

A branch-and-price approach to the vehicle routing problem with simultaneous distribution and collection

Mauro Dell'Amico*

Dipartimento di Scienze e Metodi per l'Ingegneria
Università di Modena e Reggio Emilia

Giovanni Righini, Matteo Salani
Dipartimento di Tecnologie dell'Informazione
Università degli Studi di Milano

March 2005 - Revised version

Abstract. The vehicle routing problem with simultaneous distribution and collection (VRPSDC) is the variation of the capacitated vehicle routing problem that arises when the distribution of goods from a depot to a set of customers and the collection of waste from the customers to the depot must be performed by the same vehicles of limited capacity and the customers can be visited in any order. We study how the branch-and-price technique can be applied to the solution of this problem and in particular we compare two different ways of solving the pricing subproblem: exact dynamic programming and state space relaxation. By applying a bi-directional search we experimentally prove its effectiveness in solving the subproblem. We also devise suitable branching strategies for both the exact and the relaxed approach and we report on an extensive set of computational experiments on benchmark instances with both simple and composite demands.

Keywords: reverse logistics, vehicle routing, branch-and-price, dynamic programming.

*Corresponding author (dellamico@unimore.it)

Introduction

Environmental concerns in recent years have raised the need for efficient recycling and reuse of goods and packages. As a result *reverse logistics* has emerged as a new field in supply chain optimization. Reverse logistics aims at the optimal integration of a forward flow of goods from producers to consumers with a backward flow of waste or used products from consumers to specialized warehouses or recycling sites. In this framework classical routing problems such as the traveling salesman problem (TSP) and the capacitated vehicle routing problem (CVRP) must be revisited, taking into account simultaneous goods delivery and waste collection. For a recent review on vehicle routing problems the reader is referred to the book edited by Toth and Vigo [25]. The *VRP with simultaneous delivery and collection* (VRPSDC) is a basic problem in reverse logistics and can be described as follows: a set of vehicles of limited capacity must visit a set of customers located on a transportation network; each customer requires a delivery of a given amount of goods, a collection of a given amount of waste, or both; all vehicles start from a common depot and must travel back to it; all goods are transported from the depot to the customers and all waste must be transported from the customers to the depot; the goal is to minimize the overall length of the vehicle routes.

A problem similar to the VRPSDC is the VRP with backhauls (VRPB), that is the problem in which each vehicle must complete all deliveries before starting to collect. Heuristic approaches to the VRPB have been proposed by Goetschalckx and Jacobs-Blecha [18] and by Toth and Vigo [24], while exact algorithms have been presented by Toth and Vigo [23] and Mingozzi et al. [21]. Wade and Salhi [27] developed an insertion-type algorithm for a particular version of the VRPB in which pick-up operations may start as soon as a given fraction of deliveries has been carried out.

The scientific literature is rather lacking in contributions on the VRPSDC. The problem was studied by Golden et al. [19] and Casco et al. [6], where the authors examined some greedy constructive algorithms. These papers consider the case in which every customer requires either a pick-up or a delivery (*simple demands*). Dethloff [11] took into account the setting in which each customer may require both kinds of operations (*composite demands*) and studied some constructive algorithms. Other constructive, local search, and tabu search algorithms have been developed by Bianchessi and Righini [5]. The VRPSDC with time windows has been recently investigated by Angelelli and Mansini [2], who proposed a branch-and-price algorithm, where the pricing problem is solved via dynamic programming. The presence of backhaul constraints

or time windows makes the problem easier for this kind of algorithm, since more constrained instances have smaller state spaces to explore with dynamic programming. For this reason algorithms developed for the VRPSDC with additional constraints cannot solve pure VRPSDC instances of the same size.

Apart from the algorithm of Angelelli and Mansini [2], whose effectiveness relies on time windows, no other exact optimization algorithm for the VRPSDC has been published so far to the best of our knowledge. The aim of this paper is to provide a first contribution to the exact solution of the problem: we present an optimization algorithm based on column generation, dynamic programming and branch-and-price, modulated on the general approach of Desrochers et al. [9] and Desaulniers et al. [8]. We consider the VRPSDC in which every customer must be served by one vehicle, demands can be composite and all vehicles are identical.

The paper is organized as follows: in Section 1 we define the 0-1 linear programming model of the VRPSDC; in Section 2 we present dynamic programming algorithms for the pricing subproblem; in Section 3 we describe the column generation algorithm; in Section 4 we illustrate the branch-and-bound algorithm; computational results are reported in Section 5, while Section 6 contains our conclusions.

1 Problem formulation

A VRPSDC instance is defined by the following data:

- a set \mathcal{N} of customers to be visited, numbered $1, \dots, N$;
- a depot, numbered 0;
- a directed graph $\mathcal{G}(\mathcal{N}^+, \mathcal{A})$, with vertex set $\mathcal{N}^+ = \mathcal{N} \cup \{0\}$ including the customers and the depot and with a complete arc set \mathcal{A} ;
- a weight function $c : \mathcal{A} \rightarrow \mathcal{Z}^+$ satisfying the triangular inequality (i.e. $c_{ij} \leq c_{ik} + c_{kj}$ for all $i, j, k \in \mathcal{N}^+$);
- an integer non-negative collection demand p_i and an integer non-negative delivery demand d_i associated with each customer $i \in \mathcal{N}$;
- a maximum number K of available vehicles;
- a capacity Q of each vehicle.

The mixed-integer programming model of the VRPSDC here below includes three variables for each arc: a binary variable x_{ij} takes value 1 if and only if arc $(i, j) \in \mathcal{A}$ belongs to the solution; continuous non-negative variables P_{ij} and D_{ij} indicate respectively the amount of collected load and delivery load carried along arc (i, j) .

$$VRPSDC) \quad \min \quad \sum_{(i,j) \in \mathcal{A}} c_{ij} x_{ij} \quad (1)$$

$$\text{s.t.} \quad \sum_{j \in \mathcal{N}^+} x_{ij} = 1 \quad \forall i \in \mathcal{N} \quad (2)$$

$$\sum_{j \in \mathcal{N}} x_{0j} \leq K \quad (3)$$

$$\sum_{j \in \mathcal{N}^+} x_{ij} = \sum_{j \in \mathcal{N}^+} x_{ji} \quad \forall i \in \mathcal{N}^+ \quad (4)$$

$$\sum_{j \in \mathcal{N}^+} P_{ij} - \sum_{j \in \mathcal{N}^+} P_{ji} = p_i \quad \forall i \in \mathcal{N} \quad (5)$$

$$\sum_{j \in \mathcal{N}^+} D_{ji} - \sum_{j \in \mathcal{N}^+} D_{ij} = d_i \quad \forall i \in \mathcal{N} \quad (6)$$

$$P_{ij} + D_{ij} \leq Q x_{ij} \quad \forall (i, j) \in \mathcal{A} \quad (7)$$

$$P_{ij}, D_{ij} \geq 0 \quad \forall (i, j) \in \mathcal{A} \quad (8)$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in \mathcal{A} \quad (9)$$

Constraints (2) force every customer to be visited once; constraint (3) implies that no more than K vehicles can be used; constraints (4), (5) and (6) are flow conservation constraints on the number of vehicles and on the amounts of pick-up and delivery load; constraints (7) ensure that the vehicle capacity is not exceeded.

The VRPSDC is \mathcal{NP} -hard, since it includes the capacitated vehicle routing problem as a special case, arising when $p_i = 0$ for all customers.

An alternative disaggregated formulation can be obtained from (1)-(9) through Dantzig-Wolfe decomposition. Such formulation is the starting point for the development of a branch-and-price algorithm in which the master problem is solved by column generation at every node of the search-tree. In the remainder we assume that the reader is familiar with branch-and-price and column generation techniques (see Desaulniers et al. [8] for an explanation of such techniques applied to routing problems).

