Les Cahiers du GERAD ISSN: 0711–2440

Selective arc-ng pricing for vehicle routing

L. Costa, C. Contardo,
G. Desaulniers, D. Pecin

G–2020–07

January 2020

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2020
– Bibliothèque et Archives Canada, 2020

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2020
– Library and Archives Canada, 2020

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Selective arc-ng pricing for vehicle routing

Luciano Costa [a]
Claudio Contardo [b]
Guy Desaulniers [a]
Diego Pecin [c]

[a] *GERAD & Département de mathématiques et de génie industriel, Polytechnique Montréal, Montréal (Québec) Canada, H3C 3A7*

[b] *GERAD, CIRRELT & Département de management et technologie, ESG UQÀM, Montréal (Québec) Canada, H2X 3X2*

[c] *Department of Econometrics, Erasmus School of Economics (ESE), Erasmus University Rotterdam, Rotterdam 3000 DR, The Netherlands*

luciano.costa@gerad.ca
claudio.contardo@gerad.ca
guy.desaulniers@gerad.ca
galindopecin@ese.eur.nl

January 2020
Les Cahiers du GERAD
G–2020–07

Abstract: Column generation algorithms for solving vehicle routing problems often rely on a relaxed pricing subproblem where routes may be non-elementary and which is solved by a labeling algorithm. This pricing algorithm is said to be selective if it can discard non-elementary paths that may be Pareto-optimal but guarantees finding at least one (not necessarily elementary) path of negative reduced cost when there exists at least one elementary path of negative reduced cost. In this paper, we propose a selective pricing mechanism for a recently introduced variant of the *ng*-route relaxation, in which the neighborhoods are associated with arcs instead of nodes. We extend a state-of-the-art set-based dominance criterion for this problem and show, by means of an exhaustive computational campaign, that the resulting mechanism is effective at reducing the number of non-dominated labels as compared to the original pricing algorithm for the same problem. As a result, typically shorter computing times are required to compute similar lower bounds when the proposed mechanism is embedded within a column generation solver.

Keywords: Column generation, shortest path problem with resource constraints, selective pricing, vehicle routing

Acknowledgments: This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada [Grant 2017–05683]. This support is gratefully acknowledged.

1 Introduction

The vehicle routing problem (VRP) is a classic optimization problem arising in many applications in logistics. Since the seminal work of Dantzig and Ramser (1959), many researchers and practitioners have attempted to tackle this problem as efficiently as possible. Unfortunately, solving the VRP is $\mathcal{NP}$-hard as the traveling salesman problem (TSP, Dantzig et al., 1954) can be reduced to a VRP in polynomial time. Over the years, several variants of the VRP have been proposed to model the different attributes associated with real-life problems, namely: heterogeneous fleet, multiple depots, combined pickup and delivery, split services, and vehicles with special capabilities (e.g., electric vehicles), among others. For recent compendiums and a broad view of the recent trends in vehicle routing applications, the reader is referred to Toth and Vigo (2014) and Vidal et al. (2019).

Nowadays, the dominant methodology for solving many variants of the VRP is branch-price-and-cut applied to a set-partitioning-type formulation (Costa et al., 2019), which can be described generically as follows. Let N^+ be a set of customers that need to be serviced by exactly one vehicle each. Let Ω be the set of feasible vehicle routes, where the feasibility of a route depends on the VRP variant considered. With each route $r \in \Omega$, associate a cost c_r and, for each customer $i \in N^+$, a binary parameter a_i^r that indicates whether or not customer i is visited in route r. Furthermore, define a binary variable ω_r, which is equal to 1 if route r is selected in the solution, or 0 otherwise. With this notation, a pure set-partitioning formulation for a VRP is as follows:

$$\min \quad \sum_{r\in\Omega} c_r\omega_r \tag{1}$$

$$\text{s.t.} \quad \sum_{r\in\Omega} a_i^r\omega_r = 1 \qquad \forall i \in N^+ \tag{2}$$

$$\omega_r \in \{0,1\} \qquad \forall r \in \Omega. \tag{3}$$

The objective function (1) aims at minimizing the total routing cost. Constraints (2) impose a single visit to each customer $i \in N^+$. Finally, constraints (3) define the domain of the variables. For certain problem variants, additional constraints and variables may be required, yielding a set-partitioning-type model. Note also that, under certain conditions, model (1)–(2) can be replaced by a set-covering model, obtained by replacing the equality in (2) by a greater-than-or-equal relation.

Formulations like (1)–(3) have an exponential number of variables with respect to the number of customers. Handling these variables all at once may be computationally prohibitive. This drawback can be overcome using branch-and-price, that is a branch-and-bound algorithm applying column generation to solve the linear relaxations encountered throughout the search tree. Column generation is an iterative algorithm that solves at each iteration a restricted master problem (RMP) and a pricing subproblem. The RMP is defined as the linear relaxation of (1)–(3) considering only a subset of its variables. It is solved by a linear programming solver to yield a pair of primal and dual solutions. The pricing subproblem looks for negative reduced cost variables to add to the RMP. For most VRPs, it is cast as an elementary shortest path problem with resource constraints (ESPPRC) defined over an application-dependent network and solved by a labeling algorithm. When negative reduced cost are identified, they are added to the RMP before starting a new iteration. Otherwise, the column generation algorithm stops and the current RMP solution yields a lower bound. To strengthen the linear relaxations, cutting planes can be added dynamically to yield a branch-price-and-cut algorithm.

As mentioned above, the pricing subproblem is often an ESPPRC, which is strongly $\mathcal{NP}$-hard (Dror, 1994). At the opposite, the subproblem arising from completely dropping the elementarity constraints, called the shortest path problem with resource constraints (SPPRC), is solvable in pseudo-polynomial time (Desrosiers et al., 1984), but proves to be weak when the resource constraints are not sufficiently tight. To overcome this issue, several authors have focused their attention on strengthening the bounds by partially imposing path elementarity. Two main frameworks can be found in the literature to deal with the issue of partially addressing the elementarity constraints in an efficient manner: decremental state-space relaxation (DSSR, Boland et al., 2006; Righini and Salani, 2006); and cycle elimination (Desrochers et al., 1992; Irnich and Villeneuve, 2006; Baldacci et al., 2011; Bulhões et al., 2018).

In DSSR, the elementarity constraints are imposed gradually. Starting from a pure SPPRC, the path with the least reduced cost is checked for feasibility. If it is, an optimal solution has been found; otherwise, a cycle is detected and the elementarity requirements are strengthened for the subsequent iterations. Cycle elimination techniques, in turn, are closely related to the notion of route relaxations. In the seminal article of Desrochers et al. (1992), the authors presented the first successful application of column generation for VRPs. They considered the vehicle routing problem with times windows (VRPTW, Desaulniers et al., 2014) and solved by branch-and-price. The authors recognized the difficulty of the underlying pricing subproblem and described for the first time the SPPRC without cycles of length 2 (*2-cyc*-SPPRC). Unlike the ESPPRC, this problem can be solved in pseudo-polynomial time. Later, Irnich and Villeneuve (2006) studied the *k-cyc*-SPPRC, for values of k that range between 3 and 5, which provides bounds that are much stronger than those obtained by the *2-cyc*-SPPRC, yet, in longer computing times. Baldacci et al. (2011) introduced the *ng*-route relaxation (*ng*-SPPRC), which relies on a memory mechanism to remember nodes that have been visited recently within a route. Thus, a feasible path may still contain cycles, but those cycles should be long and have a limited negative impact on the resulting bounds. Some hybrid strategies have been proposed by Martinelli et al. (2014), who successfully achieved improved results by adapting DSSR to tackle the *ng*-SPPRC, and Contardo et al. (2015), who showed that elementary paths can be obtained by imposing partial elementarity in an escalated way within a column-generation-based solver for routing problems.

More recently, Bulhões et al. (2018) proposed the *arc-ng*-SPPRC, an extension of the classical *ng*-SPPRC, in which the neighborhoods are associated with arcs instead of nodes. The authors proposed a novel set-based dominance mechanism. Arc-based neighborhoods allow a better management of the undesirable cycles. However, when embedded into a DSSR framework, they usually induce many more iterations compared to a traditional node-based *ng*-SPPRC.

Desaulniers et al. (2019) introduced the concept of *selective* pricing when the elementarity constraints are dropped at least partially. A pricing algorithm (associated with a column generation pricing subproblem) is said to be *selective* if it can discard non-elementary paths that may be Pareto-optimal but guarantees finding at least one path (possibly with cycles) of negative reduced cost when there exists at least one elementary path with a negative reduced cost. Such an algorithm can stop when it proves that no such elementary paths exist, even if there are still some non-elementary paths with a negative reduced cost. A selective pricing strategy can be used to derive valid dual bounds, and those bounds have the potential of being stronger than those that would be achieved using a non-selective pricing strategy. The authors illustrate the selective pricing strategy on the *ng*-SPPRC, but the algorithm is quite demanding and requires the addition of several data structures and subprocedures, whose overhead is not always beneficial in the long run.

In this article, we extend the set-based dominance criterion of Bulhões et al. (2018) for the *arc-ng*-SPPRC by making it selective. Unlike the original approach of using additional data structures to keep track of the non-dominated labels (Desaulniers et al., 2019), the proposed mechanism requires minimal modifications of the labeling algorithm and is therefore easy to implement. As a result, the proposed selective pricing strategy proves to be faster in practice than the original (non-selective) *arc-ng*-SPPRC when embedded within a column generation algorithm for solving benchmark instances of the VRPTW.

The remainder of this article is organized as follows: Section 2 presents some of the main route relaxations proposed in the literature, with an emphasis on the *arc-ng*-SPPRC. In Section 3, we describe the selective *arc-ng*-SPPRC, provide some theoretical proofs of its correctness, and discuss how this new framework extends the traditional *arc-ng*-SPPRC. In Section 4, we report the results of our computational experiments to show the impact of considering the new framework when tackling some hard VRPTW instances. Finally, some conclusions are drawn in Section 5.

2 Route relaxations

As discussed in the previous section, the natural way of modeling pricing subproblems arising from most VRP applications is by considering an ESPPRC. However, due to the high computational complexity of this problem, route relaxations have been preferred to design efficient column-generation-based algorithms. In this section, we discuss some of the main route relaxations proposed in the literature, namely: the SPPRC, which is obtained when all elementarity requirements are relaxed; and the *ng*-SPPRC and *arc-ng*-SPPRC, which tend to avoid the formation of cycles containing nodes that might be located close to each other. To simplify the exposition, we consider that the pricing subproblem is defined over a digraph $G = (N, A)$, where $N = \{0, \ldots, n, n+1\}$ is its set of nodes and $A \subseteq N \times N$ its set of arcs. Nodes 0 (source) and $n+1$ (sink) represent the origin and the destination of a path, respectively, whereas $N^+ = N \setminus \{0, n+1\}$ is the set of customer nodes. The following discussion can be easily extended to other networks.

2.1 The SPPRC

The SPPRC can be formally described as follows. Consider network G and let $\mathcal{R}$ be a set of resources, which are used to model some route feasibility rules for the problem at hand, such as vehicle capacity, time windows, or precedence, among others (Irnich and Desaulniers, 2005). The objective of the SPPRC is to find a least-cost path, not necessarily elementary, from node 0 to node $n+1$, in such a way that the path respects the following resource constraints.

With each node $v \in N$ and resource $r \in \mathcal{R}$, define resource windows $[e_v^r, l_v^r]$, with $0 \leq e_v^r \leq l_v^r$, representing the allowed limits on the consumption of each resource r by a path ending at node v. Specifically, it is assumed that $l_0^r = 0$ and $l_{n+1}^r = H^r$, where H^r represents the maximum consumption allowed for resource r when arriving at the sink.

With each arc $(u, v) \in A$, we associate a modified routing cost $\bar{c}_{uv} \in \mathbb{R}$ that depends on the RMP dual values such that the sum of the modified costs of the arcs composing a path between nodes 0 and $n+1$ is equal to the reduced cost of the corresponding route variable. For example, if we denote by α_i, $i \in N^+$, the dual variables associated with constraints (2), then the modified cost of arc $(u, v) \in A$ is given by:

$$\bar{c}_{uv} = \begin{cases} c_{uv} - \alpha_u & \text{if } u \in N^+ \\ c_{uv} & \text{otherwise (i.e., } u = 0), \end{cases} \tag{4}$$

where c_{uv} is the routing cost along arc (u, v). With each arc (u, v), we also associate a vector of resource consumptions $(\gamma_{uv}^r)_{r \in \mathcal{R}} \in \mathbb{N}_+^{|\mathcal{R}|}$ (e.g., travel time, load delivered, etc.). In many classical SPPRC variants, it is assumed that, if a path ending at node v uses less than e_v^r for some resource $r \in \mathcal{R}$, it is still possible to consume the extra amount of resource, so as to respect the lower bound e_v^r. In other words, when a path is extended along an arc (u, v), assuming the resource consumptions at node u given by ξ_u^r, $\forall r \in \mathcal{R}$, they are updated at node v with the resource extension functions $\xi_v^r \leftarrow \max\{e_v^r, \xi_u^r + \gamma_{uv}^r\}$, $\forall r \in \mathcal{R}$. The extension is deemed *resource-feasible* (i.e., it satisfies the resource constraints) if $e_v^r \leq \xi_v^r \leq l_v^r$, $\forall r \in \mathcal{R}$, otherwise it is deemed infeasible. Note that, depending on the problem at hand, the resources may vary along the paths according to more complex resource extension functions (Irnich, 2008). For the sake of simplicity and without loss of generality, we limit our discussion to the above resource extension functions.

2.2 The ng-SPPRC of Baldacci et al. (2011)

Currently, the state-of-the-art –and widely adopted– framework to eliminate cycles is the *ng*-route relaxation (*ng*-SPPRC) proposed by Baldacci et al. (2011). The *ng*-SPPRC differs from the *2-cyc*-SPPRC and *k-cyc*-SPPRC because the notion of length of a cycle does not depend on the number of arcs (or edges) traversed along the cycle, but rather on the notion of proximity between the nodes that belong to it. In the *ng*-SPPRC, cycles are forbidden whenever they contain customers that are close to each other. For every customer $v \in N^+$, one considers a *ng*-set (neighborhood) $\mathcal{N}_v \subseteq N^+$. Because the goal is to prevent short cycles, it is common to define the *ng*-set $\mathcal{N}_v$ as containing the Δ

geographically closest customers to v, and the vertex v itself, where $\Delta > 0$ is a predefined parameter. Let $P = \{v_1, \ldots, v_h = v, \ldots, v_{h'} = v, \ldots, v_\ell\}$ be a path generated while solving the *ng*-SPPRC. The cycle $H = \{v_h = v, \ldots, v_{h'} = v\}$ is feasible if and only if $v \notin \cap_{i=h+1}^{h'-1} \mathcal{N}_{v_i}$, i.e., v does not belong to the *ng*-set of at least one intermediate node in the cycle. The theoretical complexity of the *ng*-SPPRC is also pseudo-polynomial if $|\mathcal{N}_v| \leq \Delta + 1$ for every $v \in N^+$. In this case, a labeling algorithm can solve the *ng*-SPPRC in pseudo-polynomial time with a multiplying constant that depends on $2^{2\Delta}$. The *ng*-SPPRC normally provides much tighter bounds than those provided by the *k-cyc*-SPPRC in comparable CPU times. In fact, experiments carried out by Poggi and Uchoa (2014) using some hard capacitated VRP instances have shown that *ng*-sets with $\Delta = 8$ yield bounds similar to those obtained by the *5-cyc*-SPPRC, but spending a time comparable to the *4-cyc*-SPPRC. For this reason, the *ng*-SPPRC has become the route relaxation of preference ever since.

2.3 The arc-ng-SPPRC of Bulhões et al. (2018)

In a recent article by Bulhões et al. (2018), the *ng*-SPPRC is extended to consider neighborhoods associated with arcs instead of nodes. We call this problem the *ng*-SPPRC with arc-based neighborhoods, and denote it *arc-ng*-SPPRC. The *arc-ng*-SPPRC can be defined as follows. With every arc $a = (u, v) \in A$, we associate a *ng*-set $\mathcal{N}_a \subseteq N^+$ that corresponds to the set of customer nodes that can be remembered by arc a. The *ng*-sets $\mathcal{N}_a$ may be defined according to any suitable criterion. Most successful implementations use some notion of distance to favor neighborhoods containing nodes that are close to each other. An example is to consider the set of nodes which are close to both extremities of the arcs, i.e., $\mathcal{N}_a \subseteq \mathcal{N}_u \cap \mathcal{N}_v$, with $\mathcal{N}_u$ and $\mathcal{N}_v$ defined as for the *ng*-SPPRC. Let $P = (a_0, a_1, \ldots, a_p)$ be a partial path, composed of arcs $a_0 = (0, v_1)$, $a_1 = (v_1, v_2)$, $\ldots$, $a_p = (v_p, v_{p+1})$ and $\pi(v_p)$ be the memory of this path upon arrival at node v_p. An extension using arc $a = (v_p, v_{p+1})$ is deemed *arc-ng-feasible* only if $v_{p+1} \notin \pi(v_p)$. The memory upon arrival to v_{p+1} is updated according to the formula $\pi(v_{p+1}) \leftarrow (\pi(v_p) \cap \mathcal{N}_a) \cup \{v_{p+1}\}$. If $\mathcal{N}_a = \mathcal{N}_u \cap \mathcal{N}_v$, the *arc-ng*-SPPRC coincides with the classical *ng*-SPPRC, and, therefore, it is more general.

