) ∈ Γ_j^bw do
              if C^fw − λ_i/2 + c_ij − λ_j/2 + C^bw < UB then
                if Feasible(l_i, l_j) AND HalfWay(l_i, l_j) then
                  Save(l_i, l_j)
                end if
              end if
            end for
          end if
        end for
      end if
    end for
  end if
end for
```

Feasible(l_i, l_j) checks the resource compatibility of states l_i and l_j according to problem-dependent rules; *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 defined above; *Save*(l_i, l_j) saves the solution obtained from the two states l_i and l_j.

When implementing the *Join* procedure it is possible to avoid the evaluation of all pairs of forward and backward labels, exploiting some bounds on the label costs, as shown in Algorithm 3. We indicate with ψ^{bw} the minimum cost among all backward labels, with ψ_i^{fw} the minimum cost among all labels in Γ_i^{fw} and with ψ_j^{bw} the minimum cost among all labels in Γ_j^{bw}.

To give an example of how the function *HalfWay* works, we further elaborate on the example presented in Table 1, referred to the VRPDC. Table 2 shows how the same path of Table 1 can be constructed by the bi-directional search algorithm. We have added a superscript fw or bw to identify forward and backward resource consumptions.

The forward extension stops at vertex 2, because the critical resource consumption is 12, that is Q. The backward extension stops at vertex 2, because the critical resource consumption is 18, that is greater than Q. The path can be generated by joining the forward label of vertex 1 with the backward label of vertex 2, yielding an unbalance $\Phi = |\rho^{fw}(1) - \rho^{bw}(2)| = |5 - 18| = 13$, as well as by joining the forward label of vertex 2 with the backward label of vertex 3, yielding an unbalance $\Phi = |\rho^{fw}(2) - \rho^{bw}(3)| = |12 - 11| = 1$. Therefore the function *HalfWay* returns "false" in the former case and "true" in the latter.

4. Computational results

4.1. Instances

We derived our test instances from the well-known Solomon's data-set of VRPTW instances. 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 demands; the instances differ only for the time windows.

When solving the RCESPP with capacity we considered one instance taken from each one of the three Solomon's data-sets, we kept the original customer locations and demands 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. In both cases the vehicle capacity varies from 10 to 100 with an increasing step of 10.

Table 6
RCESPP with distribution and collection — 100 vertices

Instance	Mono-directional		Arc bounding		Resource bounding	
	Labels	Time	Labels	Time	Labels	Time
c_100_01	47	0.00	48	0.00	47	0.00
c_100_02	382	0.00	363	0.01	166	0.00
c_100_03	2 415	0.08	1 244	0.06	381	0.02
c_100_04	13 009	1.42	3 950	0.64	1 426	0.12
c_100_05	83 462	49.91	12 982	0.89	3 689	0.22
c_100_06	520 592	1999.5700	34 342	3.62	14 800	2.04
c_100_07			80 098	14.82	29 977	5.10
c_100_08			209 776	89.44	123 907	82.29
c_100_09			545 612	523.66	229 386	218.82
c_100_10						
r_100_01	245	0.01	253	0.00	153	0.00
r_100_02	3 688	0.21	1 986	0.08	994	0.06
r_100_03	43 242	20.67	11 622	1.05	4 706	0.55
r_100_04	409 513	1806.75	50 204	19.34	18 995	4.44
r_100_05			201 088	144.00	83 158	47.10
r_100_06			704 226	1640.25	351 405	686.55
r_100_07						
r_100_08						
r_100_09						
r_100_10						
rc_100_01	72	0.00	67	0.01	47	0.00
rc_100_02	401	0.00	501	0.01	229	0.01
rc_100_03	1 950	0.07	1 422	0.34	642	0.04
rc_100_04	8 290	0.70	4 640	1.17	1 776	0.15
rc_100_05	32 216	8.19	10 988	2.25	4 331	0.49
rc_100_06	117 793	98.23	26 644	6.18	10 794	1.54
rc_100_07	418 620	1109.8000	53 871	19.13	24 657	5.83
rc_100_08			131 416	52.61	55 131	31.83
rc_100_09			223 042	225.25	116 239	87.50
rc_100_10					242 383	688.90

For the RCESPP with distribution and collection we kept the original delivery requests and we derived the pick-up requests as follows: $p_i = \lfloor 0.8 d_i \rfloor$ if i is odd and $p_i = \lfloor 1.2 d_i \rfloor$ if i is even. We varied the capacity of the vehicle as in the previous case.