Let \mathcal{R} be the set of all feasible routes, that is the set of routes starting from the depot, visiting some customers and going back to the depot, complying with the flow conservation and the capacity constraints. The VRPSDC can be reformulated using a

binary variable z_r for each route $r \in \mathcal{R}$, taking value 1 if and only if the route belongs to the solution.

$$MP) \quad \min \quad \sum_{r \in \mathcal{R}} c_r z_r \quad (10)$$

$$\text{s.t.} \quad \sum_{r \in \mathcal{R}} y_{ir} z_r = 1 \quad \forall i \in \mathcal{N} \quad (11)$$

$$\sum_{r \in \mathcal{R}} z_r \leq K \quad (12)$$

$$z_r \in \{0, 1\} \quad \forall r \in \mathcal{R} \quad (13)$$

The cost c_r of each route in \mathcal{R} is given by the sum of the costs of the arcs belonging to the route. Coefficients y_{ir} indicate how many times customer i is visited along route r . As long as the columns of the master problem are cycle-free, coefficients y_{ir} are binary; if routes containing cycles are allowed, as explained in the remainder, the coefficients can take non-negative integer values. This master problem (*MP*) explicitly retains constraints (2) and (3) while the other constraints are taken into account when feasible routes are generated. A complete VRPSDC solution is represented by a subset of disjoint routes taken from \mathcal{R} .

Partitioning constraints (11) can be replaced by the following covering constraints:

$$\sum_{r \in \mathcal{R}} y_{ir} z_r \geq 1 \quad \forall i \in \mathcal{N} \quad (14)$$

This replacement is a common practice when designing column generation algorithms because no optimal solution visits any customer more than once (hence the two models have the same optimal solution) and because set covering formulations are preferable to set partitioning formulations: indeed, when integrality requirements (13) are relaxed, a set covering model has a smaller dual space. Moreover, when only a restricted subset of columns is available, feasible solutions are easier to find.

The linear relaxation of the master problem, indicated hereafter by *LMP*, is solved through column generation by considering a restricted linear master problem *RLMP* with a limited subset of columns $\mathcal{R}' \subseteq \mathcal{R}$ and by repeatedly solving a pricing subproblem to generate columns with negative reduced cost, if any exists. The pricing problem consists in finding an elementary route with additional constraints. In its ILP formulation binary variables x_{ij} indicate the use of arc $(i, j) \in \mathcal{A}$ and each variable y_i indicates whether customer i is visited along the route; its value is reported into the

coefficient y_{ir} in the corresponding column r of the *RMP*.

$$\min \tilde{c} = \sum_{(i,j) \in \mathcal{A}} c_{ij} x_{ij} - \sum_{i \in \mathcal{N}} \lambda_i y_i + \mu \quad (15)$$

$$\text{s.t.} \quad \sum_{j \in \mathcal{N}^+} x_{ij} = y_i \quad \forall i \in \mathcal{N} \quad (16)$$

$$\sum_{j \in \mathcal{N}^+} x_{ji} = y_i \quad \forall i \in \mathcal{N} \quad (17)$$

$$\sum_{j \in \mathcal{N}} x_{0j} = 1 \quad (18)$$

$$\sum_{j \in \mathcal{N}} x_{j0} = 1 \quad (19)$$

$$\sum_{j \in \mathcal{N}^+} P_{ij} - \sum_{j \in \mathcal{N}^+} P_{ji} = p_i y_i \quad \forall i \in \mathcal{N} \quad (20)$$

$$\sum_{j \in \mathcal{N}^+} D_{ji} - \sum_{j \in \mathcal{N}^+} D_{ij} = d_i y_i \quad \forall i \in \mathcal{N} \quad (21)$$

$$P_{ij} + D_{ij} \leq Q x_{ij} \quad \forall (i, j) \in \mathcal{A} \quad (22)$$

$$P_{ij}, D_{ij} \geq 0 \quad \forall (i, j) \in \mathcal{A} \quad (23)$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in \mathcal{A} \quad (24)$$

$$y_i \in \{0, 1\} \quad \forall i \in \mathcal{N} \quad (25)$$

We have indicated with \tilde{c} the reduced cost of a column, with λ the vector of non-negative dual variables associated with covering constraints (14), and with μ the non-negative dual variable associated with constraint (12) when it is put in “ \geq ” form.

This problem can be reformulated as an elementary shortest path problem with two resource constraints in which the path starts from the depot and goes back to it and the resources consist of the amount of load the vehicle can still collect and deliver. The cost of the path, given by (15), depends on the costs of the arcs traversed as well as on the prizes λ collected at the vertices visited. This corresponds to formulate the problem on a graph whose arc costs can be negative.

The lower bound obtained from this disaggregated formulation is stronger than that obtained from the linear relaxation of the original formulation, since it results from the convexification of integral points (corresponding to routes) contained in the original polyhedron and the pricing subproblem that generates such points does not possess the integrality property. The linear relaxation of the initial formulation (1)-(9) has a feasible region given by $Q_1 \cap Q_2$, where $Q_1 = \{x \in \mathbb{R}^{|\mathcal{A}|} : (2), (3), 0 \leq x \leq 1\}$ and $Q_2 = \{x \in \mathbb{R}^{|\mathcal{A}|}, P \in \mathbb{R}^{|\mathcal{A}|}, Q \in \mathbb{R}^{|\mathcal{A}|} : (4) - (8), 0 \leq x \leq 1\}$. The effect of the

reformulation proposed above is that of replacing Q_2 by the convex hull of its integer points $\text{conv}(Q_2)$. Since $Q_1 \cap \text{conv}(Q_2) \subseteq Q_1 \cap Q_2$, a column generation approach gives tighter lower bounds than linear programming at the expense of the necessity of optimizing an integer pricing subproblem at each iteration. This well-known property is at the base of all applications of column generation to ILP problems in general and VRP problems in particular. More details on this can be found in classical textbooks on integer programming (see for instance Wolsey [28]) and VRP references (see for instance Desrosiers et al. [10]). The computational results reported in the last section of this paper show that the disaggregated formulation actually is much stronger than the linear relaxation of the original formulation.

2 The pricing subproblem

The shortest path problem with resource constraints is \mathcal{NP} -hard, even when the costs are non-negative (see Garey and Johnson [16], problem ND30). The requisite that the path must be elementary is significant when the graph contains negative cost cycles; Dror [12] proved that in this case the problem is \mathcal{NP} -hard in the strong sense.

Different approaches may be used to solve the pricing subproblem; in this paper we focus on dynamic programming. We consider a task-on-arc formulation where the consumption of resources is associated with the arcs. We translate all constraints into resource constraints and we search for shortest paths in which the available resource amounts are non-negative.

For notational convenience we suppose that the depot is split into two vertices s and t .

In our dynamic programming algorithm the capacity constraint is taken into account by resources π and δ . Resource π at vertex j indicates the amount of load that the vehicle can collect after visiting j ; it is initialized at the value of the vehicle capacity (that is, $\pi_0 = Q$) and it decreases after every pick-up operation. When the vehicle visits vertex j , it consumes p_j units of this resource. Therefore, each time arc (i, j) is traversed, the new value of resource π is:

$$\pi' = \pi - p_j$$

Resource δ at node j indicates the amount of load that the vehicle can deliver after visiting j and it is initially set to Q (that is, $\delta_0 = Q$). The maximum amount of load that can be delivered after visiting vertex j is limited by the difference between Q and the maximum amount of load carried at some point by the vehicle since its departure

from s up to vertex j . Therefore, this resource decreases each time a delivery operation is performed but it may decrease also after pick-up operations. Each time arc (i, j) is traversed, the new value of resource δ is:

$$\delta' = \min\{\delta - d_j, \pi - p_j\}$$

To impose that paths be elementary we also consider N dummy resources, each one corresponding to a vertex; when a vehicle leaves the depot it has one unit of each of these resources and each time the vehicle visits vertex j it consumes a unit of the j -th resource. This method, first suggested by Beasley and Christofides [4], ensures that no vertex can be visited more than once. We indicate with S the vector of these dummy resources, initialized as $S^0 = \{1, \dots, 1\}$. Whenever arc (i, j) is used to extend a feasible path from vertex i to vertex j , the resource vector is updated as follows:

$$S' := S - e^j$$

where e^j is a binary vector whose j -th component is equal to 1 and the others are equal to 0. Note that S does not keep any explicit information about the order in which the vertices have been visited.