By considering the notions presented in Section 2.1, a path $P' = (v_1 = 0, \ldots, v_{p+1} = n + 1)$ is said to be feasible only if every prefix $P'_i = (v_1, \ldots, v_i)$, $i \in \{2, \ldots, p+1\}$ is both resource- and *ng*-feasible. The cost of P' is given by $c(P') = \sum_{1 \leq q \leq p} c_{v_q, v_{q+1}}$. The *arc-ng*-SPPRC consists of finding a feasible path P' of minimum total cost.

Not only this mechanism allows a better management of the undesired cycles, but also yields a sharper dominance rule. Indeed, while the traditional dominance rule for the *ng*-SPPRC allows only pairwise comparisons, Bulhões et al. (2018) introduce a novel set-based dominance criterion that includes the former as a particular case.

2.3.1 Labeling algorithm

A general description of a labeling algorithm similar to the one adopted by Bulhões et al. (2018) to solve the *arc-ng*-SPPRC is as follows. A label L encodes a partial path $(v_1 = 0, \ldots, v_p)$ from the source until reaching a node v_p through the following data structures: a terminal node $v(L) = v_p$; resource consumptions $\xi^r(L)$, $r \in \mathcal{R}$; a (reduced) cost $\bar{c}(L)$; a memory $\Pi(L)$; and a pointer $pred(L)$, which indicates the predecessor of label L'.

An extension to a node w over an arc $a = (v_p, w)$ yields a label L' with the associated data structures computed as follows:

$$v(L') \leftarrow w \tag{5}$$

$$\xi^r(L') \leftarrow \max\{e^r_w, \xi^r(L) + \gamma^r_a\} \qquad \forall r \in \mathcal{R} \tag{6}$$

$$\bar{c}(L') \leftarrow \bar{c}(L) + \bar{c}_{v_p w} \tag{7}$$

$$\Pi(L') \leftarrow (\Pi(L) \cap \mathcal{N}_a) \cup \{w\} \tag{8}$$

$$pred(L') \leftarrow L. \tag{9}$$

Label L' is deemed feasible if and only if it is both resource- and *ng*-feasible. For the sake of conciseness, we denote from now on an extension of label L to a node w by $L \oplus w$.

A generic representation of a labeling algorithm analogous to the one considered by Bulhões et al. (2018) is outlined in Algorithm 1. Although they have implemented a bi-directional labeling algorithm (Righini and Salani, 2006), we present a mono-directional (forward) labeling algorithm for the sake of simplicity and because it illustrates the main parts of their algorithm. Similarly to what is done in most of the recent implementations (e.g., Pecin et al., 2017b,a), Bulhões et al. adopt label buckets to store non-dominated labels. This data structure allows a more efficient application of dominance rules. For each node $v \in N$ and possible consumption ξ^{r^*} of a given resource $r^* \in \mathcal{R}$, one defines a bucket $\mathcal{B}(v, \xi^{r^*})$ that stores all the non-dominated labels associated with node v and having consumption ξ^{r^*} for resource r^*. It is initialized as empty and enlarged dynamically. The choice for the resource defining the buckets may vary according to the problem at hand (e.g., load or time). In Algorithm 1, the set $\mathcal{V}$ keeps the unextended labels waiting in a queue. This set is kept sorted at all times in lexicographic order with regard to the consumption of resource r^*, i.e., the labels with lowest consumptions of r^* are positioned before those with larger consumptions (Desrochers and Soumis, 1988). As mentioned before, the maximum consumption allowed for resource r^* is H^{r^*}. The set $\mathcal{A}(L) = \{a = (u, v) \in A : u = v(L), v \in N \setminus \Pi(L)$ and $p(L) \oplus v$ is resource-feasible$\}$ contains the arcs that can be used to extend L. For two labels L and L', $L \prec L'$ means that L dominates L' (according to the rules defined in Sections 2.3.2 and 3). Let L_0 be the label encoding an empty path of cost zero, with zero resource consumptions, empty memory and such that $pred(L_0) = nil$. Algorithm 1 returns the label L_{best} representing a path with lowest cost. This labeling algorithm extends non-dominated labels to create new labels (line 8), coupled with a dominance rule to discard non-promising labels (lines 12 and 14). The dominance procedure prevents a combinatorial explosion and, therefore, keeps the computational burden within reasonable limits.

Algorithm 1 Labeling algorithm

1: $\mathcal{V} \leftarrow \{L_0\}$, $L_{best} \leftarrow nil$, $\bar{c}_{min} \leftarrow +\infty$,
2: $\mathcal{B}(v, \xi^{r^*}) \leftarrow \emptyset$, $\forall v \in N, \forall \xi^{r^*} \in \{0, 1, \ldots, H^{r^*}\}$
3: **while** $\mathcal{V} \neq \emptyset$ **do**
4: Let L be the first label in $\mathcal{V}$
5: Set $\mathcal{V} \leftarrow \mathcal{V} \setminus \{L\}$
6: Set $\mathcal{B}(v(L), \xi^{r^*}(L)) \leftarrow \mathcal{B}(v(L), \xi^{r^*}(L)) \cup \{L\}$
7: **for all** $a \in \mathcal{A}(L)$ **do**
8: Let L' be the label resulting from extending L along arc a using (5)–(9)
9: **if** $v(L') = n + 1$ **then**
10: **if** $\bar{c}(L') < \bar{c}_{min}$ **then**
11: Set $L_{best} \leftarrow L', \bar{c}_{min} \leftarrow \bar{c}(L')$
12: **else if** there exists no label $L'' \in \mathcal{B}(v(L'), \xi^{r^*}(L'))$ such that $L'' \prec L'$ **then**
13: Set $\mathcal{B}(v(L'), \xi^{r^*}(L')) \leftarrow \mathcal{B}(v(L'), \xi^{r^*}(L')) \cup \{L'\}$
14: Remove from $\mathcal{V}$ all labels L'' such that $L' \prec L''$
15: **return** L_{best}

2.3.2 Multiple partial label dominance

Dominance in Bulhões et al. (2018) is not only performed pairwise. The authors use a sharper *multiple partial label dominance rule* similar to that introduced by Irnich and Villeneuve (2006) for the *k-cyc*-SPPRC. To formalize this idea in the context of the *arc-ng*-SPPRC, we use the notion of *weak* and *strong* dominance described in Irnich and Villeneuve (2006). Let L and L' be two feasible labels. Label L is said to *weakly dominate* L' if all the following conditions hold:

$$v(L) = v(L'), \tag{10}$$

$$\xi^r(L) \leq \xi^r(L'), \qquad \forall r \in \mathcal{R} \tag{11}$$

$$\bar{c}(L) \leq \bar{c}(L'). \tag{12}$$

Conditions (10)–(12), however, do not ensure a proper dominance of L' by L. In fact, depending on the label memories $\Pi(L)$ and $\Pi(L')$, it may still be possible for L' to be extended to a node w that would not be a *ng*-feasible extension of L. This would happen for every node $w \in \Pi(L) \setminus \Pi(L')$. The additional condition

$$\Pi(L) \subseteq \Pi(L') \tag{13}$$

ensures the correctness of the pairwise dominance.

Bulhões et al. (2018) realized that condition (13) may be unnecessarily too restrictive. In fact, if a label L weakly dominates another label L', one may restrict the extensions of L' only to the nodes $w \in \Pi(L) \setminus \Pi(L') \cup \{j \in N \setminus \Pi(L') \mid \mathcal{N}_{(v(L),j)} \cap (\Pi(L) \setminus \Pi(L')) \neq \emptyset\}$, i.e., the nodes where L' can be directly extended but not L or the nodes that remember at least one such node. This brings out the following set-based dominance criterion. Let $\mathcal{L}$ be a collection of labels $(L_i)_{i\in\mathcal{L}}$. It is said to *strongly dominate* a label $L' \notin \mathcal{L}$ if the two following conditions hold:

i. L_i weakly dominates L' for all $i \in \mathcal{L}$,
ii. for every *ng*-feasible extension of L' to a node w, there exists $i \in \mathcal{L}$ such that L_i can also be extended to w and $\Pi(L_i \oplus w) \subseteq \Pi(L' \oplus w)$.

Conditions (i.) and (ii.) combined assure that any path obtained by feasibly extending label L' would be dominated. Hence, L' can be discarded. Moreover, the authors also demonstrate that, whenever a weakly dominating label L_i is identified, one can immediately restrict feasible extensions of L', by discarding those that would be dominated by path $L_i \oplus w$ for all $w \in N \setminus \Pi(L_i)$ such that $L_i \oplus w$ is resource-feasible. As a result, not only full dominance reduces the number of labels, but also partial dominance contributes to avoiding certain extensions. On the other hand, their computational experiments showed that it may be computationally expensive to store the set of dominated extensions for each label. Consequently, as a compromise, instead of representing this set explicitly, they rely on an implicit representation which requires to only extend the memory $\Pi(L')$ of a dominated label L' along a dominated extension, thus, avoiding the extension of the label cost and resource components.

3 Selective arc-ng-SPPRC

In this section, we redefine strong dominance. For the sake of readability, we present the mechanism in two steps. First of all, we show that the pairwise comparison criterion (13) can be replaced by:

$$\Pi(L) \subseteq \Pi(pred(L')) \cup \{v(L')\}, \tag{14}$$

Proposition 1 below formalizes the correctness of this criterion. Its proof relies on the following lemma.

Lemma 1 *A labeling algorithm using conditions (10)–(12), (14) to establish pairwise dominance produces, for every label E representing an elementary partial path P, at least one non-dominated label L such that $v(L) = v(E)$, $\xi^r(L) \leq \xi^r(E)$, $\forall r \in R$, $\bar{c}(L) \leq \bar{c}(E)$, and $\Pi(L) \subseteq V(E)$, where $V(E)$ denotes the set of customer nodes visited by P.*

Proof. The proof works by induction on the size p of $P = \{v_1 = 0, \ldots, v_p\}$. If $p = 1$, then E is non-dominated and $L = E$. If $p \geq 2$, let us assume that E is obtained by extending another label E' over an arc $(v_p, v_{p+1}) \in A$, with $v_p = v(E')$ and $v_{p+1} \notin V(E')$. Using the induction hypothesis, the labeling algorithm generates a non-dominated label L' such that $v(L') = v(E')$, $\xi^r(L') \leq \xi^r(E')$, $\forall r \in R$, $c(L') \leq c(E')$, $\Pi(L') \subseteq V(E')$. Denote by L'' the label produced by extending L' along arc (v_p, v_{p+1}). It is easy to see that L'' is a feasible extension of L', and that $v(L'') = v(E)$, $c(L'') \leq c(E)$ and $\xi^r(L'') \leq \xi^r(E)$, $\forall r \in R$, hold. In addition, $\Pi(L'') \subseteq \Pi(L') \cup \{v_{p+1}\} \subseteq V(E') \cup \{v_{p+1}\} = V(E)$. If L'' is not dominated, then $L = L''$. Otherwise, if L'' is dominated by a non-dominated label L''', then L''' satisfies $v(L''') = v(E)$, $c(L''') \leq c(L'') \leq c(E)$, $\xi^r(L''') \leq \xi^r(L'') \leq \xi^r(E)$, $\forall r \in R$, and $\Pi(L''') \subseteq \Pi(pred(L'')) \cup \{v_{p+1}\} = \Pi(L') \cup \{v_{p+1}\} \subseteq V(E') \cup \{v_{p+1}\} = V(E)$. In this case, $L = L'''$. □

From this lemma, we deduce the following result.

Proposition 1 *If there exists an elementary path P from 0 to $n+1$ with a negative reduced cost, then a labeling algorithm based on the dominance rule (10)–(12) and (14) finds at least one (possibly non-elementary) path P' from 0 to $n+1$ with a negative reduced cost.*

Proof. Assume that there exists such a path P which is represented by a label E. If E is not dominated, then $P' = P$. Otherwise, according to Lemma 1, there exists a non-dominated label L such that $v(L) = v(E)$ and $\bar{c}(L) \leq \bar{c}(E)$. This label represents path P'. □

To illustrate the strength of this new dominance rule, let us consider the following example. Let L_1 and L_2 be two labels such that $v(L_1) = v(L_2) = 1$, $c(L_1) \leq c(L_2)$, $\xi^r(L_1) \leq \xi^r(L_2)$, $\forall r \in R$, and let us assume that $\Pi(L_1) = \{1, 2, 3\}$ and $\Pi(L_2) = \{1, 2, 4\}$. Although L_1 weakly dominates L_2, the latter cannot be discarded according to (10)–(13) because $\Pi(L_1) \not\subseteq \Pi(L_2)$ (Figure 1). Now, let us assume that label L_2 was extended from label $pred(L_2)$ with $v(pred(L_2)) = 2$, over the arc $a = (2, 1)$, and such that $\mathcal{N}_a = \{1, 2, 4\}$ and $\Pi(pred(L_2)) = \{2, 3, 4\}$. We now have that $\Pi(L_1) = \{1, 2, 3\} \subseteq \{1, 2, 3, 4\} = \Pi(pred(L_2)) \cup \{1\}$, and label L_2 can be safely discarded according to condition (14) (Figure 2).

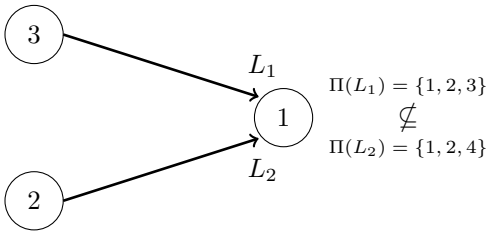


Figure 1: Standard dominance

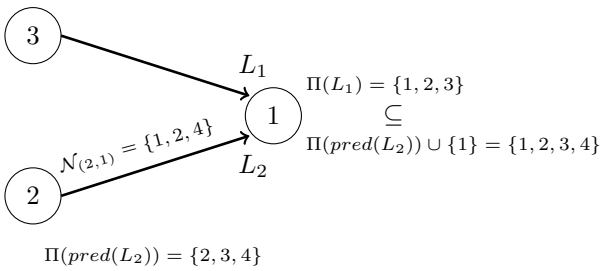


Figure 2: Selective dominance

Observe that, in this example, it might be feasible to extend label L_2 to node 3 and that this extension might subsequently yield non-dominated ng-paths between 0 and $n+1$, possibly an optimal one. Nevertheless, when label L_2 is dominated according to conditions (10)–(12) and (14), this extension is not possible and the algorithm remains valid because the discarded ng-paths are not elementary (they visit node 3 more than once). The pricing is, thus, selective.

The second step in the presentation of our selective set-based dominance rule is stated in a corollary of the pairwise dominance condition, and extends the strong dominance criterion given by conditions (i.)–(ii.).

Corollary 1 *For a given label $L^\star$, an extension to a node $w \in N^+ \setminus \Pi(L^\star)$ can be safely omitted if there exists a label L such that all the following conditions hold:*

$$v(L) = v(L^\star \oplus w) = w, \tag{15}$$
$$\bar{c}(L) \leq \bar{c}(L^\star \oplus w), \tag{16}$$
$$\xi^r(L) \leq \xi^r(L^\star \oplus w), \qquad \forall r \in R \tag{17}$$
$$\Pi(L) \subseteq \Pi(L^\star) \cup \{w\}. \tag{18}$$

In fact, one can observe that (15)–(18) correspond to (10)–(12) and (14) when setting $L' = L^\star \oplus w$, showing that, if L exists and $L^\star$ was extended along arc $(v(L^\star), w)$, the resulting label $L' = L^\star \oplus w$ would be dominated by L.

Similarly to the idea proposed by Bulhões et al. (2018), the dominance rule given by conditions (15)–(18) intends to consider some available information to identify dominated label extensions. Yet, the dominance rule described in Corollary 1 is stronger than the one proposed by Bulhões

et al. (2018) because it uses the selection mechanism given by condition (18), which is stronger than condition (13) (with $L' = L^\star \oplus w$). Another difference between the two algorithms is that our set-based dominance rule does not assume L and $L^\star \oplus w$ to be extended from the same node (see Figure 3), unlike that of Bulhões et al. (2018) (see Figure 4). In these cases, label $L^\star \oplus w$ can be identified as dominated by L without extending the cost and resource components of $L^\star$ in the algorithm of Bulhões et al. (2018) and without extending the memory of $L^\star$ in our selective algorithm. Therefore, if the average memory size is much larger than the number of cost and resource components, the selective dominance rule proposed in our algorithm can yield an additional speedup.