Finally, for the RCESPP with capacity 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; we kept the original starting times of the time windows, a_i, and we set the end times as follows: $b_i = a_i + (1 + \gamma)\theta_i$ for $\gamma = 0.25k$ and $k = 0, \ldots, 24$, where θ_i is the original service time at vertex i.

We generated the dual variables λ_i as random integer variables uniformly distributed in $\{0, \ldots, 20\}$, as proposed by [11], in order to have a reasonable number of negative arcs. We rounded up all the Euclidean distances between customers to integer values.

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

4.2. Preprocessing

Dumitrescu and Boland [10] proposed very effective preprocessing techniques for the RCSPP, which is equivalent to the RCESPP with no negative cost cycles. These techniques are mainly based on the computation of a lower bound on the resource consumption that is necessary to complete a partial solution returning to the depot. There are two

Table 7
ESPPRC with capacity and time windows — 50 vertices

Instance	Mono-directional		Arc bounding		Resource bounding	
	Labels	Time	Labels	Time	Labels	Time
c101_50	524	0.02	678	0.02	500	0.02
c102_50	4 548	0.93	3 920	0.30	2 747	0.22
c103_50	106 795	393.47	41 106	26.76	27 656	13.73
c104_50						
c105_50	609	0.03	726	0.03	603	0.02
c106_50	565	0.03	686	0.03	509	0.02
c107_50	652	0.04	827	0.04	661	0.03
c108_50	1 019	0.07	1 114	0.07	924	0.04
c109_50	2 255	0.22	2 378	0.23	2 177	0.20
r101_50	166	0.00	274	0.01	189	0.00
r102_50	663	0.03	982	0.04	642	0.03
r103_50	2 546	0.16	3 352	0.19	1 950	0.11
r104_50	32 697	10.55	30 228	4.72	10 592	1.22
r105_50	344	0.01	398	0.01	368	0.01
r106_50	970	0.04	1 294	0.06	882	0.04
r107_50	3 457	0.24	4 334	0.26	2 349	0.16
r108_50	34 460	12.36	32 640	5.62	11 253	1.33
r109_50	683	0.03	768	0.03	741	0.02
r110_50	2 003	0.12	2 254	0.13	1 769	0.10
r111_50	2 571	0.19	3 041	0.19	2 202	0.15
r112_50	4 552	0.39	5 213	0.39	3 760	0.32
rc101_50	386	0.01	394	0.01	357	0.00
rc102_50	1 368	0.04	1 101	0.04	1 020	0.03
rc103_50	4 788	0.42	4 900	0.27	3 448	0.18
rc104_50	12 805	3.47	12 584	1.69	8 926	0.84
rc105_50	1 208	0.03	976	0.03	1 000	0.03
rc106_50	1 194	0.04	957	0.03	999	0.03
rc107_50	5 380	0.31	3 663	0.18	3 479	0.15
rc108_50	12 780	2.29	11 465	0.62	8 671	0.53

reasons for which such techniques are not effective in our case. First, the graphs we have considered are complete: hence every partial path can be closed by an arc directly reaching the depot. Second, in the CVRP and VRPDC the resource consumption is associated with the nodes, not with the arcs; therefore the depot can always be reached with no resource consumption; in the CVRPTW, where the resource consumption is associated with the arcs, the time resource is always enough to complete any feasible path, by definition of the maximum arrival time T. Therefore we could obtain no simplification of our instances through preprocessing.