Each state in our dynamic programming algorithm is represented by a quadruple (S, π, δ, j) , and $cost(S, \pi, \delta, j)$ indicates the minimum cost of the path going from the depot s to vertex j such that it may be extended using non-negative amounts of available resources S, π and δ . The cost of the state is given by the sum of the costs of the arcs in the path from s to j minus the prizes collected at the vertices visited. It is initialized at 0 (that is $cost(S^0, \pi_0, \delta_0, s) = 0$) and each time arc (i, j) is traversed it is increased by $c_{ij} - \lambda_j$ if $j \neq t$ and by $c_{ij} + \mu$ if $j = t$. The optimal solution corresponds to the minimum cost of a feasible state (S, π, δ, t) , that is the minimum cost of a feasible path reaching vertex t .

Our algorithm is a “push” algorithm, that is for each current feasible state (S, π, δ, i) only its feasible successors (S', π', δ', j) are generated for all $(i, j) \in \mathcal{A}$ such that $S' \geq 0$, $\pi' \geq 0$ and $\delta' \geq 0$.

Domination rules. The effectiveness of a dynamic programming algorithm depends on domination rules that make it possible to fathom some states. In our basic dynamic programming algorithm a path $(S^1, \pi_1, \delta_1, j)$ dominates a path $(S^2, \pi_2, \delta_2, j)$ if

- (a) $cost(S^1, \pi_1, \delta_1, j) \leq cost(S^2, \pi_2, \delta_2, j)$
- (b) $S^1 \geq S^2$
- (c) $\pi_1 \geq \pi_2$
- (d) $\delta_1 \geq \delta_2$

and at least one of the inequalities is strict. Note that condition (c) is redundant, since it is implied by condition (b) (the same is not true in general for condition (d) because the amount that can be delivered depends on the order in which the vertices have been visited).

An exact dynamic programming algorithm using the above definitions should consider an exponential number of states and no pseudo-polynomial algorithm is known for this problem. Based on our computational experience it is clear that this exact approach is impractical even for problem instances of small size. Feillet et al. [15] suggested improving the performance of the basic dynamic programming approach to the solution of the elementary shortest path problem with resource constraints by setting to zero the j -th component of resource vector S for all vertices j which cannot be visited because of the resource constraints on π and δ . We implemented this idea but the improvement we obtained was negligible. Our feeling is that the reduction in the number of states achievable by this technique is significant only when the resource constraints are very tight, for instance in problem instances with narrow time windows.

In order to reduce the computing time of the pricing algorithm, we designed two accelerated dynamic programming algorithms: an upper bounding algorithm based on heuristic graph reduction and a lower bounding algorithm based on state space relaxation.

2.1 Upper bounding

A heuristic algorithm based on dynamic programming is obtained by discarding all arcs $(i, j) \in \mathcal{A}$ such that $c_{ij} > \alpha \max_{k \in \mathcal{N}} \{\lambda_k\}$, where parameter α is set between 0 and 1. When the dynamic programming algorithm described above is run on the reduced graph, it is very fast and, though it may miss the optimal solution, it can find negative reduced cost columns.

In our experiments we set $\alpha = 0.1$ and we observed that the first time the algorithm was executed at each node of the search-tree it found a lot of negative reduced cost columns, including the optimal one, in a small fraction of the computing time required by the same algorithm on the complete graph. However, if the value of α was left unchanged, the algorithm failed to find negative reduced cost columns in subsequent runs. Therefore we raise α by 0.2 in every subsequent trial; if no column with negative reduced cost is found with $\alpha = 0.9$, the heuristic algorithm stops.

2.2 Lower bounding

A common way to reduce the computing time of a dynamic programming algorithm is state space relaxation, a technique introduced by Christofides et al. [7]: the state space \mathcal{S} 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). A 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 does not guarantee finding an optimal solution but rather a lower bound.

Our state space relaxation consists in mapping each state (S, π, δ, j) onto a new state (σ, π, δ, j) , where $\sigma = \sum_{k=1}^N S_k$, that is, the number of vertices that can be visited after j . Resource σ is initialized at N and it is simply decreased by one unit each time an arc is traversed. Since the state does no longer keep information about the set of already visited vertices, cycles are no longer forbidden (however, infinite loops are prevented, since resources σ , π and δ are monotonically decreasing). Therefore the optimal path is guaranteed to be feasible with respect to capacity constraints but it is not guaranteed to be elementary.

In the relaxed state space a state $(\sigma_1, \pi_1, \delta_1, j)$ dominates a state $(\sigma_2, \pi_2, \delta_2, j)$ if

- (a) $cost(\sigma_1, \pi_1, \delta_1, i) \leq cost(\sigma_2, \pi_2, \delta_2, i)$
- (b) $\sigma_1 \geq \sigma_2$
- (c) $\pi_1 \geq \pi_2$
- (d) $\delta_1 \geq \delta_2$

and at least one of the four inequalities is strict. Note that in this case condition (c) is no longer redundant.

A dynamic programming algorithm based on these new recurrence equations has pseudo-polynomial time complexity, since the number of possible states is limited by $(Q + 1)^2 N^2$.

2.3 Implementation

We found that the effectiveness of the algorithms outlined above strongly depends on several design issues. First of all we adopted the technique of Desrochers et al. [9] to avoid cycles of length 2. More powerful methods have been proposed by Irnich and Villeneuve [20] to forbid cycles of length $k \geq 3$, but we did not incorporate them in our algorithm because they are too time-consuming.

In addition we introduced two new ideas, namely bi-directional dynamic programming and bounded number of steps.

Bi-directional search. A first refinement consists in implementing bi-directional search. Although this is a well known idea for computing shortest paths (see, for instance, Ahuja et al. [1], chapter 4.5), we could not find any attempt to exploit it in the literature on dynamic programming algorithms for routing problems solved via column generation. In bi-directional search states are extended both forward from vertex s to their successors and backward from vertex t to their predecessors. States, recurrence equations, and domination rules are symmetrical to those presented above. The idea is applicable to exact dynamic programming as well as to the state space relaxation algorithm. In the former case each vertex $i \in \mathcal{N} \cup \{s, t\}$ has associated forward states indicated by $(S^{fw}, \pi^{fw}, \delta^{fw}, i)$ and backward states indicated by $(S^{bw}, \pi^{bw}, \delta^{bw}, i)$. Resources π^{bw} and δ^{bw} represent, respectively, the amounts of load the vehicle can collect and deliver *before* vertex i . A path from s to t is detected each time a forward state $(S^{fw}, \pi^{fw}, \delta^{fw}, i)$ and a backward state $(S^{bw}, \pi^{bw}, \delta^{bw}, j)$ can be feasibly joined through arc (i, j) . The feasibility test in the exact dynamic programming algorithm is:

$$\begin{aligned} S_k^{fw} + S_k^{bw} &\geq 1 \quad \forall k = 1, \dots, N \\ \pi^{fw} + \pi^{bw} &\geq Q \\ \delta^{fw} + \delta^{bw} &\geq Q \end{aligned}$$

The first condition imposes that the same vertex cannot be visited by both paths. The second condition states that the sum of the amounts of resource π used in the two paths (i.e. $(Q - \pi^{fw}) + (Q - \pi^{bw})$) cannot exceed the overall capacity Q ; the third condition is analogous. In the state space relaxation algorithm the first condition is surrogated by $\sigma^{fw} + \sigma^{bw} \geq N$. Clearly any path will be at most N vertex long. In our implementation forward and backward states are extended alternately so that the numbers of steps in forward paths and in backward paths differ at most by one. This is done in order to keep the two state sets as balanced as possible. After each iteration we check for compatible pairs of paths for all arcs $(i, j) \in \mathcal{A}$. In this way the algorithm considers each $s - t$ path only once.