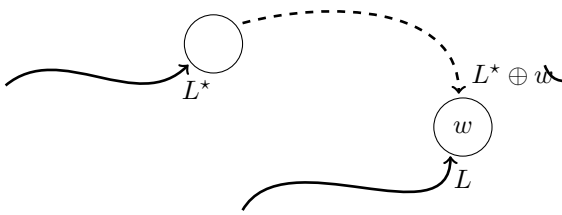


Figure 3: Selective set-based comparison

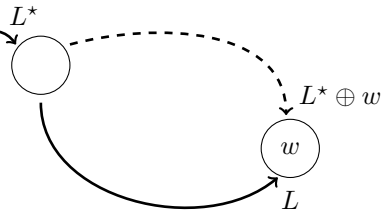


Figure 4: Bulhões et al.'s mechanism

One way to reinforce dominance relations (14) and (18) is by adding to the set Π of the dominating label, the unreachable nodes, i.e., the nodes to which extending the path would not be resource-feasible (equivalently, it is possible to do the same for relation (13)). Assuming that the resource consumptions γ^r_{ij}, $(i,j) \in A$ satisfy the triangle inequality for each resource $r \in R$, the set of unreachable nodes of a label L is defined as $\mathcal{U}(L) = \{w \in N^+ |\ \exists r \in R \text{ such that } \xi^r(L) + \gamma^r_{v(L),w} > l^r_w\}$. Given that L and $L^\star \oplus w$ cannot be feasibly extended to any node in $\mathcal{U}(L)$ and $\mathcal{U}(L^\star \oplus w)$, respectively, these sets can be included in the right-hand side of their respective dominance conditions (14) and (18), which can be rewritten as:

$$\Pi(L) \subseteq \Pi(pred(L')) \cup \{v(L')\} \cup \mathcal{U}(L') \tag{19}$$

$$\Pi(L) \subseteq \Pi(L^\star) \cup \{w\} \cup \mathcal{U}(L^\star \oplus w). \tag{20}$$

To avoid computing the unreachable set of label $L^* \oplus w$ before testing dominance condition (20), the following slightly weaker condition can be used:

$$\Pi(L) \subseteq \Pi(L^\star) \cup \mathcal{U}(L^\star) \cup \{w\}. \tag{21}$$

4 Computational experiments

To assess the effectiveness of the proposed selective dominance rules, we performed a series of computational experiments on well-known benchmark instances of the VRPTW which can be defined as follows. Consider the network $G = (N, A)$ described in Section 2.1, where N is the set of nodes, A is the set of arcs, and N^+ is the set of customer nodes. Each customer $i \in N^+$ is associated with a demand q_i, a service time s_i, and a time window $[e_i, l_i]$ within which the service must start. Moreover, we set $q_0 = s_0 = e_0 = q_{n+1} = s_{n+1} = e_{n+1} = 0$ and $l_0 = l_{n+1} = H$, where H is the planning horizon duration. Each arc $(i,j) \in A$ is associated with a routing cost c_{ij} and a traveling time t_{ij}. An unlimited fleet of homogeneous vehicles with capacity Q is available at a single depot. The VRPTW consists of finding feasible routes visiting all customers $i \in N^+$ exactly once such that the total routing cost is minimized. A route is deemed feasible if: it is elementary; all visited customers are served within their time windows; the total demand of these customers does not exceed the vehicle capacity Q. Furthermore, the cost of a route is computed as the sum of the costs of the arcs traversed by the route.

For the VRPTW, the pricing subproblem involves a set R of two resources, namely, time and load. At a node $i \in N$, the resource windows $[e^r_i, l^r_i]$, $r \in R$, are defined by their time window $[e_i, l_i]$ and by

the load window $[0, Q]$. Along an arc $(i, j) \in A$, the resource consumptions γ_{ij}^r, $r \in R$, are given by $t_{ij} + s_i$ for the time resource and by q_j for the load resource.

In this section, we provide some details on how we conducted our computational experiments and report the results obtained. The performance of the selective *arc-ng*-SPPRC pricing is compared against the default *arc-ng*-SPPRC pricing. Both of these settings are implemented over the same column-and-cut generation framework, which is described in Section 4.1. In Section 4.2, we provide details on the experiments design. Finally, in Section 4.3, we report and discuss the computational results obtained.

4.1 Column-and-cut-generation framework

In our experiments, we consider a column-and-cut-generation framework very similar to the one developed by Pecin et al. (2017a) for the VRPTW. The algorithm includes several refinements that enhance its performance. The labeling algorithm used to solve the pricing subproblem relies on a bidirectional DSSR procedure (without completion bounds), having the time as the critical resource (Righini and Salani, 2006). The maximum size of *ng*-sets in the algorithm vary according to the instance being solved as discussed below. Furthermore, three fast heuristic labeling algorithms are always executed before any execution of the exact pricing algorithm. The first heuristic keeps only the least-cost label associated with a given time arriving at each customer node. The second and third heuristics rely on a reduced version of the network G, where around 7 and 12 (for the second and third, respectively) arcs entering and leaving each customer node are kept (see Desaulniers et al., 2008, for details on how these arcs are selected). At each column generation iteration, these three heuristic labeling algorithms are called in sequence until one of them finds negative reduced cost columns. If they all fail, the cut separation routines (see below) are invoked. When cuts are found, they are added to the RMP and column generation is re-started. Otherwise, the exact pricing algorithm is called. If it finds negative reduced cost columns, the process starts over again. Otherwise, the algorithm stops.

Because we wanted to make the implementation of our column generation algorithm as simple as possible, we decided not to consider the dynamic mechanism described by Bulhões et al. to increase and reduce the size of the *ng*-sets when needed. We judged that this procedure would make the implementation of the algorithm very cumbersome. Hence, we decided to rely on Martinelli et al. (2014)'s framework to handle the size of the *ng*-sets. In our implementation, *ng*-sets are initially empty and are updated throughout the solution process. Two strategies were considered to update the arc *ng*-sets: i) `only-cycle`, where only *ng*-sets associated with arcs appearing in an infeasible cycle are increased; and ii) `all-arcs`, where all the arcs induced by the customer nodes in an infeasible cycle have their *ng*-sets increased. The strategy `only-cycle` is the same considered by Bulhões et al. (2018). As pointed out by the authors, and also as we have observed in our preliminary experiments, the `only-cycle` strategy produces a higher number of DSSR iterations. However, the computational complexity of each such iteration remains lower (this is more notorious towards the end).

The baseline algorithm also includes the separation of several cuts, namely: rounded capacity cuts (RCCs, Laporte et al., 1985), limited-memory rank-1 cuts (Pecin et al., 2017a,b), and elementary cuts (Pecin et al., 2017a). Except for the RCCs, all the other cuts are non-robust, i.e., their dual variables cannot be considered directly in the modified arc costs when solving the pricing subproblem. The explicit handling of these dual variables increases the difficulty of solving it. Consequently, the solution process is divided in two phases, called the *robust* and *non-robust* phases. In the robust phase, the search for violated cuts is restricted to the RCCs. This phase ends with a call to the exact pricing algorithm that yielded no new columns and the application of variable arc fixing (Irnich et al., 2010). The non-robust phase then starts by searching for non-robust violated cuts. If some are found, the solution process is re-started. This phase also ends with a final unsuccessful call to the exact pricing algorithm, ensuring that the value of the current RMP optimal solution provides a valid lower bound.

4.2 Experiments design

Our tests were performed on two datasets: 1) the 14 hardest 100-customer VRPTW instances from Solomon (1987); and 2) the 200-customer instances of Gehring and Homberger (2001) that are reported to be solved in less than 5 hours by Sadykov et al. (2017). These instances are called hereafter the S and the GH instances. Both datasets contain instances whose node locations are: chosen at random, clustered, and mixed (random and clustered). This is denoted in the S and GH instances, respectively, as: C, R, and RC; and C1, C2, R1, R2, RC1 and RC2. For the GH instances, specifically, instances in sets C1, R1, and RC1 have tight time windows and vehicle capacity, and typically admit optimal solutions with short vehicle routes. On the other hand, sets C2, R2, and RC2 have large time windows and loose vehicle capacity and, typically, admit optimal solutions with fewer but longer vehicle routes.

Similarly to what is done in Pecin et al. (2017a), for all S instances except R208, and all C1, RC1, and R1 GH instances, the cardinality of *ng*-sets $\mathcal{N}_i$, $i \in N^+$ are limited to 10. For the other instances, where allowing more cycles can be very harmful to the lower bound, we consider $|\mathcal{N}_i| = 20$. These node *ng*-sets are used to generate the arc *ng*-sets $\mathcal{N}_a$, $a = (u, v) \in A$, by using the formula $\mathcal{N}_a \leftarrow \mathcal{N}_u \cap \mathcal{N}_v$. Furthermore, for all instances except the C1, RC1, and R1 GH instances, we observed that the capacity constraint is not binding. Therefore, to alleviate the solution of the pricing subproblem, we did not consider this constraint, which is later enforced using RCCs whenever needed.

Given that our goal is not to solve to optimality the considered instances but rather to illustrate the impact of applying the selective *arc-ng*-SPPRC pricing, we focus only on the root node results. The algorithm described previously was coded in C++, and IBM ILOG CPLEX Optimizer 12.8 was used as the LP solver. The experiments were carried out on an Intel Xeon ES-2637 3.5GHz with 128GB RAM, running Linux Oracle Server 7.6. A time limit of 24h was set to solve all instances.

In our experiments, we compare the performance of the default *arc-ng*-SPPRC pricing against three versions of the selective *arc-ng*-SPPRC pricing, which differ by the dominance rule employed. All settings considered in our study are detailed below:

1. `Default`: The baseline algorithm that considers relations (10)–(13) as dominance rule. Relation (13) is reinforced by including the set $\mathcal{U}(L')$ of unreachable nodes in its right-hand side (which becomes $\Pi(L') \cup \mathcal{U}(L')$).
2. `SetBased`: Considers relations (10)–(12) and the set-based relation (21) as dominance rule.
3. `Pairwise`: Considers relations (10)–(12) and the pairwise relation (19) as dominance rule.
4. `SetPair`: Considers relations (10)–(12), both pairwise (19) and set-based (21) relations as dominance rule.

Additionally, each pricing setting is executed twice, each time considering a different DSSR strategy, i.e., `only-cycle` or `all-arcs`. Hence, in total, we test eight distinct configurations.

Notice that the `Default` setting applies a pairwise comparison on sets Π. We do not consider in our analysis the set-based dominance rule of the default *arc-ng*-SPPRC pricing developed by Bulhões et al. (2018) because it would have required major changes to our implementation. Because our set-based dominance rule encompasses that of Bulhões et al., it is expected to yield better results.

For each instance, we retrieve the following information: lower bound (**lb**), number of column generation iterations performed by each algorithm (**#iters**), CPU time in seconds (**T(s)**), average number of DSSR iterations per column generation iteration performed by each algorithm (**#DSSR**), and average number of labels generated per iteration (**#labels**). This information is collected before (robust phase) and after (non-robust phase) the addition of non-robust cuts. Yet, because the main indicators for evaluating the efficiency of the selective algorithm are computational time and average number of labels generated per iteration, in Section 4.3, we focus our analysis on these two measures. Detailed results for all indicators are presented in Appendix A.

4.3 Computational results

In this section, we discuss the results obtained during our experiments. For the sake of conciseness, we only report summary results (see Appendix A for complete results). In Tables 1–4, we provide the average and the median for **T(s)** and **#labels** per instance class, where an instance class is preceded by S or GH to identify if they correspond to S or GH instances. In each table, we compare the performance of settings `Default`, `SetBased`, `Pairwise`, and `SetPair`. We analyze separately the performance of the algorithms with respect to the DSSR strategy employed (`only-cycle` and `all-arcs`). We provide distinct tables for the algorithms before and after separating non-robust cuts. Finally, line **#Best** shows the number of times that each setting provides the best results for each indicator and line **#OPD** displays the number of times that the corresponding setting outperforms the `Default` setting. Notice that, in the case where both algorithms produce equal results for a given instance, this instance is counted for both algorithms. As a consequence, the sum of the **#Best** values can exceed the number of instances.

In Tables 1 and 2, we present the summary results produced by the algorithms before the separation of non-robust cuts. These results show a clear advantage of the selective settings over the default algorithm in terms of running time and the number of labels produced. This statement is supported by the **#OPD** values. Settings `SetPair` and `SetBased` are the ones yielding the best gains in terms of running time for `only-cycle` and `all-arcs`, respectively. Regarding the number of labels, for both DSSR strategies, the setting `Pairwise` is the most effective at reducing the number of non-dominated labels kept for both DSSR strategies. Moreover, as one can see from detailed results in Appendix A, during the robust phase of the algorithm, the selective algorithms allow better bounds to be achieved. This observation is more notable when solving GH instances using the `only-cycle` DSSR strategy. The selective strategies also tend to require less DSSR iterations.

When comparing the performance of the selective algorithms employing different DSSR strategies, the results confirm what has already been hinted by Bulhões et al. (2018), i.e., the algorithms employing `only-cycle` produce a larger number of DSSR iterations. Also, they tend to keep more non-dominated labels. On the other hand, these algorithms seem to require *ng*-sets of smaller sizes.

Table 1: Aggregated results using the `only-cycle` DSSR strategy before adding non-robust cuts

Classes		T(s)				#labels			
		`Default`	`SetBased`	`Pairwise`	`SetPair`	`Default`	`SetBased`	`Pairwise`	`SetPair`
S-C	Average	14,830	8,723	10,232	9,247	44,428	38,831	39,232	43,842
	Median	14,830	8,723	10,232	9,247	44,428	38,831	39,232	43,842
S-R	Average	1,581	1,254	1,168	1,230	76,292	56,785	57,885	61,668
	Median	641	551	531	515	38,749	32,613	33,084	37,482
S-RC	Average	7,015	4,733	4,517	4,894	502,004	343,281	319,508	328,526
	Median	3,071	2,473	2,425	2,476	632,136	463,053	425,550	443,300
GH-C1	Average	604	525	541	494	319,713	258,680	246,711	261,399
	Median	566	503	492	479	296,760	236,845	237,015	245,659
GH-C2	Average	6,136	5,386	4,763	4,826	21,571	20,257	17,651	18,199
	Median	6,342	6,322	5,605	5,688	15,499	13,644	14,472	12,457
GH-R1	Average	175	142	175	147	209,212	187,153	196,498	183,037
	Median	153	134	157	135	190,129	170,515	191,588	179,043
GH-R2	Average	4,789	4,342	4,399	4,250	13,845	11,459	11,717	12,536
	Median	4,217	3,836	3,639	3,938	11,431	9,902	10,221	9,247
GH-RC1	Average	201	167	192	174	438,176	369,259	390,951	379,880
	Median	144	128	137	129	253,216	218,390	228,745	214,626
GH-RC2	Average	6,689	5,757	5,887	5,816	46,391	36,427	40,806	41,672
	Median	6,621	5,665	5,793	5,739	45,338	36,552	36,654	36,959
#Best		2/45	11/45	15/45	17/45	3/45	19/45	20/45	9/45
#OPD		–	42/45	38/45	42/45	–	38/45	42/45	40/45

Table 2: Aggregated results using the `all-arcs` DSSR strategy before adding non-robust cuts

Classes		T(s)				#labels			
		`Default`	`SetBased`	`Pairwise`	`SetPair`	`Default`	`SetBased`	`Pairwise`	`SetPair`
S-C	Average	5,143	11,198	4,897	8,214	36,339	29,880	30,321	29,100
	Median	5,143	11,198	4,897	8,214	36,339	29,880	30,321	29,100
S-R	Average	815	749	747	773	46,854	41,961	43,092	42,001
	Median	390	365	348	375	24,038	25,711	21,949	24,779
S-RC	Average	1,812	1,526	1,496	1,452	163,293	135,035	136,881	121,865
	Median	904	786	826	814	218,110	163,647	178,990	163,170
GH-C1	Average	378	323	326	329	170,063	164,153	148,659	160,393
	Median	400	317	342	329	147,946	127,657	115,431	126,360
GH-C2	Average	3,156	3,068	3,030	2,891	13,658	14,850	12,871	13,136
	Median	3,564	3,412	3,405	3,144	11,771	13,276	12,090	11,894
GH-R1	Average	139	119	136	121	133,901	128,970	118,771	122,280
	Median	132	123	128	121	117,367	124,032	122,804	106,912
GH-R2	Average	3,738	3,436	3,508	3,467	9,810	10,231	8,693	8,666
	Median	3,616	3,471	3,339	3,352	7,647	8,488	8,044	7,994
GH-RC1	Average	131	112	128	116	225,358	217,014	213,540	222,278
	Median	116	102	117	108	170,003	164,044	154,841	171,926
GH-RC2	Average	3,959	3,777	3,580	3,653	25,770	25,056	23,105	25,354
	Median	3,411	3,372	3,314	3,289	25,309	23,846	23,324	22,316
#Best		4/45	16/45	13/45	12/45	11/45	11/45	18/45	12/45
#OPD		–	36/45	34/45	38/45	–	23/45	37/45	32/45

Tables 3 and 4 show the results obtained during the non-robust phase of the algorithms. In the presence of non-robust cuts, the selective algorithms can become more time-consuming. Nevertheless, they are still faster than the default algorithms for the majority of the instances. The better performance of the selective algorithms is confirmed when we analyze the performance of the algorithms pairwise in Appendix A. The somewhat high average times associated with some selective settings are due to the poor performance of the selective algorithms when solving particular instances. However, when analyzing a less biased indicator such as the median, the performance of the algorithms tend to be more similar. Sometimes, the selective algorithms can even yield better results, as it is the case for the setting `SetBased` when solving instances from classes R2 and RC2 in Table 3. Regarding the number of labels, the selective algorithms remain more effective in reducing the number of generated labels in each algorithm.