4.3. Results

Tables 3–10 report on the experimental comparison between the mono-directional dynamic programming algorithm, the bi-directional algorithm with arc bounding and the bi-directional algorithm with resource bounding. For each algorithm we report the total number of non-dominated states that are in memory at the end of the extension procedure and the time needed to compute the optimal path. Empty cells mean that the solution has not been computed within the time limit of one hour.

Capacity

Results reported in Tables 3 and 4 show that the bi-directional algorithm with resource bounding outperforms the other two in all instances, where the computing time is significant (greater than 0.01 s). For the loosely constrained instances it reduces the computing time by one order of magnitude and it reduces significantly the number of non-dominated states. The bi-directional algorithm with arc bounding outperforms the mono-directional algorithm when

Table 8
RCESPP with capacity and time windows — 100 vertices

Instance	Mono-directional		Arc bounding		Resource bounding	
	Labels	Time	Labels	Time	Labels	Time
c101_100	994	0.16	1 394	0.19	1 039	0.14
c102_100	18 126	22.53	16 301	8.65	9 759	3.97
c103_100			195 398	527.36	95 138	148.50
c104_100						
c105_100	1 149	0.23	1 514	0.27	1 256	0.22
c106_100	1 448	0.37	1 849	0.41	1 502	0.33
c107_100	1 225	0.31	1 691	0.37	1 378	0.30
c108_100	2 094	0.64	2 449	0.71	2 109	0.58
c109_100	4 739	2.03	5 326	2.12	4 816	1.90
r101_100	746	0.04	1 005	0.05	765	0.05
r102_100	36 969	49.66	35 037	15.79	13 021	4.51
r103_100			418 229	1053.52	75 599	105.77
r104_100					349 866	1278.57
r105_100	2 191	0.21	1 947	0.20	1 679	0.17
r106_100	52 182	126.21	52 210	39.18	19 411	11.03
r107_100			467 533	1488.83	83 422	141.20
r108_100					312 346	1094.81
r109_100	6 389	1.35	4 631	1.06	4 417	0.87
r110_100	39 042	47.09	38 200	21.91	22 744	12.71
r111_100			145 671	187.36	44 094	39.38
r112_100					269 888	1019.10
rc101_100	1 196	0.09	1 552	0.01	1 038	0.08
rc102_100	8 268	1.73	8 612	1.59	5 209	0.82
rc103_100	76 457	100.67	81 085	56.53	22 618	9.87
rc104_100			878 304	2946.87	137 013	202.25
rc105_100	3 253	0.45	3 842	0.55	3 288	0.41
rc106_100	3 130	0.44	3 426	0.48	3 124	0.37
rc107_100	14 224	3.56	152 257	3.64	10 651	2.17
rc108_100	57 637	46.54	69 200	31.45	39 880	20.12

the resource capacity grows, while for tightly constrained instances it produces more labels. For instances with 100 vertices the memory space and the computing time grow very quickly for all three algorithms. However the resource bounded bi-directional algorithm solves more and larger instances than the mono-directional algorithm and it reduces the computing time by one order of magnitude. The computational results show that the bi-directional algorithm is a bit faster than the mono-directional one also when it considers more labels (see for instance the computing time for r_100_02 and rc_100_07); this is due to the implementation of the *Join* procedure illustrated in Section 3.4, which allows us to discard entire subsets of labels.

Distribution and collection

When solving the RCESPP with distribution and collection we obtained results similar to those above: they are reported in Tables 5 and 6. The resource bounded bi-directional algorithm solved all instances with 50 vertices in less than 150 s and it failed to solve 5 instances with 100 vertices within one hour, while the bi-directional algorithm with arc bounding failed to solve 6 instances. On this problem also the bi-directional algorithm with arc bounding dominates the mono-directional one in almost all cases.