Bounded number of steps. Another improvement consists in computing an upper bound $\bar{\sigma}$ on the number of customers that a vehicle can visit. A forward dynamic programming algorithm could be limited to explore only paths with up to $\bar{\sigma}$ steps, while our bi-directional search algorithm is limited to paths with up to $\lceil \bar{\sigma}/2 \rceil$ steps.

An upper bound $\hat{\sigma}$ to the number of customers in a feasible path can be computed by solving the following two-constraints knapsack problem:

$$\begin{aligned}
\max \quad & \sum_{i \in \mathcal{N}} u_i \\
\text{s.t.} \quad & \sum_{i \in \mathcal{N}} p_i u_i \leq Q \\
& \sum_{i \in \mathcal{N}} d_i u_i \leq Q \\
& u_i \in \{0, 1\} \qquad \qquad \qquad \forall i \in \mathcal{N}
\end{aligned}$$

The two constraints impose that neither pick-up demands nor delivery demands on the vertices to be selected will exceed the capacity. In our implementation we do not solve this knapsack problem to optimality, but we take the upper bound $\bar{\sigma}$ computed as follows: we solve separately the two knapsack problems obtained by considering only one constraint at a time and we take the minimum of the two values. Each knapsack problem can be solved in linear time with the method of Balas and Zemel [3], since the profits in the objective function are all identical. This technique does not prevent the algorithm from discovering the optimal elementary path, but it can significantly reduce the number of states considered. Moreover, when state space relaxation is used, it also reduces the number of non-elementary paths generated; it may also fathom the optimal path with cycles, thus improving the lower bound given by the relaxed pricing algorithm.

The combination of this idea with bi-directional search yields an impressive computational advantage.

2.4 Overall pricing algorithm

In column generation algorithms it is common to execute some fast heuristics to quickly generate columns with negative reduced cost whenever possible. In our algorithm at each column generation iteration the search for new columns is made by successive steps of increasing computational complexity. The search is stopped as soon as M columns with negative reduced cost have been found (in our tests M was set to 200).

Step 1: Search in the pool. A pool of previously computed columns is scanned and the reduced cost is evaluated for all columns which are feasible for the current node. Columns with cycles are also included in the pool since otherwise they could be

re-generated by the pricing algorithm.

Step 2: Deterministic greedy algorithm. A nearest neighbor algorithm produces a solution by adding arcs to a path starting from vertex s and going forward or starting from vertex t and going backward. At each iteration the most valuable arc is added, among those which do not close cycles. The value of an arc (i, j) is computed as $\lambda_j - c_{ij}$. The algorithm is executed once forward and once backward.

Step 3: Randomized greedy algorithm. This step is similar to the previous one but at every iteration the next arc is randomly chosen among the four most valuable arcs extending the path. The algorithm is multistarted forward and backward for a number of times depending on the size of the problem instance.

Step 4: Approximate dynamic programming. The exact (bi-directional) dynamic programming algorithm is executed on the graph reduced as explained in subsection 2.1, with values of α increasing from 0.1 to 0.9 until a new column is found.

Step 5: Dynamic programming. This last step consists in running the dynamic programming algorithm, with or without state space relaxation, as described in subsections 2.2 and 2.3. The comparison between these two strategies is discussed in Section 5. This step is executed only if no column with negative reduced cost has been generated in steps 1-4.

3 Column generation algorithm

In this section we present the overall column generation algorithm and in particular we discuss initialization, columns management, termination and stabilization. This algorithm is executed at each node of a search-tree, as described in the next section.

Initialization. The restricted linear master problem (*RLMP*) must be properly initialized to guarantee feasibility and to speed up the convergence of the column generation algorithm. To ensure the existence of a feasible solution we insert a dummy column with a very high cost, representing a fictitious (and infeasible) route visiting all the customers.

At the root node we include into the *RLMP* the feasible routes obtained by the tabu search algorithm of Bianchessi and Righini [5], that was devised to solve the VRPSDC

without constraints on the number of vehicles. The number of such columns depends on the problem instance and it usually ranges from 20 to 80. The VRPSDC solutions found by this algorithm are often optimal and in some cases super-optimal, since the algorithm is not forced to use the right number of vehicles. However we observed that even when the solutions provided by the tabu search algorithm were infeasible, the overall algorithm took benefit from the initial set of columns generated in this way.

At each non-root node of the search-tree, our algorithm initializes the *RLMP* with the same set of columns used by the last considered node, except the columns that have become infeasible because of branching. Note that the last node considered does not necessarily coincide with the father of the current node, but depends on the search strategy.

Columns management. At each iteration of the column generation algorithm all columns with negative reduced cost which have been found are inserted into the *RLMP*. When the number of columns in the *RLMP* exceeds a predefined fixed value (set to 5000 in our tests), we delete the columns with a reduced cost greater than the threshold value $1.5 \bar{\sigma} \min_{(i,j) \in \mathcal{A}} \{c_{ij}\}$, where $\bar{\sigma}$ is the maximum number of steps in a feasible route as defined in Section 2.3. Each column deleted from the master problem is stored in a pool together with the list of the arcs used in the corresponding route. These additional data are necessary to test whether the column is feasible at a node of the search-tree, since the column itself does not contain explicit information on the sequence in which the customers are visited. A column is eliminated from the pool after ten consecutive evaluations (step 1 of the pricing algorithm, see Section 2.4) yielding a positive reduced cost.

Lower bounding and termination. It is well-known that one of the drawbacks of column generation is the so-called “tailing-off” effect: a lot of iterations that do not significantly modify the optimal value of the *RLMP* are often necessary to prove optimality. Therefore, many authors (see, for instance, Farley [14] and Vanderbeck and Wolsey [26]) have proposed alternative lower bounding techniques, that allow an earlier termination of column generation yet providing a valid lower bound.

Our algorithm exploits the bound obtained from the Lagrangean relaxation of covering constraints (14). When the optimal value \tilde{c}^* of the (possibly relaxed) pricing subproblem is negative, we can compute the lower bound

$$\omega_{lb} = K\tilde{c}^* + \omega_{RLMP}^*$$

where ω_{RLMP}^* is the optimal value of the current *RLMP* (see Farley [14]). When ω_{lb} is greater than or equal to the best incumbent feasible solution, the current node is fathomed. If column generation is terminated because the time limit has expired, ω_{lb} is kept as the final lower bound of the node.

Stabilization. We compared two different stabilization techniques, namely the interior point technique described by Rousseau et al. [22] and the one based on dual boxes described by du Merle et al. [13]. From our experiments we found that, for our problem, the former technique works much better than the latter. The interior point technique consists in finding a point in the interior of the optimal dual polyhedron whenever the primal problem is degenerate; its dual therefore has many distinct optimal solutions. The stabilization algorithm generates a prespecified number of different optimal dual solutions (we set this number to 100) and then it takes a strict convex combination of them. The effect is to distribute the dual prizes λ among all the customers instead of concentrating them on a few ones. We observed that the effect of this stabilization technique was favourable both in terms of size of the search-tree and in terms of the overall computing time.

4 The branch-and-price algorithm

The column generation algorithm described above is executed at every node of a search-tree in a branch-and-bound framework. In this section we describe the search strategy, the upper bounding technique and the branching strategies we employed.