Finally, we present time profiles (Dolan and More, 2002) to compare the performance of the algorithms, which are generated as follows. Let $\mathcal{A}$ be a set of algorithms, namely, $\mathcal{A} = \{$`Default`, `SetBased`, `Pairwise`, `SetPair`$\}$ in our case. Let I be a set of instances, namely, the 45 instances considered in our experiments. For each algorithm $A \in \mathcal{A}$ and each instance $i \in I$, denote by T_A^i the time spent by A to solve instance i and let $T_{\text{Best}}^i = \min_{A \in \mathcal{A}}\{T_A^i\}$ be the best time achieved by an algorithm in $\mathcal{A}$ to solve this instance. For an algorithm $A \in \mathcal{A}$, a time profile is a function $\rho_A(\tau)$ of a ratio $\tau \geq 1$ that is equal to the percentage of instances such that $\frac{T_A^i}{T_{\text{Best}}^i} \leq \tau$, i.e.,

$$\rho_A(\tau) = 100 \, \frac{|\{i \in I \mid \frac{T_A^i}{T_{\text{Best}}^i} \leq \tau\}|}{|I|}. \tag{22}$$

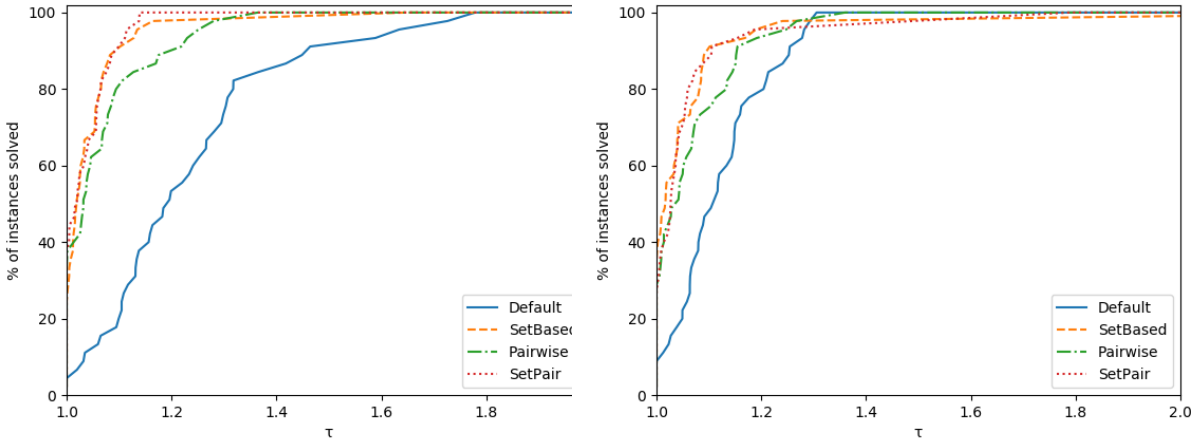
In particular, $\rho_A(1)$ represents the percentage of instances for which algorithm A was the fastest. Figures 5 and 6 display the time profiles of the four algorithms for the robust and the non-robust phases, respectively. From these profiles, one can see that the selective algorithms yield considerable speed-ups when compared to the `Default` algorithm. The acceleration is more notable when comparing algorithms employing the `only-cycle` DSSR strategy. Additionally, we remark that setting `SetPair` is the fastest during the robust phase, but becomes time consuming once the separation of non-robust cuts starts.

Table 3: Aggregated results using the `only-cycle` DSSR strategy after adding non-robust cuts

Classes		T(s)				#labels			
		Default	SetBased	Pairwise	SetPair	Default	SetBased	Pairwise	SetPair
S-C	Average	14,831	8,723	10,232	9,248	44,428	38,831	39,232	43,842
	Median	14,831	8,723	10,232	9,248	44,428	38,831	39,232	43,842
S-R	Average	24,607	28,713	23,606	25,376	161,918	161,095	151,546	134,202
	Median	3,034	2,219	2,682	2,374	30,501	29,865	30,571	27,684
S-RC	Average	8,342	6,231	5,926	6,442	265,520	188,351	175,184	182,177
	Median	4,626	4,107	3,940	4,172	347,683	249,233	233,377	244,941
GH-C1	Average	31,590	10,321	14,917	60,826	211,779	186,186	173,657	181,226
	Median	1,090	967	1,006	911	209,024	188,646	168,723	179,803
GH-C2	Average	8,575	7,606	7,172	7,114	14,852	13,912	12,893	12,495
	Median	7,449	7,748	6,624	7,322	8,996	9,145	7,490	7,256
GH-R1	Average	6,769	4,550	5,270	8,425	162,425	155,355	154,977	155,259
	Median	2,041	2,014	2,377	2,497	86,138	81,608	79,223	82,495
GH-R2	Average	7,356	7,563	7,343	6,994	5,939	5,014	5,208	5,153
	Median	5,808	5,294	5,232	5,487	5,578	4,926	4,638	4,622
GH-RC1	Average	39,091	37,398	43,179	50,820	292,527	297,061	302,914	285,138
	Median	15,333	19,823	17,230	33,158	114,444	113,465	115,899	113,536
RC2	Average	19,399	20,420	19,846	23,343	15,501	13,658	13,924	12,968
	Median	13,046	11,836	9,761	11,356	14,820	13,225	12,231	11,468
#Best		6/45	18/45	14/45	7/45	4/45	12/45	14/45	18/45
#OPD		–	33/45	32/45	23/45	–	34/45	31/45	39/45

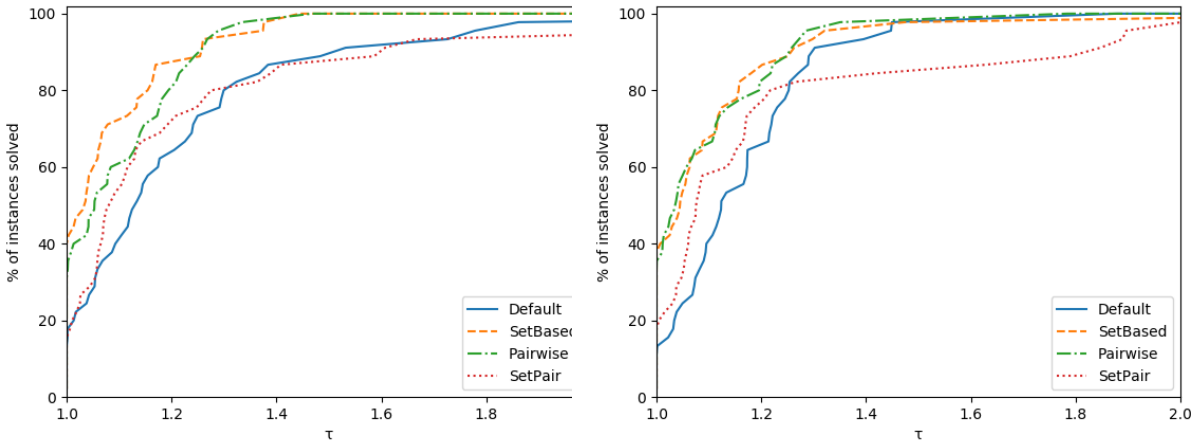
Table 4: Aggregated results using the `all-arcs` DSSR strategy after adding non-robust cuts

Classes		T(s)				#labels			
		Default	SetBased	Pairwise	SetPair	Default	SetBased	Pairwise	SetPair
S-C	Average	5,144	11,199	4,897	8,215	36,339	29,880	30,321	29,100
	Median	5,144	11,199	4,897	8,215	36,339	29,880	30,321	29,100
S-R	Average	24,899	22,678	23,555	24,346	116,572	113,544	120,109	103,872
	Median	1,784	1,631	2,035	1,739	21,936	27,285	25,382	28,997
S-RC	Average	2,847	2,554	2,515	2,613	93,557	78,568	77,082	73,720
	Median	1,961	1,746	1,766	1,852	128,015	95,283	96,170	90,537
GH-C1	Average	8,893	11,382	11,834	24,217	23,675	23,930	23,447	23,065
	Median	670	573	576	572	25,871	26,420	24,454	25,218
GH-C2	Average	5,730	4,790	4,498	4,643	4,444	4,497	4,072	4,135
	Median	4,633	4,390	4,494	4,234	4,033	4,423	3,986	3,850
GH-R1	Average	4,447	3,942	4,402	6,188	51,827	52,685	51,153	52,042
	Median	2,140	2,074	1,940	1,923	29,849	31,203	29,589	29,301
GH-R2	Average	6,032	5,883	5,855	6,247	2,782	2,903	2,767	2,742
	Median	4,994	4,842	4,653	4,747	2,213	2,287	2,287	2,233
GH-RC1	Average	23,394	18,850	27,210	27,301	55,442	58,633	58,564	55,439
	Median	16,704	14,473	13,318	25,329	39,128	40,898	38,055	40,736
GH-RC2	Average	12,091	11,194	10,633	10,593	7,473	8,120	6,662	7,342
	Median	7,410	7,946	7,370	7,212	7,422	7,932	6,863	7,360
#Best		5/45	17/45	15/45	8/45	9/45	15/45	11/45	13/45
#OPD		–	37/45	30/45	29/45	–	25/45	33/45	30/45



(a) Using `only-cycle` DSSR **(b) Using `all-arcs` DSSR**

Figure 5: Time profiles for all instances before adding non-robust cuts



(a) Using `only-cycle` DSSR **(b) Using `all-arcs` DSSR**

Figure 6: Time profiles for all instances after adding non-robust cuts

5 Concluding remarks

In this paper, we have presented two new selective dominance rules for the *arc-ng*-SPPRC, namely, one that applies pairwise label comparison and one that extends the set-based rule of Bulhões et al. (2018). The latter rule is stronger than the one proposed by Bulhões et al. as it considers the selective mechanism expressed by condition (18), which increases the chances that a label L dominates a label $L' = L^\star \oplus w$. Furthermore, an additional speedup may be observed if the average memory size is much larger than the number of cost and resource components in a label. Our computational experiments on VRPTW benchmark instances showed that, in general, the new mechanism allows a reduction in terms of the number of treated labels and, consequently, of the computational time. Note also that one of the main interests for using this mechanism is that it does not rely on complex data structures like the ones described by Bulhões et al. (2018, Section 4.2) and Desaulniers et al. (2019, Section 3.2) to work satisfactorily.

An extension of our idea would be to investigate the impact of generalizing relation (14) to consider other predecessors further down in the path. Theoretically, this generalization would strengthen even more the dominance rule. However, it seems that an elaborated data structure would be required to address the overhead incurred by the new relations.

Appendix

A Detailed results

In this section, we report detailed results for all instances solved by settings `Default`, `SetBased`, `Pairwise`, and `SetPair`. We provide pairwise comparisons between the `Default` algorithm and each of the selective settings. Sections A.1 and A.2 present results for S and GH instances, respectively. For each instance, we report: the lower bound (**lb**) achieved, the number of column generation iterations performed by each algorithm (**#iters**), the CPU time in seconds (**T(s)**), the average number of DSSR iterations per column generation iteration performed by each algorithm (**#DSSR**), and the average number of labels generated per iteration (**#labels**). When comparing the performance of two algorithms, we highlight in boldface the best results for each indicator. Line **#Best** shows the total number of instances for which each algorithm was capable of achieving the best values. Notice that, in the case where both algorithms produce equal results for a given instance, this instance is counted for both algorithms.

A.1 Solomon's instances

Table 5: Detailed results for comparing `Default` vs `SetBased` settings using `only-cycle` DSSR before adding non-robust cuts

Instance	Default					SetBased				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	4,454	800	**2.1**	13,516	5,887.0	**2,841**	**517**	2.3	**12,527**
C204	5,881.0	25,207	1,239	**2.4**	75,340	5,881.0	**14,604**	**780**	2.7	**65,135**
R202	10,164.8	290	**130**	4.2	10,223	10,164.8	**268**	136	**3.6**	**8,169**
R203	8,597.0	785	169	5.1	31,121	8,597.0	**615**	**131**	5.1	**27,872**
R204	7,201.6	2,434	175	6.9	121,553	7,201.6	**1,704**	**171**	**5.3**	**78,889**
R206	8,600.1	582	**103**	7.6	38,749	8,600.1	**551**	113	**6.0**	**30,732**
R207	7,805.0	1,152	152	**5.9**	54,037	7,805.0	**881**	**117**	6.1	**52,027**
R208	6,910.9	7,539	**234**	8.7	291,927	6,910.9	**6,059**	250	**7.0**	**204,117**
R209	8,371.8	340	**81**	7.2	29,284	8,371.8	**303**	86	**5.8**	**22,617**
R210	8,813.2	463	93	8.5	34,247	8,813.2	**398**	**82**	**8.0**	**32,613**
R211	7,311.0	641	115	7.8	75,489	7,311.0	**503**	115	**6.2**	**54,027**
RC204	7,764.9	17,166	269	11.9	789,806	7,764.9	**11,096**	**256**	**10.8**	**463,053**
RC207	9,415.8	808	112	**14.3**	84,070	9,415.8	**630**	**94**	14.8	**73,374**
RC208	7,636.4	3,071	148	22.0	632,136	7,636.4	**2,472**	**139**	**21.9**	**493,415**
#Best	14/14	0/14	5/14	5/14	0/14	14/14	14/14	10/14	10/14	14/14

Table 6: Detailed results for comparing `Default` vs `SetBased` settings using `only-cycle` DSSR after adding non-robust cuts

Instance	Default					SetBased				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	4,454	800	**2.1**	13,516	5,887.0	**2,842**	**517**	2.3	**12,527**
C204	5,881.0	25,207	1,239	**2.4**	75,340	5,881.0	**14,605**	**780**	2.7	**65,135**
R202	**10,296.0**	796	736	1.7	**3,788**	10,288.2	**615**	**647**	1.6	3,826
R203	8,708.0	1,239	725	2.1	11,040	8,708.0	**998**	**554**	2.1	**10,953**
R204	7,313.0	5,352	**638**	2.9	73,417	7,313.0	**4,442**	645	**2.5**	**58,024**
R206	8,759.0	1,435	519	2.6	18,251	8,759.0	**1,362**	**515**	**2.4**	**17,473**
R207	7,940.0	3,034	654	2.4	31,611	7,940.0	**2,219**	**523**	2.5	**29,865**
R208	6,994.1	**196,397**	**594**	5.3	1,125,610	**7,000.3**	234,324	710	**4.3**	**1,124,685**
R209	8,548.0	**2,071**	**580**	2.4	24,821	8,548.0	2,095	591	**2.3**	**24,635**
R210	9,005.0	4,354	674	2.3	**30,501**	9,005.0	**3,812**	**635**	2.3	30,887
R211	7,467.0	**6,785**	**357**	4.2	**138,224**	7,467.0	8,549	380	**3.8**	149,502
RC204	7,835.0	18,715	546	6.5	413,895	7,835.0	**12,956**	**535**	**5.9**	**249,233**
RC207	9,629.0	1,683	459	4.6	34,981	9,629.0	**1,628**	**457**	**4.3**	**31,885**
RC208	7,761.0	4,626	373	10.0	347,683	7,761.0	**4,107**	**370**	**9.6**	**283,936**
#Best	13/14	3/14	4/14	6/14	3/14	13/14	11/14	10/14	12/14	11/14

Table 7: Detailed results for comparing `Default` vs `Pairwise` settings using `only-cycle` DSSR before adding non-robust cuts