Capacity and time windows

All but one of Solomon's instances with 50 and 100 vertices were solved by the resource bounded bi-directional algorithm as reported in Tables 7 and 8. Instance c_104 is nasty for all algorithms. The irregular growth in the number of states and computing time is due to the local nature of the time windows constraints. The superiority of both bounded bi-directional algorithms is quite evident and systematic for loosely constrained instances. For some tightly

Table 9
RCESPP with capacity and time windows — Instance c_104, 50 vertices

Instance	Mono-directional		Arc bounding		Resource bounding	
	Labels	Time	Labels	Time	Labels	Time
c104_50_01	96	0.00	150	0.00	100	0.00
c104_50_02	166	0.00	250	0.01	157	0.00
c104_50_03	246	0.01	355	0.01	229	0.00
c104_50_04	257	0.01	357	0.01	235	0.01
c104_50_05	271	0.01	388	0.01	244	0.01
c104_50_06	370	0.01	412	0.01	323	0.01
c104_50_07	614	0.02	636	0.01	532	0.01
c104_50_08	730	0.04	737	0.04	637	0.04
c104_50_09	871	0.05	865	0.05	772	0.05
c104_50_10	991	0.07	935	0.07	841	0.06
c104_50_11	1 751	0.11	1 621	0.12	1 479	0.11
c104_50_12	2 664	0.23	2 470	0.24	2 349	0.24
c104_50_13	4 158	0.48	3 892	0.47	3 827	0.46
c104_50_14	4 495	0.58	4 154	0.58	4 081	0.53
c104_50_15	6 257	0.93	5 695	0.90	5 556	0.76
c104_50_16	10 426	2.55	9 595	2.25	9 463	2.16
c104_50_17	20 072	6.01	18 728	5.49	18 631	5.15
c104_50_18	23 086	7.52	21 901	6.99	21 792	6.63
c104_50_19	27 539	11.12	25 851	10.51	25 698	9.42
c104_50_20	39 652	27.11	37 009	24.72	36 875	23.86
c104_50_21	89 920	97.87	82 258	82.11	82 108	76.42
c104_50_22	112 830	136.46	106 349	127.05	106 183	115.17
c104_50_23	135 902	189.24	127 474	166.10	127 235	157.86
c104_50_24	170 507	350.34	160 154	311.12	159 960	302.80
c104_50_25			338 839	1345.13	335 617	1166.70

constrained instances the bi-directional algorithms produce more labels than the mono-directional one, while the computing times are almost the same.

Tightness of the constraints

Tables 9 and 10 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 dramatically. Also in these experiments the superiority of bounded bi-directional algorithms is clear. For tightly constrained instances the bi-directional algorithm with arc bounding produces more labels than the mono-directional one. As expected, the arc bounding technique is useful only when the optimal path is made of a significant number of arcs.

5. Conclusions

In this paper we have proposed an improved technique, bounded bi-directional dynamic programming, for the exact optimization of the resource constrained elementary shortest path problem. We have shown how bounded bi-directional dynamic programming can be applied to the RCESPP with one or more resource constraints, interacting or independent resources, local or global constraints, and resource consumptions depending on visited vertices or traversed arcs. Our experiments show that bounded bi-directional dynamic programming definitely outperforms the mono-directional algorithm commonly used and reported in the literature.

The long-term goal of this research is the effective solution of vehicle routing problems with additional constraints through branch-and-price algorithms, where the RCESPP arises as a pricing subproblem. Future developments include the comparison of this approach with that based on state space relaxation and the application of these ideas to the development of more effective algorithms for the exact optimization of vehicle routing problems with additional constraints.