Search strategy. We explore the search-tree according to a best-first policy, where subproblems are ranked on the basis of the associated lower bound. Experiments with depth-first search yielded inferior results in terms of number of problems solved to optimality within the same amount of computing time.

Upper bounding. We did not insert in our algorithm any sophisticated upper bounding technique because computing a feasible solution of the VRPSDC is an \mathcal{NP} -complete problem in itself. To quickly produce feasible solutions during the search we devised the following greedy procedure, exploiting the structure of the current (possibly fractional) *RLMP* solution. We iteratively fix to 1 the binary variable z_r of a column and we solve the resulting *RLMP* again. At each iteration the variable fixed to 1 corresponds to the cycle-free column with the highest fractional value. We iterate

until either a complete integer solution is obtained or the algorithm fails. This heuristic is quite fast, so we execute it at each column generation iteration. This is especially useful for large instances, where the initial upper bound provided by tabu search is not so close to the optimum.

Branching. We implemented different branching strategies, namely, branching on cycles, branching on resources and branching on arcs. The former two are used mainly to eliminate cycles from the optimal solution of the *RLMP*. All three strategies are compliant with the structure of the pricing subproblem and with the implementation of the dynamic programming algorithms. Notably, the strategies can also be combined so that the branch-and-price algorithm can use any of them.

Branching on cycles. First we determine the minimum length, say k , of the cycles among all the basic columns in the current optimal solution of the *RLMP*; then we select the route \hat{r} with a cycle of length k whose variable $z_{\hat{r}}$ has the largest value. Then k children nodes are generated: at child node $h = 0 \dots k - 1$, we impose the first h arcs of the cycle and we forbid the $h + 1$ -th arc. Fixing arc variables is easy to take into account in the pricing algorithm at the children nodes: when arc (i, j) is imposed, vertices i and j are shrunk into a single vertex; when arc (i, j) is forbidden, it is deleted from the graph. We experimentally observed that the number of children generated is often equal to 3.

Branching on resources. It is common that in the optimal *RLMP* solution a customer is visited several times, that is, there are several arcs entering and leaving a vertex \hat{i} . This may happen when a route has a cycle as well as when two or more routes have some vertex in common. The branching strategy consists in adding a constraint on the quantity of resource consumed by the vehicle(s) up to the visit of vertex \hat{i} . This idea was proposed by Gélinas et al. [17] for routing problems with time windows and it can be adapted to any problem in which some resource varies monotonically along the route.

Consider a vertex \hat{i} visited twice or more in the paths used by the optimal solution of the *RLMP* and choose two visits to \hat{i} along such paths. The dynamic programming recursion associates a forward and a backward state to each visit. Let $(\sigma^1, \pi^1, \delta^1, \hat{i})$ and $(\sigma^2, \pi^2, \delta^2, \hat{i})$ be the two forward states associated to two visits to \hat{i} . Without loss of generality assume $\pi^1 < \pi^2$ (the same idea can be applied to resource δ as well as to backward states). Then an integer branching value $\bar{\pi}_{\hat{i}}$ is chosen such that $\pi^1 < \bar{\pi}_{\hat{i}} \leq \pi^2$;

then two children nodes are generated imposing $\pi \geq \bar{\pi}_i$ in one child node and $\pi \leq \bar{\pi}_i - 1$ in the other.

To choose the branching vertex, we first select the set H of vertices visited the least number of times but at least twice in the basic columns of the RLMP; to this purpose we disregard the possibly fractional values associated with the columns and we only consider whether a column is basic or not. Then we choose $\hat{i} \in H$ and $\bar{\pi}_i$ so that the sum of the z values of the columns which become infeasible is the most balanced between the two children nodes.

Branching on arcs. When there are no cycles in the basic columns but there are still fractional variables, we perform branching on arcs. This binary branching scheme consists in selecting a vertex visited by different fractional routes through different arcs. One such arc is then imposed in one child node and forbidden in the other.

Mixed branching. Branching-on-cycles is only applicable when state space relaxation is used to solve the pricing subproblem. If exact dynamic programming is used, then the columns have no cycles and branching on arcs must be used. Branching-on-resources can be used in both cases, but in our experiments it yielded inferior results in terms of computing time and number of search nodes explored, compared with the other branching policies. The only case in which we found branching-on-resources useful is when a vertex is visited in two different cycles by two different routes, that is the “butterfly” case illustrated in Figure 1.

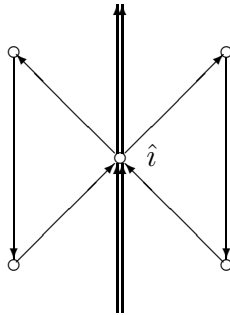


Figure 1: A “butterfly”: vertex \hat{i} belongs to two different cycles

In this case by branching-on-resources it is often possible to forbid both cycles in both children nodes. Therefore, when state space relaxation is adopted, we use a mixed branching strategy: if any “butterfly” is found, branching is done on resources to eliminate it, if possible; otherwise, branching is done on cycles.

5 Computational results

We implemented two branch-and-price algorithms: one (indicated by *EXACT*) based on exact dynamic programming with branching-on-arcs strategy and the other (indicated by *RELAX*) based on state space relaxation with mixed branching strategy.

We tested our algorithms on five classes of benchmark instances: Class 1, Class 2S, Class 2C, Class 3S and Class 3C, where S stands for “simple demands” and C for “composite demands”.

Class 1 consists of twelve instances derived from Solomon’s instances r101, c101 and rc101, originally proposed for the VRPTW. We considered the first 20 or 40 customers, we neglected the time-windows and we followed the method proposed by Angelelli and Mansini [2] to generate composite demands: the demands given in Solomon’s instances were assumed to represent delivery demands d_i , while pick-up demands p_i were generated as $p_i = \lfloor (1 - \gamma) d_i \rfloor$ if i is even, and $p_i = \lfloor (1 + \gamma) d_i \rfloor$ if i is odd. For each instance in Solomon’s benchmark we generated two VRPSDC instances with $\gamma = 0.2$ and $\gamma = 0.8$. Vehicles capacity was set to 100. The Euclidean distance between customers was rounded up to a multiple of 0.1 to guarantee that the triangular inequality held.

Class 2S was made of 18 instances generated as described by Toth and Vigo [23]. The number of customers is equal to 20 or 40. Customer coordinates are uniformly distributed in the intervals $[0, 24000]$ for the x values and $[0, 32000]$ for the y values; the depot is located at $(12000, 16000)$. The cost of each arc was defined as the Euclidean distance rounded up to an integer value. Demands were generated at random from a normal distribution with mean value equal to 50 and standard deviation equal to 20. The fraction of delivery customers is equal to $\frac{1}{2}$, $\frac{2}{3}$ or $\frac{4}{5}$ (this is indicated by the percentage values 50, 66 and 80 in the name of the instances). The vehicles capacity varies in $\{100, 150, 200\}$.

Class 2C was obtained from class 2S using the following method: all demands of the corresponding instance in class 2S were considered as delivery demands and the pick-up demand of each customer i was computed as $p_i = (0.5 + r)d_i$, where r was taken from a uniform distribution in the interval $[0,1]$.

Class 3S consists of 36 instances, obtained from four real world CVRP instances of the VRPLIB with N equal to 20 or 40. For each CVRP instance nine VRPSDC instances were generated, each one corresponding to a fraction of delivery customer equal to $\frac{1}{2}$, $\frac{2}{3}$ and $\frac{4}{5}$ and to a vehicle capacity equal to 150, 200 and 300.

Class 3C was obtained from class 3S with the same technique as class 2C.

For classes 2S, 2C, 3S and 3C we computed K as the minimum number of vehicles for which a feasible solution exists: for each instance with simple demands we solved two bin packing problems, one related to pick-up demands and one to delivery demands, and we took the maximum of the two optimal solutions. For each instance with composite demands we solved a bi-dimensional vector packing problem. All these problems were solved by the mixed integer solver of CPLEX 6.5.