Instance	Default					Pairwise				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	4,454	800	**2.1**	**13,516**	5,887.0	**2,504**	**360**	3.0	15,993
C204	5,881.0	25,207	1,239	2.4	75,340	5,881.0	**17,959**	**981**	2.4	**62,472**
R202	10,164.8	290	130	4.2	10,223	10,164.8	**271**	130	**3.9**	**7,618**
R203	8,597.0	785	169	5.1	31,121	**8,598.1**	**606**	**168**	**3.9**	**21,073**
R204	7,201.6	2,434	175	6.9	121,553	7,201.6	**1,791**	**174**	**5.4**	**82,007**
R206	8,600.1	582	**103**	7.6	38,749	8,600.1	**531**	109	**6.3**	**28,842**
R207	7,805.0	1,152	152	**5.9**	54,037	7,805.0	**907**	**117**	6.3	**53,843**
R208	6,910.9	7,539	234	8.7	291,927	6,910.9	**5,204**	**213**	**8.0**	**206,766**
R209	8,371.8	340	81	7.2	29,284	8,371.8	**300**	**73**	**6.8**	**25,270**
R210	8,813.2	463	93	**8.5**	34,247	**8,813.4**	**375**	**73**	8.7	**33,084**
R211	7,311.0	641	115	7.8	75,489	**7,311.1**	**529**	**105**	**7.1**	**62,460**
RC204	7,764.9	17,166	**269**	11.9	789,806	7,764.9	**10,507**	279	**10.1**	**425,550**
RC207	9,415.8	808	112	14.3	84,070	9,415.8	**620**	**104**	**13.0**	**62,470**
RC208	7,636.4	3,071	148	22.0	632,136	7,636.4	**2,425**	**141**	**21.1**	**470,504**
#Best	11/14	0/14	3/14	4/14	1/14	14/14	14/14	12/14	11/14	13/14

Table 8: Detailed results for comparing `Default` vs `Pairwise` settings using `only-cycle` DSSR after adding non-robust cuts

Instance	Default					Pairwise				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	4,454	800	**2.1**	**13,516**	5,887.0	**2,505**	**360**	3.0	15,993
C204	5,881.0	25,207	1,239	2.4	75,340	5,881.0	**17,960**	**981**	2.4	**62,472**
R202	**10,296.0**	796	736	1.7	3,788	10,292.2	**641**	**632**	1.7	**3,548**
R203	8,708.0	1,239	725	2.1	11,040	8,708.0	**1,011**	**652**	**1.8**	**9,182**
R204	7,313.0	5,352	**638**	2.9	73,417	7,313.0	**4,789**	661	**2.6**	**58,075**
R206	8,759.0	**1,435**	519	2.6	**18,251**	8,759.0	1,441	**503**	**2.5**	18,458
R207	7,940.0	3,034	654	2.4	31,611	7,940.0	**2,682**	**614**	2.3	**30,571**
R208	6,994.1	196,397	**594**	5.3	1,125,610	**6,998.3**	**186,267**	636	**4.7**	**1,027,970**
R209	8,548.0	2,071	**580**	2.4	24,821	8,548.0	**2,067**	584	**2.3**	**22,306**
R210	9,005.0	**4,354**	674	2.3	**30,501**	9,005.0	4,504	**612**	2.3	32,925
R211	7,467.0	**6,785**	357	4.2	**138,224**	7,467.0	9,054	**355**	4.2	160,883
RC204	7,835.0	18,715	**546**	6.5	413,895	7,835.0	**12,217**	570	**5.6**	**233,377**
RC207	9,629.0	1,683	459	4.6	34,981	9,629.0	**1,622**	**444**	**4.3**	**29,640**
RC208	7,761.0	4,626	**373**	10.0	347,683	7,761.0	**3,940**	374	**9.3**	**262,534**
#Best	13/14	3/14	5/14	6/14	4/14	13/14	11/14	9/14	13/14	10/14

Table 9: Detailed results for comparing `Default` vs `SetPair` settings using `only-cycle` DSSR before adding non-robust cuts

Instance	Default					SetPair				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	4,454	800	2.1	13,516	5,887.0	**2,782**	**529**	**2.0**	**12,556**
C204	5,881.0	25,207	1,239	**2.4**	75,340	5,881.0	**15,712**	**722**	2.8	**75,129**
R202	10,164.8	290	130	4.2	10,223	10,164.8	**264**	**120**	4.2	**8,734**
R203	8,597.0	785	169	5.1	31,121	**8,597.9**	**631**	**151**	**4.4**	**24,619**
R204	7,201.6	2,434	175	6.9	121,553	7,201.6	**1,662**	**157**	**5.8**	**85,759**
R206	8,600.1	582	103	7.6	38,749	8,600.1	**515**	**91**	**7.2**	**37,482**
R207	7,805.0	1,152	152	**5.9**	54,037	**7,806.1**	**874**	**116**	6.2	**51,982**
R208	6,910.9	7,539	234	8.7	291,927	6,910.9	**5,928**	**218**	**8.1**	**230,242**
R209	8,371.8	340	81	7.2	29,284	8,371.8	**301**	**79**	**6.6**	**24,265**
R210	8,813.2	463	93	**8.5**	**34,247**	8,813.2	**407**	**72**	9.3	34,768
R211	7,311.0	641	115	7.8	75,489	**7,311.5**	**491**	**107**	**6.7**	**57,156**
RC204	7,764.9	17,166	**269**	11.9	789,806	7,764.9	**11,550**	277	**10.0**	**443,300**
RC207	9,415.8	808	112	14.3	84,070	9,415.8	**655**	**102**	**13.6**	**64,651**
RC208	7,636.4	3,071	148	22.0	632,136	7,636.4	**2,476**	**140**	**21.3**	**477,627**
#Best	11/14	0/14	1/14	4/14	1/14	14/14	14/14	13/14	11/14	13/14

Table 10: Detailed results for comparing `Default` vs `SetPair` settings using `only-cycle` DSSR after adding non-robust cuts

Instance	Default					SetPair				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	4,454	800	2.1	13,516	5,887.0	**2,783**	**529**	**2.0**	**12,556**
C204	5,881.0	25,207	1,239	**2.4**	75,340	5,881.0	**15,712**	**722**	2.8	**75,129**
R202	**10,296.0**	796	736	**1.7**	3,788	10,292.3	**631**	**592**	1.8	**3,602**
R203	8,708.0	1,239	725	2.1	11,040	8,708.0	**1,053**	**663**	**1.9**	**9,486**
R204	7,313.0	**5,352**	**638**	2.9	73,417	7,313.0	6,055	737	**2.4**	**60,685**
R206	8,759.0	**1,435**	519	2.6	**18,251**	8,759.0	1,541	**503**	**2.4**	18,905
R207	7,940.0	3,034	654	2.4	31,611	7,940.0	**2,374**	**579**	2.3	**27,824**
R208	**6,994.1**	**196,397**	594	5.3	1,125,610	6,989.0	199,145	**574**	**4.9**	**897,083**
R209	8,548.0	**2,071**	**580**	2.4	24,821	8,548.0	2,335	616	**2.2**	**22,051**
R210	9,005.0	**4,354**	674	2.3	30,501	9,005.0	4,487	**631**	2.3	**27,684**
R211	7,467.0	**6,785**	**357**	4.2	**138,224**	7,467.0	10,761	385	**3.7**	140,496
RC204	7,835.0	18,715	**546**	6.5	413,895	7,835.0	**13,344**	558	**5.7**	**244,941**
RC207	9,629.0	**1,683**	459	4.6	34,981	9,629.0	1,810	**442**	**4.4**	**31,714**
RC208	7,761.0	4,626	373	10.0	347,683	7,761.0	**4,172**	**366**	**9.5**	**269,875**
#Best	14/14	7/14	4/14	4/14	2/14	12/14	7/14	10/14	12/14	12/14

Table 11: Detailed results for comparing `Default` vs `SetBased` settings using `all-arcs` DSSR before adding non-robust cuts

Instance	Default					SetBased				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	1,416	**247**	2.2	12,776	5,887.0	**1,364**	324	**1.7**	**8,603**
C204	5,881.0	**8,871**	**552**	1.8	59,903	5,881.0	21,033	1,462	**1.3**	**51,157**
R202	10,164.8	228	116	**3.2**	**7,485**	10,164.8	**205**	**96**	3.4	7,641
R203	8,597.0	509	140	3.8	22,172	8,597.0	**454**	**118**	3.8	**22,035**
R204	7,202.0	1,221	147	4.0	64,842	7,202.0	**1,075**	147	**3.6**	**58,091**
R206	8,600.1	390	99	4.4	22,766	8,600.1	**342**	**91**	4.4	**21,152**
R207	7,805.0	**686**	**120**	**4.1**	**37,176**	**7,851.4**	851	134	4.8	50,241
R208	6,910.9	3,379	**168**	4.5	178,741	6,910.9	**2,909**	177	**3.9**	**131,674**
R209	8,371.8	260	90	**4.2**	**16,451**	8,371.8	**242**	**79**	4.4	17,052
R210	8,813.2	315	82	**5.6**	**24,038**	**8,813.4**	**300**	**73**	6.0	25,711
R211	7,311.0	**343**	**85**	5.2	48,010	7,311.0	365	97	**4.8**	**44,054**
RC204	7,764.9	4,193	**178**	4.6	237,054	7,764.9	**3,474**	212	**3.6**	**163,647**
RC207	9,415.8	338	90	**6.6**	**34,715**	9,415.8	**319**	**81**	7.0	37,632
RC208	7,636.4	904	111	**8.1**	218,110	7,636.4	**786**	**95**	8.5	**203,826**
#Best	12/14	3/14	7/14	8/14	5/14	14/14	11/14	8/14	8/14	9/14

Table 12: Detailed results for comparing `Default` vs `SetBased` settings using `all-arcs` DSSR after adding non-robust cuts

Instance	Default					SetBased				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	1,417	**247**	2.2	12,776	5,887.0	**1,364**	324	**1.7**	**8,603**
C204	5,881.0	**8,871**	**552**	1.8	59,903	5,881.0	21,033	1,462	**1.3**	**51,157**
R202	10,283.2	**488**	**554**	1.5	3,370	**10,289.2**	586	685	**1.4**	**3,056**
R203	8,708.0	856	574	1.7	9,160	8,708.0	**828**	**568**	1.6	**8,764**
R204	7,313.0	3,653	566	1.9	**49,541**	7,313.0	**3,434**	**559**	**1.7**	50,281
R206	8,759.0	1,056	**429**	1.9	14,763	8,759.0	**995**	436	1.8	**14,239**
R207	7,940.0	1,618	481	**1.8**	**21,936**	7,940.0	**1,324**	**299**	2.9	34,890
R208	7,002.0	203,902	**480**	2.4	806,409	**7,005.1**	**187,043**	503	**2.2**	**782,421**
R209	8,548.0	1,784	541	1.7	18,585	8,548.0	**1,631**	**534**	1.7	**18,336**
R210	9,005.0	4,918	**598**	1.7	29,395	9,005.0	**4,244**	620	1.7	**27,285**
R211	7,467.0	5,820	**295**	2.6	95,989	7,467.0	**4,020**	305	2.6	**82,622**
RC204	7,835.0	5,498	**420**	2.6	128,015	7,835.0	**4,953**	488	**2.2**	**95,283**
RC207	9,629.0	1,084	**399**	2.4	**19,037**	9,629.0	**965**	411	2.4	19,383
RC208	7,761.0	1,961	314	3.6	133,620	7,761.0	**1,746**	**308**	**3.4**	**121,038**
#Best	12/14	2/14	9/14	7/14	3/14	14/14	12/14	5/14	13/14	11/14

Table 13: Detailed results for comparing `Default` vs `Pairwise` settings using `all-arcs` DSSR before adding non-robust cuts

Instance	Default					Pairwise				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	**1,416**	**247**	2.2	12,776	5,887.0	1,518	309	**1.8**	**10,577**
C204	5,881.0	8,871	**552**	1.8	59,903	5,881.0	**8,276**	595	**1.6**	**50,065**
R202	10,164.8	228	116	**3.2**	7,485	10,164.8	**211**	**98**	3.3	**6,732**
R203	8,597.0	509	**140**	3.8	22,172	8,597.0	**502**	147	**3.3**	**18,966**
R204	**7,202.0**	1,221	147	4.0	64,842	7,201.7	**1,065**	**142**	**3.8**	**60,584**
R206	8,600.1	390	99	**4.4**	22,766	8,600.1	**339**	**83**	4.5	**21,907**
R207	7,805.0	**686**	**120**	**4.1**	**37,176**	**7,851.4**	933	145	4.7	51,794
R208	6,910.9	3,379	168	4.5	178,741	6,910.9	**2,794**	**157**	**4.3**	**142,616**
R209	8,371.8	260	90	**4.2**	**16,451**	8,371.8	**229**	**69**	4.9	18,478
R210	8,813.2	315	82	5.6	24,038	8,813.2	**298**	**77**	**5.5**	**21,949**
R211	7,311.0	**343**	**85**	5.2	48,010	7,311.0	348	89	**5.0**	**44,801**
RC204	7,764.9	4,193	**178**	4.6	237,054	7,764.9	**3,351**	181	**4.4**	**178,990**
RC207	9,415.8	338	90	**6.6**	**34,715**	9,415.8	**313**	**71**	7.9	39,903
RC208	7,636.4	904	111	8.1	218,110	7,636.4	**826**	**101**	8.1	**191,750**
#Best	13/14	3/14	6/14	6/14	3/14	13/14	11/14	8/14	9/14	11/14

Table 14: Detailed results for comparing `Default` vs `Pairwise` settings using `all-arcs` DSSR after adding non-robust cuts

Instance	Default					Pairwise				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	**1,417**	**247**	2.2	12,776	5,887.0	1,518	309	**1.8**	**10,577**
C204	5,881.0	8,871	**552**	1.8	59,903	5,881.0	**8,276**	595	**1.6**	**50,065**
R202	**10,283.2**	488	554	**1.5**	3,370	10,276.6	**397**	**408**	1.6	**3,103**
R203	8,708.0	**856**	**574**	1.7	9,160	8,708.0	920	701	**1.5**	**7,577**
R204	7,313.0	3,653	566	1.9	**49,541**	7,313.0	**3,336**	**495**	1.9	51,287
R206	8,759.0	1,056	**429**	1.9	14,763	8,759.0	**943**	430	**1.7**	**12,851**
R207	7,940.0	1,618	481	**1.8**	**21,936**	7,940.0	**1,483**	**325**	2.9	37,612
R208	7,002.0	203,902	**480**	2.4	**806,409**	**7,006.0**	**194,571**	492	**2.3**	839,060
R209	8,548.0	**1,784**	541	1.7	18,585	8,548.0	2,035	**530**	1.7	**18,008**
R210	9,005.0	4,918	598	1.7	29,395	9,005.0	**3,394**	**561**	1.7	**25,382**
R211	7,467.0	5,820	295	2.6	95,989	7,467.0	**4,912**	295	2.6	**86,102**
RC204	7,835.0	5,498	**420**	2.6	128,015	7,835.0	**4,683**	437	**2.4**	**96,170**
RC207	9,629.0	**1,084**	399	2.4	**19,037**	9,629.0	1,096	**394**	2.4	19,045
RC208	7,761.0	1,961	**314**	3.6	133,620	7,761.0	**1,766**	315	**3.4**	**116,031**
#Best	13/14	4/14	8/14	7/14	4/14	13/14	10/14	7/14	12/14	10/14

Table 15: Detailed results for comparing `Default` vs `SetPair` settings using `all-arcs` DSSR before adding non-robust cuts

Instance	Default					SetPair				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	**1,416**	**247**	2.2	12,776	5,887.0	1,620	395	**1.5**	**9,900**
C204	5,881.0	**8,871**	**552**	1.8	59,903	5,881.0	14,809	1,073	**1.4**	**48,300**
R202	10,164.8	228	116	**3.2**	7,485	10,164.8	**208**	**101**	3.4	**7,441**
R203	8,597.0	509	140	3.8	22,172	8,597.0	**479**	**122**	3.7	**21,951**
R204	7,202.0	1,221	147	4.0	64,842	7,202.0	**1,104**	**142**	**3.8**	**60,708**
R206	8,600.1	390	99	4.4	22,766	8,600.1	**375**	99	**4.3**	**22,188**
R207	7,805.0	**686**	**120**	**4.1**	**37,176**	**7,851.4**	1,023	154	4.6	53,288
R208	6,910.9	3,379	**168**	4.5	178,741	6,910.9	**2,889**	184	**3.8**	**130,503**
R209	8,371.8	260	90	**4.2**	**16,451**	8,371.8	**227**	**75**	4.5	16,458
R210	8,813.2	315	82	**5.6**	**24,038**	8,813.2	**296**	**70**	6.1	24,779
R211	7,311.0	**343**	**85**	5.2	48,010	7,311.0	357	101	**4.5**	**40,691**
RC204	7,764.9	4,193	**178**	4.6	237,054	7,764.9	**3,212**	185	**4.3**	**163,170**
RC207	9,415.8	338	90	6.6	34,715	9,415.8	**331**	**87**	6.6	**33,335**
RC208	7,636.4	904	**111**	8.1	218,110	7,636.4	**814**	113	**7.2**	**169,089**
#Best	13/14	4/14	8/14	6/14	3/14	14/14	10/14	7/14	10/14	11/14

Table 17: Detailed results for comparing `Default` vs `SetPair` settings using `all-arcs` DSSR after adding non-robust cuts