Table 10
RCESPP with capacity and time windows — Instance c_104, 100 vertices

Instance	Mono-directional		Arc bounding		Resource bounding	
	Labels	Time	Labels	Time	Labels	Time
c104_100_01	199	0.00	357	0.01	205	0.01
c104_100_02	299	0.02	456	0.02	286	0.02
c104_100_03	447	0.03	606	0.03	415	0.03
c104_100_04	495	0.05	674	0.06	463	0.05
c104_100_05	510	0.05	717	0.06	477	0.06
c104_100_06	698	0.07	804	0.08	603	0.06
c104_100_07	1 121	0.13	1 181	0.15	975	0.12
c104_100_08	1 416	0.23	1 439	0.24	1 232	0.22
c104_100_09	1 685	0.36	1 670	0.35	1 475	0.34
c104_100_10	1 882	0.44	1 864	0.44	1 645	0.42
c104_100_11	3 105	0.71	2 967	0.69	2 738	0.65
c104_100_12	5 122	1.43	4 816	1.41	4 595	1.36
c104_100_13	8 168	2.74	7 604	2.61	7 437	2.63
c104_100_14	9 244	3.74	8 786	3.59	8 579	3.58
c104_100_15	12 088	5.22	11 271	4.88	11 053	4.86
c104_100_16	20 841	11.40	19 449	10.65	19 267	10.42
c104_100_17	42 948	28.64	39 398	25.09	39 260	24.91
c104_100_18	56 769	45.05	53 959	41.56	53 823	41.34
c104_100_19	69 921	65.26	66 598	59.75	66 373	58.19
c104_100_20	96 971	125.26	92 308	116.27	92 042	112.48
c104_100_21			198 953	355.52	198 464	350.88
c104_100_22			324 494	750.19	318 067	740.42
c104_100_23			449 425	1336.48	441 254	1238.62
c104_100_24			555 030	2144.44	554 831	2087.82
c104_100_25						

Acknowledgements

We thank Dominique Feillet for kindly providing his code and three anonymous referees for their comments. We acknowledge the support of ACSU — Associazione Cremasca Studi Universitari — to the Operations Research Laboratory of our department, where this research was done.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows, Prentice Hall, 1993.
[2] J.E. Beasley, N. Christofides, An algorithm for the resource constrained shortest path problem, Networks 19 (1989) 379–394.
[3] N. Boland, J. Dethridge, I. Dumitrescu, Accelerated label setting algorithms for the elementary resource constrained shortest path problem, Operations Research Letters 34 (2006) 58–68.
[4] J. Bramel, D. Simchi-Levi, Set-covering-based algorithms for the capacitated VRP, in: P. Toth, D. Vigo (Eds), The Vehicle Routing Problem, in: SIAM Monographs on Discrete Mathematics and Applications, Philadelphia, 2002.
[5] G. Desaulniers, J. Desrosiers, I. Ioachim, M.M. Solomon, F. Soumis, D. Villeneuve, A unified framework for deterministic time constrained Vehicle Routing and crew scheduling Problems, in: T.G. Crainic, G. Laporte (Eds.), Fleet Management and Logistics, Kluwer, Boston, 1998, pp. 57–93.
[6] M. Desrochers, F. Soumis, A generalized permanent labelling algorithm for the shortest path problem with time windows, INFOR 26 (1988) 191–212.
[7] J. Desrosiers, Y. Dumas, M. Solomon, F. Soumis, Time constrained routing and scheduling in Network Routing, in: M.O. Ball et al. (Eds.), Handbooks in Operations Research and Management Science, Elsevier Science, 1995.
[8] J. Desrosiers, P. Pelletier, F. Soumis, Plus court chemin avec contraintes d'horaires, RAIRO 17 (1983) 357–377.
[9] M. Dror, Note on the complexity of the shortest path models for column generation in VRPTW, Operations Research 42 (1994) 977–978.
[10] I. Dumitrescu, N. Boland, Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem, Networks 42 (2003) 135–153.
[11] D. Feillet, P. Dejax, M. Gendreau, 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.
[12] S. Irnich, G. Desaulniers, Shortest path problems with resource constraints, Cahier du GERAD G-2004-11, Université de Montréal, 2004.
[13] K. Mehlhorn, M. Ziegelmann, Resource constrained shortest paths, Lecture Notes in Computer Science 1879 (2000) 326–337.