All tests were done on a PC Pentium-IV 1.6 GHz with 512 MB of RAM under Linux Red Hat 7.3 operating system. Algorithms were implemented in ANSI-C and compiled with gcc version 2.96. To solve the *RLMPs* we used the primal simplex algorithm of ILOG CPLEX 6.5 with standard settings. The time limit for each run of each branch-and-price algorithm was set to one hour. The computing time spent for the initialization with the tabu search algorithm was limited to one minute.

In the tables hereafter we report for each instance the number of vertices N , the number of vehicles K , the upper bound $\bar{\sigma}$ on the number of customers per vehicle computed at the root node as explained in Section 2.3, and the percentage gap *LB Gap* between the lower bound given by the linear relaxation of the original formulation (1)-(9) and the lower bound given by the set covering formulation (10), (12), (13) and (14) at the root node.

We also report results concerning the root node and the complete search-tree. For the root node we report the percentage improvement *LB impr.* from the lower bound at the root node, say LB_{root} to the final lower bound, say LB_{final} (that is $LB\ impr. = 100(LB_{final} - LB_{root})/LB_{root}$), the time spent in seconds (T_{root}) and the number of columns generated (C_{root}). For the search-tree we report the final upper bound (UB), the percentage gap between the final upper bound and the final lower bound (*Gap*), the overall time spent (T_{tree}), the number of nodes explored (*Nodes*), the percentage of time spent for pricing (T_{price}) and the total number of columns generated (C_{tree}).

Exact dynamic programming vs. state space relaxation. In the first set of our experiments, reported in Table 1, we compared the exact dynamic programming algorithm with the state space relaxation algorithm. We used Class 1 instances as a testbed. Table 1 does not report the results of algorithm EXACT for instances with 40 customers because the algorithm could not complete the computation at the root node within the time limit. On instances with 20 customers algorithm EXACT spent a large amount of time at the root node but yielded stronger lower bounds (see column *LB impr.*) than algorithm RELAX. However, in spite of the better lower bounds producing smaller search-trees, this approach is not competitive with algorithm

RELAX (see column T_{tree}).

These results contrast with those obtained by Feillet et al. [15] on the VRPTW, because the effectiveness of the exact dynamic programming algorithm depends heavily on the tightness of the constraints in the pricing problem. To have a quantitative measure of the effect of the constraints on the dynamic programming algorithm, we observed that the average number of non-dominated labels generated by our pricing algorithm for the VRPSDC was 10 to 20 times larger than that reported by Feillet et al. [15] for VRPTW. Therefore, we are convinced that the trade-off between exact dynamic programming and state space relaxation deserves further investigation in order to identify the best pricing strategy according to the structure of the problem instances at hand.

Relying upon the outcome of this first set of experiments, we tested only the algorithm RELAX on the other four classes of instances.

Experimental results with RELAX. Tables 2-4 correspond to the other four classes of problem instances. The algorithm solved all instances in classes 2C and 2S to optimality except instance 2S_40_66_2. It solved 15 out of 18 instances in Class 3S and 16 out of 18 instances in Class 3C. In these instances the computing time is replaced by an asterisk. Even in these cases, however, it yielded a very small gap, with the exception of instance 3S_40_50_3 for which the dynamic programming algorithm could not terminate at the root node within the time limit. For this instance the value in column $LB\ gap$ was determined by letting the algorithm run without time limits.

Remarkably, the percentage of time spent on pricing was very high and got close to 100% for problems in classes 3S and 3C.

There is clear correlation between the value of $\bar{\sigma}$ and the difficulty of the problem instance, measured by the overall time and the number of columns generated. When the capacity constraints are less tight, that is, $\bar{\sigma}$ is higher, the algorithm needs to generate a larger number of columns and this requires a significant amount of computing time. Therefore, $\bar{\sigma}$ can be assumed as a rough measure of the difficulty of a VRPSDC instance from the viewpoint of a branch-and-price approach.

6 Conclusions

In this paper we have introduced for the first time branch-and-price algorithms for the exact solution of the vehicle routing problem with simultaneous delivery and collection without any additional constraint. We have introduced new ideas to improve the

performance of dynamic programming algorithms used to solve the pricing problem, namely, bi-directional search and bounded number of steps, and we have studied different branching strategies. We have implemented both exact dynamic programming and state space relaxation and we have compared them through computational experiments. From such experiments we can conclude that branch-and-price is a viable approach to the solution of the VRPSDC both as an optimization method for instances of small to medium size. We foresee a number of future developments along this line of research. First, we mention the need to measure in a quantitative and reliable way the effect of constraints on the pricing problem; indeed, different constraints may have quite a different impact on the structure of the pricing problem and on the effectiveness of dynamic programming. Therefore, it is necessary to further investigate which constraints make a vehicle routing problem amenable for dynamic-programming-based branch-and-price algorithms and to what extent. Another issue is the development of different and hopefully complementary approaches to the VRPSDC, such as branch-and-cut algorithms, which can be a useful alternative to dynamic programming in the solution of the pricing subproblem if not as an alternative to branch-and-price for the overall routing problem.

Acknowledgements

This research was partially supported by Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) and by Consiglio Nazionale delle Ricerche (CNR). Giovanni Righini and Matteo Salani also acknowledge the kind support of ACSU - Associazione Creasca Studi Universitari to the Operations Research Laboratory of the Department of Information Technologies of the University of Milano, where this research was done.

References

- [1] Ahuja R.K., Magnanti T.L., Orlin J.B., *Network flows* Prentice Hall, 1993.
- [2] Angelelli E., Mansini R., *The vehicle routing problem with time windows and simultaneous pick-up and delivery* Lecture Notes in Economics and Mathematical Systems, A. Klose et al. (eds), 249-267, Springer, 2002
- [3] Balas E., Zemel E., *An algorithm for large zero-one knapsack problems* Operations Research 20, 1130-1154 (1980).
- [4] Beasley J.E., Christofides N., *An algorithm for the resource constrained shortest path problem* Networks 19, 379-394 (1989).
- [5] Bianchessi N., Righini G., *Heuristic algorithms for the vehicle routing problem with simultaneous pick-up and delivery* Note del Polo - Ricerca 50, Dipartimento di Tecnologie dell'Informazione, Università degli Studi di Milano, 2003.
- [6] Casco D.O., Golden B.L., Wasil E.A., *Vehicle routing with backhauls: models, algorithms and case studies*, in Vehicle Routing: Methods and Studies, B.L. Golden and A.A. Assad (eds), 127-147, Elsevier Science, 1988.
- [7] Christofides N., Mingozzi A., Toth P., *State-space relaxation procedures for the computation of bounds to routing problems* Networks 11, 145-164 (1981).
- [8] Desaulniers G., Desrosiers J., Ioachim I., Solomon M.M., Soumis F., Villeneuve D., *A unified framework for deterministic time constrained vehicle routing and crew scheduling problems* in Fleet management and logistics, T.G. Crainic, G. Laporte (eds), 57-93, Kluwer, Boston, 1998.
- [9] Desrochers M., Desrosiers J., Solomon M., *A new optimization algorithm for the vehicle routing problem with time windows* Operations Research 40, 342-354 (1992).
- [10] Desrosiers J., Dumas Y., Solomon M., Soumis F., *Time constrained routing and scheduling* in Network Routing, Ball M.O. et al. eds., Handbooks in Operations Research and Management Science, Elsevier Science 1995
- [11] Dethloff J., *Vehicle routing and reverse logistics: the vehicle routing problem with simultaneous delivery and pick-up*, OR Spektrum 23, 79-96 (2001).