Instance	Default					SetPair				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C203	5,887.0	**1,417**	**247**	2.2	12,776	5,887.0	1,620	395	**1.5**	**9,900**
C204	5,881.0	**8,871**	**552**	1.8	59,903	5,881.0	14,810	1,073	**1.4**	**48,300**
R202	**10,283.2**	488	554	1.5	3,370	10,281.9	**479**	**499**	1.5	**3,325**
R203	8,708.0	856	574	1.7	9,160	8,708.0	**832**	**563**	1.6	**8,416**
R204	7,313.0	3,653	566	1.9	49,541	7,313.0	**3,585**	**520**	1.9	**46,253**
R206	8,759.0	1,056	429	1.9	14,763	8,759.0	**1,026**	**421**	1.9	**14,288**
R207	7,940.0	1,618	481	**1.8**	**21,936**	7,940.0	**1,549**	**322**	2.9	37,969
R208	**7,002.0**	203,902	480	2.4	806,409	6,998.6	**197,787**	**466**	2.3	**691,352**
R209	8,548.0	1,784	**541**	1.7	18,585	8,548.0	**1,739**	557	1.6	**17,125**
R210	9,005.0	**4,918**	**598**	1.7	29,395	9,005.0	6,411	599	1.7	**28,997**
R211	7,467.0	5,820	295	2.6	95,989	7,467.0	**5,707**	**292**	2.6	**87,119**
RC204	7,835.0	5,498	**420**	2.6	128,015	7,835.0	**4,857**	471	**2.3**	**90,537**
RC207	9,629.0	**1,084**	**399**	2.4	19,037	9,629.0	1,131	403	2.3	**17,888**
RC208	7,761.0	1,961	**314**	3.6	133,620	7,761.0	**1,852**	315	**3.3**	**112,734**
#Best	14/14	4/14	7/14	10/14	1/14	12/14	10/14	7/14	13/14	13/14

A.2 Gehring and Homberger's instances

Table 18: Detailed results for comparing `Default` vs `SetBased` settings using `only-cycle` DSSR before adding non-robust cuts

Instance	Default					SetBased				
	lb	**T(s)**	**#iters**	**#DSSR**	**#labels**	**lb**	**T(s)**	**#iters**	**#DSSR**	**#labels**
C1_210	**26,197.3**	655	**72**	29.3	560,811	26,196.9	**602**	83	**22.8**	**454,343**
C1_2_3	26,597.4	476	**132**	12.4	193,347	26,597.4	**403**	145	**9.8**	**147,833**
C1_2_4	26,197.7	1,066	220	13.9	400,174	26,197.7	**885**	**209**	**12.3**	**325,856**
C1_2_9	26,396.0	219	**58**	13.6	124,522	26,396.0	**211**	59	**11.7**	**106,689**
C2_2_1	19,150.3	**349**	8	1.0	388	19,150.3	355	8	1.0	388
C2_2_2	18,396.0	**9,504**	**794**	2.5	10,412	18,396.0	9,856	919	**1.9**	**7,868**
C2_2_5	18,607.1	1,339	165	3.4	**3,829**	18,607.1	**1,300**	**150**	3.5	4,049
C2_2_6	18,406.7	4,173	324	5.3	15,499	18,406.7	**3,429**	**302**	**4.6**	**13,644**
C2_2_7	18,356.4	6,342	**429**	5.3	22,082	18,356.4	**6,322**	454	**4.9**	**20,152**
C2_2_8	18,082.4	7,977	363	8.0	42,366	18,082.4	**6,458**	**296**	8.0	**40,892**
C2_2_9	18,011.7	13,272	486	**6.7**	56,421	**18,011.9**	**9,980**	**366**	7.2	**54,808**
R1_210	32,450.2	153	**26**	16.3	210,725	**32,451.2**	**134**	27	**13.9**	**177,834**
R1_2_4	**30,010.4**	270	43	**13.2**	**357,645**	30,009.9	**215**	**40**	13.4	364,716
R1_2_5	40,068.1	42	12	**2.0**	**9,881**	40,068.1	**41**	**11**	2.2	10,827
R1_2_6	34,930.6	101	41	4.6	**53,930**	34,930.6	**92**	**37**	**4.4**	54,508
R1_2_7	31,000.6	237	51	8.7	190,129	31,000.6	**190**	51	**7.9**	**170,515**
R1_2_8	29,032.2	356	**34**	19.8	589,966	**29,032.3**	**261**	36	**17.0**	**487,423**
R1_2_9	36,788.3	68	**17**	7.4	52,206	36,788.3	**64**	20	**6.2**	**44,247**
R2_2_1	34,625.1	**1,645**	**88**	2.7	1,232	34,625.1	1,734	110	**2.4**	**1,097**
R2_2_2	30,006.1	5,397	282	5.1	12,466	30,006.1	**4,943**	**270**	**4.6**	**11,252**
R2_2_5	30,290.3	2,946	188	**3.9**	**4,670**	30,290.3	**2,798**	**144**	4.2	4,843
R2_2_6	26,455.6	9,738	414	5.9	39,427	26,455.6	**8,397**	**392**	**5.2**	**30,201**
R2_2_9	28,206.0	4,217	252	4.7	11,431	28,206.0	**3,836**	**207**	**4.5**	**9,902**
RC1_2_1	34,595.9	54	18	**7.4**	**43,828**	34,595.9	**52**	**17**	7.6	44,773
RC1_2_2	31,774.7	127	44	**7.7**	**103,660**	31,774.8	**118**	**43**	8.0	111,973
RC1_2_6	32,554.7	160	**20**	34.2	402,772	32,554.8	**137**	23	**26.7**	**324,806**
RC1_2_8	30,216.1	462	**34**	45.3	1,202,445	30,216.1	**361**	38	**38.2**	**995,481**
RC2_2_1	27,891.0	1,927	**104**	7.3	6,676	**27,892.7**	**1,913**	115	**5.6**	**5,196**
RC2_2_2	24,588.6	11,588	529	7.5	41,155	**24,588.7**	**9,785**	**511**	**6.2**	**31,888**
RC2_2_5	24,788.5	7,383	493	8.3	88,212	24,788.5	**6,263**	**475**	**7.5**	**67,406**
RC2_2_6	24,724.4	5,859	433	8.2	49,520	24,724.4	**5,067**	**388**	**7.7**	**41,217**
#Best	26/31	3/31	14/31	9/31	8/31	29/31	28/31	19/31	25/31	24/31

Table 19: Detailed results for comparing `Default` vs `SetBased` settings using `only-cycle` DSSR after adding non-robust cuts

Instance	Default					SetBased				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C1_2_10	26,247.0	822	**192**	13.9	265,467	26,247.0	**782**	201	**12.4**	**247,868**
C1_2_3	26,719.2	123,963	**433**	5.8	152,581	**26,721.9**	**39,138**	505	**4.7**	**129,423**
C1_2_4	26,256.0	1,357	374	10.1	304,547	26,256.0	**1,151**	**357**	**9.2**	**260,764**
C1_2_9	26,396.0	220	**58**	13.6	124,522	26,396.0	**212**	59	**11.7**	**106,689**
C2_2_1	19,152.8	**402**	**70**	1.0	455	**19,154.4**	414	74	1.0	**436**
C2_2_2	**18,514.0**	13,515	**1,637**	1.8	7,683	18,492.1	**13,421**	1,699	**1.5**	**6,113**
C2_2_5	18,696.0	2,091	562	1.9	**2,717**	18,696.0	**2,031**	**530**	1.8	2,997
C2_2_6	18,448.0	5,729	852	2.8	**8,996**	18,448.0	**4,328**	**619**	2.9	9,145
C2_2_7	18,422.0	**7,449**	**790**	3.5	14,261	18,422.0	7,748	933	**3.0**	**12,193**
C2_2_8	**18,137.0**	9,221	696	4.8	25,811	18,135.8	**7,987**	**654**	**4.5**	**23,848**
C2_2_9	18,150.0	21,618	1,142	3.9	44,040	18,150.0	**17,313**	**957**	3.9	**42,650**
R1_2_10	32,792.4	2,041	390	3.1	86,138	**32,793.1**	**1,958**	390	**2.9**	**81,608**
R1_2_4	**30,367.6**	32,462	576	**3.6**	410,047	30,363.9	**17,440**	**507**	3.8	**386,889**
R1_2_5	40,457.1	423	**253**	1.1	7,508	40,457.0	**411**	258	1.1	**7,360**
R1_2_6	35,454.7	**1,980**	**436**	2.0	**55,897**	**35,457.1**	2,014	467	1.9	56,159
R1_2_7	**31,393.5**	3,589	452	3.2	175,536	31,390.9	**2,872**	**412**	3.3	**164,077**
R1_2_8	29,328.1	**6,201**	**452**	4.7	379,879	**29,330.1**	6,478	468	**4.5**	**368,905**
R1_2_9	37,267.4	688	**291**	1.7	**21,969**	**37,268.2**	**676**	301	1.7	22,490
R2_2_1	34,680.0	**1,756**	**268**	1.6	710	34,680.0	1,860	294	1.6	**709**
R2_2_2	30,082.0	6,215	**568**	3.1	7,089	30,082.0	**6,069**	665	**2.6**	**5,664**
R2_2_5	30,609.2	5,358	1,405	1.5	**3,232**	**30,609.5**	**5,294**	**1,336**	1.4	3,312
R2_2_6	26,750.2	**17,642**	**2,167**	2.0	13,086	**26,750.5**	19,404	2,765	**1.7**	**10,458**
R2_2_9	**28,433.0**	5,808	992	2.0	5,578	28,415.2	**5,188**	**877**	2.0	**4,926**
RC1_2_1	34,993.5	**632**	**312**	1.4	**14,926**	**34,998.0**	728	329	1.4	15,720
RC1_2_2	32,068.1	**25,064**	465	2.5	101,532	**32,068.8**	34,489	**453**	2.5	**100,049**
RC1_2_6	32,936.9	5,602	**498**	3.8	127,355	**32,939.9**	**5,157**	544	**3.5**	**126,882**
RC1_2_8	**30,502.0**	125,068	585	7.3	**926,295**	30,499.4	**109,219**	**562**	7.4	945,591
RC2_2_1	27,960.8	2,280	638	2.1	**1,904**	**27,961.8**	**2,231**	**531**	2.1	2,009
RC2_2_2	**24,780.0**	**49,224**	6,730	1.6	10,137	24,772.0	55,776	**6,448**	**1.5**	**8,972**
RC2_2_5	24,914.0	15,122	**3,160**	2.3	30,460	24,914.0	**14,721**	3,309	**2.1**	**26,173**
RC2_2_6	24,951.0	10,970	2,403	2.6	19,502	24,951.0	**8,951**	**2,053**	2.5	**17,478**
#Best	19/31	9/31	17/31	17/31	8/31	24/31	22/31	15/31	30/31	23/31

Table 20: Detailed results for comparing `Default` vs `Pairwise` settings using `only-cycle` DSSR before adding non-robust cuts

Instance	Default					Pairwise				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C1_2_10	26,197.3	655	**72**	29.3	560,811	26,197.3	**595**	80	**23.9**	**427,899**
C1_2_3	26,597.4	476	132	12.4	193,347	26,597.4	**390**	**111**	**8.5**	**120,149**
C1_2_4	26,197.7	1,066	220	13.9	400,174	26,197.7	**965**	**196**	**13.8**	**353,881**
C1_2_9	26,396.0	219	**58**	13.6	124,522	26,396.0	**215**	68	**9.5**	**84,917**
C2_2_1	19,150.3	**349**	8	1.0	388	19,150.3	350	8	1.0	388
C2_2_2	18,396.0	9,504	794	2.5	10,412	18,396.0	**5,983**	**610**	**2.2**	**8,299**
C2_2_5	18,607.1	1,339	165	3.4	3,829	18,607.1	**1,258**	**158**	3.3	**3,662**
C2_2_6	18,406.7	4,173	324	**5.3**	15,499	18,406.7	**3,168**	**243**	5.5	**14,472**
C2_2_7	18,356.4	6,342	**429**	5.3	22,082	18,356.4	**5,605**	436	**4.7**	**17,806**
C2_2_8	18,082.4	7,977	363	8.0	42,366	18,082.4	**6,428**	**334**	**7.1**	**32,202**
C2_2_9	18,011.7	13,272	486	6.7	56,421	18,011.8	**10,553**	**417**	**6.5**	**46,727**
R1_2_10	32,450.2	**153**	**26**	16.3	210,725	**32,451.3**	157	27	**14.7**	**191,588**
R1_2_4	30,010.4	**270**	43	**13.2**	**357,645**	**30,010.6**	291	**38**	14.1	386,546
R1_2_5	40,068.1	42	12	2.0	9,881	40,068.1	**42**	12	2.0	9,881
R1_2_6	34,930.6	101	**41**	4.6	53,930	34,930.6	**98**	42	**4.3**	**49,923**
R1_2_7	31,000.6	**237**	51	**8.7**	**190,129**	31,000.6	243	**47**	9.8	213,354
R1_2_8	29,032.2	356	**34**	19.8	589,966	29,032.1	**327**	38	**16.8**	**476,911**
R1_2_9	36,788.3	**68**	**17**	7.4	52,206	36,788.3	68	18	**6.7**	**47,286**
R2_2_1	34,625.1	**1,645**	**88**	2.7	1,232	34,625.1	1,718	92	**2.5**	**1,156**
R2_2_2	30,006.1	5,397	**282**	5.1	12,466	30,006.1	**4,965**	290	**4.4**	**10,221**
R2_2_5	30,290.3	2,946	188	3.9	4,670	30,290.3	**2,764**	**163**	3.8	**4,325**
R2_2_6	26,455.6	9,738	414	5.9	39,427	26,455.6	**8,909**	**387**	**5.4**	**31,820**
R2_2_9	28,206.0	4,217	252	**4.7**	11,431	28,206.1	**3,639**	**187**	5.0	**11,062**
RC1_2_1	34,595.9	**54**	18	7.4	43,828	34,595.9	54	18	7.4	43,828
RC1_2_2	31,774.7	127	44	7.7	103,660	31,774.7	**123**	44	**7.2**	**96,327**
RC1_2_6	32,554.7	160	20	34.2	402,772	32,554.7	**152**	20	**31.2**	**361,162**
RC1_2_8	30,216.1	462	**34**	45.3	1,202,445	30,216.1	**439**	36	**41.7**	**1,062,486**
RC2_2_1	27,891.0	1,927	**104**	7.3	6,676	**27,892.7**	**1,890**	111	**5.9**	**5,281**
RC2_2_2	24,588.6	11,588	**529**	7.5	41,155	**24,592.5**	**10,071**	549	**6.0**	**29,763**
RC2_2_5	24,788.5	7,383	493	**8.3**	88,212	24,788.5	**6,327**	**387**	9.6	**84,634**
RC2_2_6	24,724.4	5,859	433	8.2	49,520	24,724.5	**5,259**	**381**	**8.0**	**43,544**
#Best	27/31	7/31	17/31	10/31	5/31	31/31	24/31	19/31	26/31	29/31

Table 21: Detailed results for comparing `Default` vs `Pairwise` settings using `only-cycle` DSSR after adding non-robust cuts