- [12] Dror M., *Note on the complexity of the shortest path models for column generation in VRPTW*, Operations Research 42(5), 977-978 (1992).
- [13] du Merle O., Villeneuve D., Desrosiers J., Hansen P., *Stabilized column generation* Discrete Mathematics 194, 229-237 (1999).
- [14] Farley A.A., *A note on bounding a class of linear programming problems, including cutting stock problems*, Operations Research 38, 922-923 (1990).
- [15] Feillet D., Dejax P., Gendreau M., Guegen C., *An exact algorithm for the elementary shortest path problem with resource constraints: application to some vehicle routing problems* Networks 44, 216-229 (2004).
- [16] Garey M.R., Johnson D.S. *Computers and intractability: a guide to the theory of NP-completeness*, Freeman & co., New York (1979).
- [17] Gélinas S., Desrochers M., Desrosiers J., Solomon M.M., *A new branching strategy for time constrained routing problems with application to backhauling* Cahiers du GERAD G-92-13, HEC Montréal, 1992.
- [18] Goetschalckx M., Jacobs-Blecha C., *The vehicle routing problem with backhauls*, European Journal of Operational Research 42, 39-51 (1989).
- [19] Golden B., Baker E., Alfaro J., Schaffer J., *The vehicle routing problem with backhauling: two approaches*, Proc. of the Twenty-first annual meeting of S.E.TIMS, R.D.Hammesfahr ed., Myrtle Beach, USA, 90-92 (1985).
- [20] Irnich S., Villeneuve D., *The shortest path problem with resource constraints and k-cycle elimination for $k \geq 3$* , Cahiers du GERAD G-2003-55, HEC Montréal, 2003.
- [21] Mingozzi A., Giorgi S., Baldacci R., *An exact method for the vehicle routing problem with backhauls*, Transportation Science 33, 315-329 (1999).
- [22] Rousseau L.M., Gendreau M., Feillet D., *Interior point stabilization for column generation* Technical report, LIA - Université d'Avignon, 2003.
- [23] Toth P., Vigo D., *An exact algorithm for the vehicle routing problem with backhauls*, Transportation Science 31, 372-385 (1997) .

- [24] Toth P., Vigo D., *A heuristic algorithm for the symmetric and asymmetric vehicle routing problem with backhauls*, European Journal of Operational Research 113, 528-543 (1999).
- [25] Toth P., Vigo D. (eds), *The vehicle routing problem*, SIAM Monographs on Discrete Mathematics and Applications (2002).
- [26] Vanderbeck F., Wolsey L.A., *An exact algorithm for IP column generation*, Operations Research Letters 19, 151-159 (1996).
- [27] Wade A.C., Salhi S., *An investigation into a new class of vehicle routing problem with backhauls* Omega 30, 6, 479-487 (2002).
- [28] Wolsey L., *Integer Programming*, Wiley-Interscience, USA 1998

Algorithm	Instance					Root node			Search tree					
	<i>Name</i>	<i>N</i>	<i>K</i>	$\bar{\sigma}$	<i>LB gap</i>	<i>LB impr.</i>	T_{root}	C_{root}	<i>UB</i>	<i>Gap</i>	T_{tree}	<i>Nodes</i>	T_{price}	C_{tree}
EXACT	c101_20_02	20	4	10	14.72	0.37	626.68	119	272	0.00	885.84	3	99.92	178
EXACT	c101_20_08	20	4	9	16.04	0.36	453.57	86	279	0.00	664.04	5	99.84	188
EXACT	r101_20_02	20	3	11	4.54	0.30	13076.00	1220	329	0.00	13076.00	1	99.96	1220
EXACT	r101_20_08	20	3	11	12.80	0.00	4646.14	197	342	0.00	4646.14	1	99.99	197
EXACT	rc101_20_02	20	5	7	13.02	2.10	28.42	126	428	0.00	892.00	187	96.54	1329
EXACT	rc101_20_08	20	5	7	17.54	0.00	20.76	37.00	458	0.00	20.76	1	98.51	37
RELAX	c101_20_02	20	4	10	14.09	1.08	4.67	255	272	0.00	58.13	40	89.87	983
RELAX	c101_20_08	20	4	9	15.43	1.09	14.90	152	279	0.00	460.60	192	91.30	5272
RELAX	r101_20_02	20	3	11	9.19	0.27	29.19	1222	329	0.00	112.75	3	88.93	2388
RELAX	r101_20_08	20	3	11	12.55	0.29	32.21	1704	342	0.00	136.73	5	98.13	1988
RELAX	rc101_20_02	20	5	7	12.82	2.27	6.25	239	428	0.00	448.09	249	92.81	1281
RELAX	rc101_20_08	20	5	7	17.36	0.00	7.61	152	458	0.00	7.61	1	94.09	152
RELAX	c101_40_02	40	8	10	13.78	0.34	79.74	569	551	2.35	*	360	87.75	20135
RELAX	c101_40_08	40	8	10	12.67	0.10	32.21	2388	569	0.00	1713.08	213	95.61	5312
RELAX	r101_40_02	40	6	14	12.60	0.00	834.41	8685	601	0.00	834.41	1	96.37	8685
RELAX	r101_40_08	40	6	14	16.61	0.00	825.25	239	629	0.00	825.25	1	99.80	239
RELAX	rc101_40_02	40	9	10	8.43	1.00	7.61	152	886	0.14	*	650	96.86	3039
RELAX	rc101_40_08	40	9	10	11.47	1.13	43.97	295	926	0.09	*	465	96.77	18385

Table 1: Computational results of algorithms EXACT and RELAX on Class 1

Instance					Root node			Search tree					
<i>Name</i>	<i>N</i>	<i>K</i>	$\bar{\sigma}$	<i>LB gap</i>	<i>LB impr.</i>	<i>T_{root}</i>	<i>C_{root}</i>	<i>UB</i>	<i>Gap</i>	<i>T_{tree}</i>	<i>Nodes</i>	<i>T_{price}</i>	<i>C_{tree}</i>
2S_20_50_1	20	6	4	12.06	0.00	0.32	4	181689	0.00	0.32	1	68.75	4
2S_20_50_2	20	4	6	13.40	0.00	1.73	466	151472	0.00	1.73	1	64.67	466
2S_20_50_3	20	3	8	12.28	1.88	21.69	301	136107	0.00	100.66	10	96.75	408
2S_20_66_1	20	7	4	9.02	0.00	0.14	9	189396	0.00	0.14	1	64.29	9
2S_20_66_2	20	5	6	12.97	0.00	3.85	55	155853	0.00	3.85	1	88.57	55
2S_20_66_3	20	4	8	9.75	1.91	29.86	187	136489	0.00	119.71	15	97.34	1056
2S_20_80_1	20	8	4	9.32	2.41	0.33	2	210732	0.00	7.94	41	65.41	124
2S_20_80_2	20	6	6	13.16	0.00	1.94	252	166408	0.00	1.94	1	84.02	252
2S_20_80_3	20	4	7	11.55	2.57	20.41	137	147820	0.00	229.00	44	96.04	2396
2S_40_50_1	40	10	4	10.02	0.02	2.12	16	357430	0.00	4.06	3	55.67	25
2S_40_50_2	40	7	6	12.28	0.00	3.42	292	269590	0.00	3.42	1	71.54	293
2S_40_50_3	40	5	8	13.66	0.00	68.71	790	229044	0.00	68.71	1	91.53	790
2S_40_66_1	40	13	4	10.77	0.00	0.88	22	377279	0.00	0.88	1	34.09	22
2S_40_66_2	40	9	6	13.55	2.06	27.80	359	291008	0.54	*	687	76.51	5867
2S_40_66_3	40	7	8	13.94	0.37	165.37	668	241347	0.00	320.78	6	95.92	1832
2S_40_80_1	40	16	4	9.36	0.00	3.33	8	425911	0.00	3.33	1	53.75	8
2S_40_80_2	40	11	6	13.75	0.03	15.49	169	324920	0.00	42.07	4	88.80	286
2S_40_80_3	40	8	8	14.23	0.11	184.14	1329	270313	0.00	951.16	14	95.14	1791

Table 2: Computational results of algorithm RELAX on Class 2S

Instance					Root node			Search tree					
<i>Name</i>	<i>N</i>	<i>K</i>	$\bar{\sigma}$	<i>LB gap</i>	<i>LB impr.</i>	T_{root}	C_{root}	<i>UB</i>	<i>Gap</i>	T_{tree}	<i>Nodes</i>	T_{price}	C_{tree}
2C_20_50_1	20	11	2	15.61	0.22	0.23	7	265504	0.00	0.90	7	72.33	14
2C_20_50_2	20	8	3	16.75	0.00	0.24	15	206425	0.00	0.24	1	85.14	15
2C_20_50_3	20	6	4	14.66	0.96	1.12	65	171236	0.00	2.67	3	62.55	100
2C_20_66_1	20	14	2	20.97	0.80	0.13	0	298493	0.00	0.71	5	86.43	15
2C_20_66_2	20	7	3	12.86	0.00	0.34	7	192727	0.00	0.34	1	72.11	7
2C_20_66_3	20	6	4	18.65	0.16	0.57	60	178629	0.00	5.84	5	79.32	121
2C_20_80_1	20	14	2	22.13	0.00	0.18	11	304412	0.00	0.18	1	79.36	11
2C_20_80_2	20	8	3	18.33	0.81	0.51	17	218072	0.00	11.52	8	83.18	79
2C_20_80_3	20	6	4	14.73	0.00	1.26	39	177215	0.00	1.26	1	74.60	39
2C_40_50_1	40	22	2	19.25	0.72	0.33	0	603715	0.00	18.92	32	79.42	35
2C_40_50_2	40	15	3	14.87	0.00	1.72	43	402309	0.00	1.72	1	67.82	43
2C_40_50_3	40	11	4	17.43	0.00	3.62	14	334830	0.00	3.62	1	82.50	14
2C_40_66_1	40	24	2	20.97	0.00	0.60	3	630257	0.00	0.60	1	82.71	3
2C_40_66_2	40	15	3	18.48	0.00	1.73	8	426517	0.00	1.73	1	55.49	8
2C_40_66_3	40	11	4	17.17	0.76	9.28	85	331459	0.00	564.23	155	77.89	758
2C_40_80_1	40	25	2	23.59	0.00	0.82	7	647539	0.00	0.82	1	83.11	7
2C_40_80_2	40	15	3	18.27	0.23	1.12	0	424368	0.00	17.61	19	71.62	71
2C_40_80_3	40	11	4	16.44	0.00	10.15	58	332957	0.00	236.52	10	71.64	175

Table 3: Computational results of algorithm RELAX on Class 2C

Instance					Root node			Search tree					
<i>Name</i>	<i>N</i>	<i>K</i>	$\bar{\sigma}$	<i>LB gap</i>	<i>LB impr.</i>	T_{root}	C_{root}	<i>UB</i>	<i>Gap</i>	T_{tree}	<i>Nodes</i>	T_{price}	C_{tree}
3S_20_50_1	20	6	7	15.64	1.98	9.46	29	8769	0.00	137.89	26	97.24	487
3S_20_50_2	20	4	9	20.83	0.00	103.98	104	7986	0.00	103.98	1	99.58	104
3S_20_50_3	20	3	12	14.75	0.00	869.61	160	6445	0.00	869.61	1	99.97	160
3S_20_66_1	20	7	7	13.39	0.96	12.56	19	9129	0.00	85.28	14	97.99	303
3S_20_66_2	20	5	9	9.19	0.00	165.75	85	7470	0.00	165.75	1	99.60	85
3S_20_66_3	20	3	12	25.47	4.15	389.18	176	8346	0.00	1345.76	4	99.92	458
3S_20_80_1	20	8	7	16.73	0.00	7.01	21	10707	0.00	7.01	1	96.86	21
3S_20_80_2	20	6	9	25.22	0.00	93.34	57	10093	0.00	93.34	1	99.42	57
3S_20_80_3	20	4	11	11.77	0.00	481.41	156	7058	0.00	481.41	1	99.89	156
3S_40_50_1	40	10	10	22.67	0.00	220.14	91	18282	0.00	220.14	1	99.01	91
3S_40_50_2	40	8	12	20.30	0.00	491.65	48	14603	0.00	2182.29	5	99.86	431
3S_40_50_3	40	5	16	21.72			4921	11610		*		99.54	4921
3S_40_66_1	40	12	10	18.01	0.00	110.41	12	18370	0.00	110.41	1	98.97	12
3S_40_66_2	40	9	12	19.23	0.00	1229.17	85	15307	0.00	1229.17	1	99.82	85
3S_40_66_3	40	6	15	14.67	0.00	3599.00	48	11725	0.00	3599.00	1	98.96	48
3S_40_80_1	40	17	10	15.60	0.00	153.13	27	20665	0.00	153.13	1	99.29	27
3S_40_80_2	40	12	11	18.47	0.77	1050.12	211	17599	0.88	*	17	99.79	586
3S_40_80_3	40	8	15	16.78	0.66	2714.01	315	13317	5.23	*	4	99.76	611

Table 4: Computational results of algorithm RELAX on Class 3S

Instance					Root node			Search tree					
<i>Name</i>	<i>N</i>	<i>K</i>	$\bar{\sigma}$	<i>LB gap</i>	<i>LB impr.</i>	T_{root}	C_{root}	<i>UB</i>	<i>Gap</i>	T_{tree}	<i>Nodes</i>	T_{price}	C_{tree}
3C_20_50_1	20	11	5	17.57	0.00	1.11	21	12720	0.00	1.11	1	88.29	21
3C_20_50_2	20	7	6	28.12	0.00	4.95	5	11559	0.00	4.95	1	97.98	5
3C_20_50_3	20	6	8	14.91	0.83	74.52	16	8387	0.00	338.12	7	99.78	134
3C_20_66_1	20	12	5	23.99	0.27	1.01	0	14578	0.00	21.91	5	98.19	17
3C_20_66_2	20	8	6	23.22	0.44	7.79	3	11178	0.00	171.12	10	99.61	172
3C_20_66_3	20	5	8	16.88	0.00	178.01	47	8160	0.00	178.01	1	99.25	47
3C_20_80_1	20	10	5	20.87	0.21	6.28	7	12802	0.00	45.71	7	96.48	23
3C_20_80_2	20	8	6	15.99	1.31	4.68	4	10087	0.00	96.47	16	98.46	64
3C_20_80_3	20	5	8	18.02	0.00	399.11	67	8317	0.00	399.11	1	99.67	67
3C_40_50_1	40	22	6	23.81	0.00	44.56	7	27559	0.00	44.56	1	99.85	4
3C_40_50_2	40	16	8	20.28	0.81	73.07	9	21773	0.00	910.80	18	99.33	1198
3C_40_50_3	40	10	10	19.82	0.01	683.18	12	15629	0.22	*	12	99.87	1450
3C_40_66_1	40	22	6	19.91	0.00	39.45	12	25981	0.00	39.45	1	97.13	12
3C_40_66_2	40	15	8	21.93	1.05	113.16	8	21319	0.00	2784.34	159	99.07	815
3C_40_66_3	40	10	10	18.83	0.00	1769.13	273	15293	0.00	1769.13	1	99.90	273
3C_40_80_1	40	21	6	19.67	0.05	9.55	5	26273	0.00	112.55	4	99.11	11
3C_40_80_2	40	16	8	17.24	1.15	56.13	11	20652	0.74	*	58	99.13	312
3C_40_80_3	40	10	10	18.64	0.34	1412.71	27	15365	0.00	2711.09	9	99.62	57

Table 5: Computational results of algorithm RELAX on Class 3C