Instance	Default					Pairwise				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C1_210	26,247.0	822	**192**	13.9	265,467	26,247.0	**763**	206	**12.3**	**220,337**
C1_2_3	26,719.2	123,963	**433**	5.8	152,581	**26,720.1**	**57,438**	458	**4.5**	**117,110**
C1_2_4	26,256.0	1,357	374	10.1	304,547	26,256.0	**1,250**	**347**	**10.0**	**272,266**
C1_2_9	26,396.0	220	**58**	13.6	124,522	26,396.0	**216**	68	**9.5**	**84,917**
C2_2_1	19,152.8	**402**	70	1.0	455	19,152.8	402	70	1.0	455
C2_2_2	18,514.0	13,515	1,637	1.8	7,683	18,514.0	**9,767**	**1,393**	**1.6**	**7,116**
C2_2_5	18,696.0	2,091	562	1.9	**2,717**	18,696.0	**1,957**	**517**	1.8	2,903
C2_2_6	18,448.0	5,729	852	2.8	8,996	18,448.0	**4,559**	**761**	**2.6**	**7,490**
C2_2_7	18,422.0	7,449	790	3.5	14,261	18,422.0	**6,624**	**785**	**3.2**	**12,041**
C2_2_8	18,137.0	9,221	**696**	4.8	25,811	18,137.0	**8,008**	781	**3.8**	**18,482**
C2_2_9	18,150.0	21,618	1,142	3.9	44,040	18,150.0	**18,888**	**1,001**	3.8	**41,764**
R1_210	32,792.4	**2,041**	**390**	3.1	86,138	**32,794.6**	2,377	410	**2.8**	**79,223**
R1_2_4	**30,367.6**	32,462	576	**3.6**	410,047	30,362.2	**22,112**	**509**	3.8	**399,613**
R1_2_5	40,457.1	423	253	1.1	7,508	40,457.1	**422**	253	1.1	7,508
R1_2_6	35,454.7	**1,980**	436	2.0	**55,897**	**35,457.0**	2,235	**430**	2.0	58,068
R1_2_7	**31,393.5**	3,589	452	**3.2**	**175,536**	31,392.3	**3,379**	**413**	3.4	177,490
R1_2_8	29,328.1	6,201	**452**	4.7	379,879	**29,328.3**	**5,554**	466	**4.4**	**340,491**
R1_2_9	37,267.4	**688**	**291**	1.7	**21,969**	**37,271.3**	807	307	1.6	22,444
R2_2_1	34,680.0	**1,756**	268	1.6	**710**	34,680.0	1,823	**230**	1.7	749
R2_2_2	30,082.0	6,215	568	3.1	7,089	**30,088.0**	**5,687**	**561**	**2.8**	**6,153**
R2_2_5	30,609.2	5,358	**1,405**	1.5	**3,232**	**30,610.3**	**5,225**	1,434	1.4	3,339
R2_2_6	**26,750.2**	**17,642**	**2,167**	2.0	13,086	26,748.6	18,751	2,462	**1.8**	**11,163**
R2_2_9	**28,433.0**	5,808	**992**	2.0	5,578	28,429.1	**5,232**	1,005	**1.8**	**4,638**
RC1_2_1	34,993.5	632	312	1.4	14,926	34,993.5	**631**	312	1.4	14,926
RC1_2_2	32,068.1	**25,064**	465	2.5	**101,532**	**32,072.1**	28,047	**448**	2.5	102,412
RC1_2_6	32,936.9	**5,602**	**498**	3.8	**127,355**	**32,941.0**	6,413	532	**3.6**	129,387
RC1_2_8	**30,502.0**	**125,068**	585	**7.3**	**926,295**	30,500.0	137,624	**547**	7.7	964,933
RC2_2_1	**27,960.8**	2,280	638	2.1	**1,904**	27,932.2	**2,153**	**491**	2.2	1,964
RC2_2_2	**24,780.0**	**49,224**	**6,730**	1.6	10,137	24,776.2	57,708	7,092	**1.4**	**8,741**
RC2_2_5	24,914.0	15,122	3,160	**2.3**	30,460	24,914.0	**10,197**	**1,915**	2.8	**29,270**
RC2_2_6	24,951.0	10,970	2,403	2.6	19,502	24,951.0	**9,325**	**2,325**	**2.3**	**15,720**
#Best	22/31	10/31	15/31	15/31	13/31	24/31	21/31	19/31	27/31	21/31

Table 22: Detailed results for comparing `Default` vs `SetPair` settings using `only-cycle` DSSR before adding non-robust cuts

Instance	Default					SetPair				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C1_210	26,197.3	655	**72**	29.3	560,811	26,197.3	**546**	74	**24.9**	**445,636**
C1_2_3	26,597.4	476	**132**	12.4	193,347	26,597.4	**412**	147	**10.3**	**162,955**
C1_2_4	26,197.7	1,066	220	13.9	400,174	26,197.7	**821**	**209**	**12.5**	**328,362**
C1_2_9	26,396.0	219	58	13.6	124,522	26,396.0	**198**	**52**	**12.2**	**108,642**
C2_2_1	19,150.3	**349**	8	1.0	388	19,150.3	357	8	1.0	388
C2_2_2	18,396.0	9,504	794	2.5	10,412	18,396.0	**6,668**	**739**	**1.8**	**6,935**
C2_2_5	18,607.1	1,339	165	3.4	3,829	18,607.1	**1,257**	**155**	3.4	**3,698**
C2_2_6	18,406.7	4,173	324	5.3	15,499	18,406.7	**3,605**	**322**	**4.6**	**12,457**
C2_2_7	18,356.4	6,342	**429**	5.3	22,082	18,356.4	**5,688**	496	**4.1**	**15,732**
C2_2_8	18,082.4	7,977	363	8.0	42,366	18,082.4	**6,843**	**333**	**7.4**	**34,009**
C2_2_9	18,011.7	13,272	486	**6.7**	56,421	**18,011.9**	**9,361**	**340**	7.6	**54,178**
R1_210	32,450.2	153	26	16.3	210,725	**32,451.3**	**135**	**25**	**16.0**	**206,761**
R1_2_4	**30,010.4**	270	43	13.2	357,645	30,009.9	**213**	**41**	**12.6**	**330,678**
R1_2_5	40,068.1	42	12	2.0	9,881	40,068.1	**42**	12	2.0	**9,874**
R1_2_6	34,930.6	101	41	4.6	**53,930**	34,930.6	**97**	**40**	4.7	56,384
R1_2_7	31,000.6	237	51	8.7	190,129	**31,000.8**	**201**	**49**	**8.4**	**179,043**
R1_2_8	29,032.2	356	**34**	19.8	589,966	29,032.2	**276**	39	**15.7**	**448,479**
R1_2_9	36,788.3	68	17	7.4	52,206	**36,788.6**	**65**	17	**7.1**	**50,041**
R2_2_1	34,625.1	**1,645**	88	**2.7**	**1,232**	34,625.1	1,679	**70**	3.0	1,396
R2_2_2	30,006.1	5,397	282	5.1	12,466	30,006.1	**4,832**	**270**	**4.6**	**11,151**
R2_2_5	30,290.3	2,946	188	**3.9**	**4,670**	30,290.3	**2,657**	**131**	4.5	5,092
R2_2_6	26,455.6	9,738	414	**5.9**	39,427	26,455.5	**8,142**	**329**	6.1	**35,795**
R2_2_9	28,206.0	4,217	252	4.7	11,431	28,206.0	**3,938**	**223**	**4.3**	**9,247**
RC1_2_1	34,595.9	54	18	**7.4**	**43,828**	34,595.9	**52**	**17**	7.8	45,483
RC1_2_2	31,774.7	127	44	7.7	103,660	31,774.7	**115**	44	7.6	**97,615**
RC1_2_6	32,554.7	160	**20**	34.2	402,772	32,554.8	**143**	23	**28.1**	**331,637**
RC1_2_8	**30,216.1**	462	**34**	45.3	1,202,445	30,215.7	**384**	35	**41.9**	**1,044,786**
RC2_2_1	27,891.0	**1,927**	**104**	7.3	6,676	**27,892.7**	1,956	127	**5.1**	**4,572**
RC2_2_2	24,588.6	11,588	529	7.5	41,155	**24,590.9**	**9,829**	**499**	**6.6**	**31,844**
RC2_2_5	24,788.5	7,383	493	**8.3**	88,212	**24,797.9**	**6,244**	**384**	9.8	**88,198**
RC2_2_6	24,724.4	5,859	433	8.2	49,520	24,724.4	**5,233**	**374**	**8.1**	**42,073**
#Best	24/31	3/31	11/31	11/31	5/31	29/31	28/31	24/31	25/31	27/31

Table 23: Detailed results for comparing `Default` vs `SetPair` settings using `only-cycle` DSSR after adding non-robust cuts

Instance	Default					SetPair				
	lb	**T(s)**	**#iters**	**#DSSR**	**#labels**	**lb**	**T(s)**	**#iters**	**#DSSR**	**#labels**
C1_210	26,247.0	822	**192**	13.9	265,467	26,247.0	**725**	203	**12.2**	**218,519**
C1_2_3	26,719.2	**123,963**	**433**	5.8	152,581	**26,721.5**	241,285	481	**5.0**	**141,088**
C1_2_4	26,256.0	1,357	374	10.1	304,547	26,256.0	**1,096**	**355**	**9.3**	**256,656**
C1_2_9	26,396.0	220	58	13.6	124,522	26,396.0	**199**	**52**	**12.2**	**108,642**
C2_2_1	19,152.8	**402**	**70**	1.0	**455**	**19,156.4**	430	98	1.0	475
C2_2_2	18,514.0	13,515	1,637	1.8	7,683	18,514.0	**10,401**	**1,630**	**1.4**	**5,195**
C2_2_5	18,696.0	2,091	562	1.9	2,717	18,696.0	**1,994**	**556**	1.8	**2,642**
C2_2_6	18,448.0	5,729	852	2.8	8,996	18,448.0	**4,944**	**831**	**2.6**	**7,256**
C2_2_7	18,422.0	7,449	**790**	3.5	14,261	18,422.0	**7,322**	1,054	**2.6**	**9,629**
C2_2_8	18,137.0	9,221	696	4.8	25,811	18,137.0	**8,071**	**674**	**4.3**	**20,740**
C2_2_9	18,150.0	21,618	1,142	**3.9**	44,040	18,150.0	**16,638**	**928**	4.1	**41,527**
R1_210	32,792.4	**2,041**	**390**	3.1	86,138	**32,794.9**	2,365	418	**2.9**	**82,495**
R1_2_4	**30,367.6**	**32,462**	576	3.6	410,047	30,366.9	40,007	**559**	3.5	**374,689**
R1_2_5	**40,457.1**	423	253	1.1	7,508	40,451.6	**328**	**221**	1.1	**7,251**
R1_2_6	35,454.7	**1,980**	**436**	2.0	55,897	**35,462.4**	2,497	509	1.9	**55,871**
R1_2_7	**31,393.5**	**3,589**	452	3.2	175,536	31,392.5	4,044	**430**	3.3	**172,441**
R1_2_8	29,328.1	**6,201**	**452**	4.7	379,879	**29,331.0**	8,925	487	**4.3**	**371,794**
R1_2_9	37,267.4	**688**	**291**	1.7	**21,969**	**37,272.2**	806	323	1.6	22,272
R2_2_1	34,680.0	**1,756**	268	**1.6**	**710**	34,680.0	1,803	**222**	1.7	751
R2_2_2	30,082.0	6,215	568	3.1	7,089	30,082.0	**5,723**	**563**	**2.8**	**6,366**
R2_2_5	**30,609.2**	5,358	1,405	1.5	3,232	30,603.9	**4,552**	**1,206**	1.5	**2,790**
R2_2_6	**26,750.2**	17,642	**2,167**	2.0	13,086	26,749.5	**17,406**	2,348	**1.8**	**11,234**
R2_2_9	**28,433.0**	5,808	**992**	2.0	5,578	28,428.0	**5,487**	1,018	**1.8**	**4,622**
RC1_2_1	34,993.5	**632**	**312**	1.4	14,926	**34,994.6**	659	318	1.4	**14,533**
RC1_2_2	32,068.1	**25,064**	**465**	2.5	101,532	**32,074.2**	57,720	494	**2.4**	**101,310**
RC1_2_6	32,936.9	**5,602**	**498**	3.8	127,355	**32,942.6**	8,596	558	**3.4**	**125,762**
RC1_2_8	**30,502.0**	**125,068**	585	**7.3**	926,295	30,497.4	136,304	**541**	7.8	**898,946**
RC2_2_1	27,960.8	**2,280**	638	2.1	**1,904**	**27,961.1**	2,283	**555**	2.1	1,916
RC2_2_2	24,780.0	**49,224**	**6,730**	1.6	10,137	24,780.0	68,376	7,839	**1.4**	**8,530**
RC2_2_5	24,914.0	15,122	3,160	2.3	30,460	24,914.0	**13,006**	**2,856**	2.3	**27,022**
RC2_2_6	24,951.0	10,970	**2,403**	2.6	19,502	24,951.0	**9,706**	2,543	**2.2**	**14,405**
#Best	21/31	15/31	15/31	14/31	4/31	24/31	16/31	16/31	28/31	27/31

Table 24: Detailed results for comparing `Default` vs `SetBased` settings using `all-arcs` DSSR before adding non-robust cuts

Instance	Default					SetBased				
	lb	**T(s)**	**#iters**	**#DSSR**	**#labels**	**lb**	**T(s)**	**#iters**	**#DSSR**	**#labels**
C1_2_10	**26,197.3**	477	74	**16.1**	**310,753**	26,197.2	**381**	**61**	16.5	319,202
C1_2_3	26,597.4	323	113	**4.1**	**73,609**	26,597.4	**253**	**105**	4.8	82,096
C1_2_4	26,197.7	525	**157**	6.2	197,897	26,197.7	**459**	177	**4.9**	**152,005**
C1_2_9	26,396.0	**184**	55	11.1	**97,994**	26,396.0	200	**51**	11.1	103,308
C2_2_1	19,150.3	**359**	8	1.0	388	19,150.3	362	8	1.0	388
C2_2_2	18,394.0	**3,905**	**518**	1.4	6,134	**18,396.0**	4,645	561	1.3	**5,711**
C2_2_5	18,607.1	**1,259**	**151**	3.3	3,707	18,607.1	1,272	165	**2.9**	**3,512**
C2_2_6	18,406.7	3,079	252	**4.1**	**11,771**	18,406.7	**2,684**	**211**	4.6	13,276
C2_2_7	18,356.4	3,905	360	3.4	13,757	18,356.4	**3,612**	360	**3.2**	**13,574**
C2_2_8	18,082.4	3,564	232	**5.0**	**23,873**	18,082.4	**3,412**	**200**	5.5	27,222
C2_2_9	**18,011.9**	6,020	306	**4.5**	**35,977**	18,011.2	**5,490**	**268**	4.8	40,270
R1_2_10	32,451.3	132	**21**	14.2	183,530	32,451.3	**123**	25	**12.0**	**160,451**
R1_2_4	**30,010.5**	198	**36**	8.7	238,385	30,009.9	**158**	40	**7.5**	**193,054**
R1_2_5	40,068.1	43	12	**2.0**	**9,881**	40,068.1	**42**	**11**	2.2	10,827
R1_2_6	34,930.6	93	44	**3.5**	**39,224**	34,930.6	**88**	**36**	3.9	48,222
R1_2_7	**31,001.0**	189	51	**5.6**	**117,367**	31,000.4	**163**	**47**	5.8	124,032
R1_2_8	29,032.3	250	35	**10.2**	**302,465**	29,032.3	**194**	**31**	11.0	318,416
R1_2_9	36,788.5	68	18	6.4	**46,453**	36,788.5	**66**	**17**	6.5	47,789
R2_2_1	34,625.1	1,751	**106**	2.3	1,060	34,625.1	**1,682**	115	**2.1**	**979**
R2_2_2	30,006.1	4,391	291	**3.2**	**7,647**	30,006.1	**3,933**	**217**	3.8	9,269
R2_2_5	30,290.3	**2,681**	**146**	3.4	4,062	30,290.3	2,713	150	**3.2**	**3,787**
R2_2_6	26,455.5	6,253	278	4.3	**28,234**	26,455.6	**5,382**	**239**	**4.2**	28,634
R2_2_9	28,206.0	3,616	205	**3.4**	**8,047**	28,206.0	**3,471**	**173**	3.6	8,488
RC1_2_1	34,595.9	54	18	**7.4**	**43,828**	34,595.9	**51**	**17**	7.6	44,773
RC1_2_2	31,774.7	112	**41**	6.5	**84,161**	31,774.7	**101**	42	**6.3**	84,595
RC1_2_6	32,554.8	121	20	21.9	255,845	**32,554.9**	**103**	20	**20.8**	**243,494**
RC1_2_8	30,216.1	236	30	19.3	517,598	30,216.1	**194**	**29**	**18.9**	**495,194**
RC2_2_1	27,891.0	1,873	117	4.9	4,509	27,891.0	**1,831**	**111**	**4.5**	**4,382**
RC2_2_2	24,588.6	7,141	490	3.7	22,238	**24,588.8**	**6,532**	**450**	3.6	**21,101**
RC2_2_5	24,788.5	**3,338**	**263**	4.9	**47,954**	24,788.5	3,432	273	4.9	48,149
RC2_2_6	24,724.4	3,484	276	4.7	28,380	24,724.4	**3,311**	**269**	4.7	**26,591**
#Best	28/31	6/31	12/31	19/31	18/31	27/31	25/31	22/31	19/31	14/31

Table 25: Detailed results for comparing `Default` vs `SetBased` settings using `all-arcs` DSSR after adding non-robust cuts

Instance	Default					SetBased				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C1_210	26,247.0	616	184	8.1	**19,180**	26,247.0	**492**	**167**	**8.0**	19,513
C1_2_3	**26,723.2**	**34,047**	498	**2.0**	**32,563**	26,717.7	44,182	**410**	2.3	33,328
C1_2_4	26,256.0	724	**283**	4.2	34,120	26,256.0	**654**	319	**3.4**	**33,587**
C1_2_9	26,396.0	**185**	55	11.1	**8,836**	26,396.0	201	**51**	11.1	9,292
C2_2_1	19,152.8	**413**	**70**	1.0	455	**19,154.4**	421	74	1.0	**436**
C2_2_2	**18,514.0**	12,368	1,415	1.2	5,863	18,489.1	**7,611**	**1,234**	1.2	**4,423**
C2_2_5	18,696.0	**2,008**	**561**	1.7	**1,602**	18,696.0	2,034	596	**1.6**	1,685
C2_2_6	18,448.0	4,111	**628**	2.3	**3,102**	18,448.0	**3,768**	653	**2.2**	3,104
C2_2_7	18,422.0	**4,968**	**737**	2.2	**4,033**	18,422.0	4,998	833	**2.0**	4,449
C2_2_8	18,137.0	4,633	577	**2.7**	5,321	18,137.0	**4,390**	**513**	2.8	**5,038**
C2_2_9	18,150.0	11,608	977	**2.2**	**10,730**	18,150.0	**10,306**	**800**	2.4	12,342
R1_210	**32,794.9**	2,140	417	2.2	**29,849**	32,793.0	**1,836**	**415**	2.2	30,208
R1_2_4	**30,364.6**	18,415	**533**	2.0	**127,556**	30,364.1	**15,173**	551	1.9	128,498
R1_2_5	40,457.1	427	**253**	1.1	7,141	40,457.0	**418**	258	1.1	**7,007**
R1_2_6	35,456.9	**1,995**	461	1.6	**29,240**	**35,461.9**	2,074	**458**	1.5	31,203
R1_2_7	**31,392.8**	2,506	422	2.1	54,672	31,391.1	**2,333**	**406**	2.1	**54,599**
R1_2_8	29,328.1	**4,897**	**451**	2.4	**100,373**	**29,330.2**	5,120	462	2.3	103,922
R1_2_9	**37,270.4**	750	308	1.5	13,958	37,266.9	**640**	**285**	1.6	**13,359**
R2_2_1	34,680.0	1,851	**230**	1.6	**452**	34,680.0	**1,806**	277	**1.5**	460
R2_2_2	30,082.0	5,271	594	**2.1**	**2,213**	**30,102.2**	**4,501**	**464**	2.3	2,287
R2_2_5	**30,609.8**	**4,836**	1,302	1.3	**2,128**	30,609.6	4,842	**1,292**	1.3	2,140
R2_2_6	**26,748.4**	13,208	**1,972**	1.5	**6,390**	26,747.2	**13,188**	2,240	**1.4**	6,780
R2_2_9	**28,433.0**	**4,994**	**911**	1.6	**2,725**	28,430.3	5,080	962	1.5	2,850
RC1_2_1	34,993.5	**634**	**312**	1.4	**10,372**	**34,998.0**	734	329	1.4	11,170
RC1_2_2	32,072.6	28,285	495	1.8	**41,646**	**32,073.3**	**25,273**	**474**	1.9	45,399
RC1_2_6	**32,938.2**	5,123	503	2.7	36,610	32,936.1	**3,672**	**476**	2.7	**36,397**
RC1_2_8	30,499.5	59,534	**543**	3.1	**133,139**	**30,500.3**	**45,722**	553	3.0	141,567
RC2_2_1	**27,966.5**	2,242	682	1.7	**909**	27,961.8	**2,124**	**550**	1.8	954
RC2_2_2	**24,780.0**	31,302	4,948	1.3	**6,565**	24,769.8	**26,760**	**4,129**	1.3	6,795
RC2_2_5	24,914.0	**7,532**	**1,817**	1.6	**14,139**	24,914.0	9,052	2,334	**1.5**	15,662
RC2_2_6	24,951.0	7,288	2,026	1.6	**8,280**	24,951.0	**6,840**	**1,953**	1.5	9,069
#Best	24/31	11/31	14/31	23/31	23/31	19/31	20/31	17/31	27/31	8/31

Table 26: Detailed results for comparing `Default` vs `Pairwise` settings using `all-arcs` DSSR before adding non-robust cuts

Instance	Default					Pairwise				
	lb	**T(s)**	**#iters**	**#DSSR**	**#labels**	**lb**	**T(s)**	**#iters**	**#DSSR**	**#labels**
C1_2_10	26,197.3	477	74	**16.1**	310,753	26,197.3	**432**	**66**	16.4	**295,977**
C1_2_3	26,597.4	323	**113**	4.1	73,609	26,597.4	**260**	118	4.2	**67,797**
C1_2_4	26,197.7	525	**157**	6.2	197,897	26,197.7	**423**	167	**5.2**	**147,922**
C1_2_9	26,396.0	**184**	**55**	11.1	97,994	26,396.0	187	60	**9.5**	**82,941**
C2_2_1	19,150.3	359	8	1.0	388	19,150.3	**351**	8	1.0	388
C2_2_2	18,394.0	**3,905**	**518**	1.4	6,134	**18,396.0**	4,871	615	1.4	**5,671**
C2_2_5	18,607.1	1,259	151	3.3	3,707	18,607.1	**1,167**	**141**	**3.1**	**3,436**
C2_2_6	18,406.7	3,079	252	**4.1**	**11,771**	18,406.7	**2,695**	**204**	4.6	12,090
C2_2_7	18,356.4	**3,905**	360	**3.4**	**13,757**	18,356.4	4,026	**339**	3.6	13,981
C2_2_8	18,082.4	3,564	232	**5.0**	23,873	18,082.4	**3,405**	**203**	5.3	**23,366**
C2_2_9	18,011.9	6,020	306	4.5	35,977	18,012.0	**4,696**	**285**	**4.3**	**31,165**
R1_2_10	**32,451.3**	132	**21**	14.2	183,530	32,450.6	**128**	24	**11.6**	**150,038**
R1_2_4	**30,010.5**	198	**36**	8.7	238,385	30,009.9	**188**	43	**6.9**	**178,591**
R1_2_5	40,068.1	43	12	2.0	9,881	40,068.1	**42**	12	2.0	9,881
R1_2_6	34,930.6	**93**	44	3.5	**39,224**	34,930.6	94	**42**	3.5	40,691
R1_2_7	31,001.0	189	51	**5.6**	**117,367**	**31,001.2**	**187**	**46**	5.9	122,804
R1_2_8	29,032.3	250	**35**	10.2	302,465	29,032.3	**245**	36	**9.8**	**286,602**
R1_2_9	**36,788.5**	68	**18**	6.4	46,453	36,788.3	**67**	19	**5.9**	**42,793**
R2_2_1	34,625.1	**1,751**	**106**	2.3	1,060	34,625.1	1,782	124	**2.0**	**899**
R2_2_2	30,006.1	4,391	291	**3.2**	**7,647**	30,006.1	**4,016**	**246**	3.4	8,324
R2_2_5	30,290.3	2,681	**146**	3.4	4,062	30,290.3	**2,640**	153	**3.1**	**3,579**
R2_2_6	26,455.5	6,253	**278**	4.3	28,234	26,455.5	**5,762**	294	**3.5**	**22,616**
R2_2_9	28,206.0	3,616	205	**3.4**	8,047	28,206.0	**3,339**	**162**	3.6	**8,044**
RC1_2_1	34,595.9	**54**	18	7.4	43,828	34,595.9	55	18	7.4	43,828
RC1_2_2	31,774.7	**112**	**41**	6.5	84,161	31,774.7	117	45	**5.8**	**76,159**
RC1_2_6	32,554.8	121	**20**	21.9	255,845	**32,554.9**	**116**	21	**20.3**	**233,524**
RC1_2_8	**30,216.1**	236	30	**19.3**	517,598	30,215.7	**224**	**29**	19.4	**500,651**
RC2_2_1	27,891.0	1,873	**117**	4.9	4,509	**27,892.7**	**1,761**	126	**4.0**	**3,736**
RC2_2_2	24,588.6	7,141	490	**3.7**	22,238	**24,589.0**	**5,930**	**421**	3.8	**20,558**
RC2_2_5	24,788.5	**3,338**	**263**	4.9	47,954	24,788.5	3,435	306	**4.5**	**42,039**
RC2_2_6	24,724.4	3,484	276	**4.7**	28,380	24,724.4	**3,193**	**242**	5.0	**26,089**
#Best	26/31	8/31	18/31	16/31	8/31	27/31	23/31	16/31	21/31	26/31

Table 27: Detailed results for comparing `Default` vs `Pairwise` settings using `all-arcs` DSSR after adding non-robust cuts

Instance	Default					Pairwise				
	lb	T(s)	#iters	#DSSR	#labels	lb	T(s)	#iters	#DSSR	#labels
C1_210	26,247.0	616	184	8.1	19,180	26,247.0	**571**	**174**	**7.9**	**18,195**
C1_2_3	**26,723.2**	**34,047**	498	**2.0**	**32,563**	26,722.6	45,995	**446**	2.2	36,151
C1_2_4	26,256.0	724	**283**	4.2	34,120	26,256.0	**581**	304	**3.6**	**30,712**
C1_2_9	26,396.0	**185**	**55**	11.1	8,836	26,396.0	187	60	**9.5**	**8,731**
C2_2_1	19,152.8	413	70	1.0	455	19,152.8	**404**	70	1.0	455
C2_2_2	**18,514.0**	12,368	1,415	1.2	5,863	18,471.6	**6,580**	**1,060**	1.2	**4,238**
C2_2_5	18,696.0	2,008	561	1.7	**1,602**	18,696.0	**1,956**	**493**	1.8	1,718
C2_2_6	18,448.0	4,111	**628**	2.3	3,102	18,448.0	**3,730**	638	**2.2**	**2,783**
C2_2_7	18,422.0	**4,968**	737	2.2	4,033	18,422.0	5,067	**704**	2.3	**3,986**
C2_2_8	18,137.0	4,633	577	2.7	5,321	18,137.0	**4,494**	**569**	2.6	**4,814**
C2_2_9	18,150.0	11,608	977	2.2	10,730	18,150.0	**9,256**	**872**	2.2	**10,514**
R1_210	**32,794.9**	2,140	417	2.2	29,849	32,794.1	**1,940**	**388**	2.2	**29,589**
R1_2_4	30,364.6	18,415	**533**	2.0	127,556	**30,366.1**	**18,143**	543	2.0	**123,662**
R1_2_5	40,457.1	427	253	1.1	7,141	40,457.1	**421**	253	1.1	7,141
R1_2_6	**35,456.9**	1,995	461	1.6	29,240	35,450.9	**1,570**	**416**	1.6	**27,229**
R1_2_7	31,392.8	**2,506**	**422**	2.1	**54,672**	**31,393.2**	2,935	462	**2.0**	57,162
R1_2_8	29,328.1	**4,897**	451	2.4	100,373	**29,329.3**	5,023	**443**	2.3	**99,639**
R1_2_9	**37,270.4**	**750**	**308**	1.5	13,958	37,270.1	779	316	1.5	**13,649**
R2_2_1	34,680.0	**1,851**	**230**	1.6	**452**	34,680.0	1,925	325	**1.4**	465
R2_2_2	30,082.0	5,271	594	2.1	**2,213**	**30,086.1**	**4,653**	**516**	2.1	2,287
R2_2_5	**30,609.8**	4,836	1,302	1.3	2,128	30,607.0	**4,341**	**1,182**	1.3	**1,994**
R2_2_6	26,748.4	**13,208**	**1,972**	1.5	**6,390**	**26,750.5**	13,681	2,188	**1.4**	6,464
R2_2_9	**28,433.0**	4,994	911	1.6	2,725	28,424.9	**4,674**	**879**	1.5	**2,625**
RC1_2_1	34,993.5	**634**	312	1.4	10,372	34,993.5	641	312	1.4	10,372
RC1_2_2	**32,072.6**	28,285	495	1.8	41,646	32,070.7	**21,917**	**488**	1.8	**39,992**
RC1_2_6	32,938.2	5,123	**503**	2.7	36,610	**32,938.8**	**4,720**	519	2.6	**36,117**
RC1_2_8	30,499.5	**59,534**	**543**	3.1	**133,139**	**30,502.3**	81,561	552	3.1	147,774
RC2_2_1	**27,966.5**	2,242	682	1.7	**909**	27,954.3	**2,025**	**538**	1.8	948
RC2_2_2	**24,780.0**	31,302	4,948	1.3	6,565	24,779.7	**25,767**	**4,458**	1.3	**6,430**
RC2_2_5	24,914.0	**7,532**	**1,817**	1.6	14,139	24,914.0	8,040	2,278	**1.5**	**11,975**
RC2_2_6	24,951.0	7,288	2,026	1.6	8,280	24,951.0	**6,701**	**1,994**	1.5	**7,296**
#Best	24/31	11/31	14/31	23/31	11/31	21/31	20/31	20/31	30/31	23/31

Table 28: Detailed results for comparing `Default` vs `SetPair` settings using `all-arcs` DSSR before adding non-robust cuts

Instance	Default					SetPair				
	lb	**T(s)**	**#iters**	**#DSSR**	**#labels**	**lb**	**T(s)**	**#iters**	**#DSSR**	**#labels**
C1_2_10	26,197.3	477	74	**16.1**	**310,753**	26,197.2	**383**	**56**	18.2	324,638
C1_2_3	26,597.4	323	**113**	4.1	73,609	26,597.4	**275**	127	**3.9**	**64,213**
C1_2_4	26,197.7	525	**157**	6.2	197,897	26,197.7	**468**	166	**5.1**	**158,242**
C1_2_9	26,396.0	**184**	55	11.1	97,994	26,396.0	190	**48**	**10.9**	**94,478**
C2_2_1	19,150.3	**359**	8	1.0	388	19,150.3	361	8	1.0	388
C2_2_2	18,394.0	**3,905**	**518**	1.4	6,134	**18,396.0**	4,507	562	1.4	**5,752**
C2_2_5	18,607.1	1,259	**151**	3.3	3,707	18,607.1	**1,229**	154	**2.9**	**3,209**
C2_2_6	18,406.7	3,079	252	**4.1**	**11,771**	18,406.7	**2,680**	**209**	4.4	11,894
C2_2_7	18,356.4	3,905	360	**3.4**	13,757	18,356.4	**3,495**	**323**	3.6	**13,573**
C2_2_8	18,082.4	3,564	232	**5.0**	**23,873**	18,082.4	**3,144**	**169**	6.2	27,415
C2_2_9	**18,011.9**	6,020	306	4.5	35,977	18,010.9	**4,818**	**289**	**4.1**	**29,724**
R1_2_10	32,451.3	132	**21**	14.2	183,530	32,451.2	**121**	24	**12.0**	**150,794**
R1_2_4	**30,010.5**	198	**36**	8.7	238,385	30,010.3	**161**	40	**7.6**	**201,241**
R1_2_5	40,068.1	43	12	2.0	9,881	40,068.1	**42**	12	2.0	**9,874**
R1_2_6	34,930.6	93	44	**3.5**	**39,224**	34,930.6	**88**	**36**	3.9	46,483
R1_2_7	31,001.0	189	**51**	5.6	117,367	31,001.0	**168**	52	**5.1**	**106,912**
R1_2_8	29,032.3	250	35	**10.2**	302,465	29,032.4	**202**	**32**	10.5	**297,856**
R1_2_9	**36,788.5**	68	**18**	6.4	46,453	36,788.3	**64**	19	**5.9**	**42,799**
R2_2_1	34,625.1	1,751	106	2.3	1,060	34,625.1	**1,668**	**105**	2.3	**1,032**
R2_2_2	30,006.1	4,391	291	**3.2**	**7,647**	30,006.1	**4,028**	**239**	3.6	8,582
R2_2_5	30,290.3	2,681	146	3.4	4,062	30,290.3	**2,514**	**137**	3.4	**3,998**
R2_2_6	26,455.5	6,253	**278**	4.3	28,234	26,455.6	**5,772**	303	**3.6**	**21,723**
R2_2_9	28,206.0	3,616	205	**3.4**	8,047	28,206.0	**3,352**	**163**	3.6	**7,994**
RC1_2_1	34,595.9	54	18	**7.4**	**43,828**	34,595.9	**52**	**17**	7.8	45,483
RC1_2_2	31,774.7	112	41	6.5	**84,161**	31,774.7	**108**	41	6.4	86,447
RC1_2_6	32,554.8	121	20	**21.9**	**255,845**	32,554.8	**108**	**19**	22.2	257,405
RC1_2_8	**30,216.1**	236	30	19.3	517,598	30,215.7	**195**	30	19.2	**499,774**
RC2_2_1	27,891.0	1,873	117	**4.9**	**4,509**	27,891.0	**1,864**	**77**	6.3	5,872
RC2_2_2	24,588.6	7,141	490	3.7	22,238	**24,588.7**	**6,172**	**428**	3.7	**19,722**
RC2_2_5	24,788.5	3,338	263	**4.9**	**47,954**	24,788.5	**3,297**	**242**	5.3	50,913
RC2_2_6	24,724.4	3,484	276	4.7	28,380	24,724.4	**3,281**	**259**	4.8	**24,910**
#Best	29/31	3/31	13/31	21/31	11/31	27/31	28/31	22/31	19/31	21/31

Table 29: Detailed results for comparing `Default` vs `SetPair` settings using `all-arcs` DSSR after adding non-robust cuts

Instance	Default					SetPair				
	lb	**T(s)**	**#iters**	**#DSSR**	**#labels**	**lb**	**T(s)**	**#iters**	**#DSSR**	**#labels**
C1_210	26,247.0	616	184	**8.1**	19,180	26,247.0	**518**	**165**	8.4	**18,126**
C1_2_3	**26,723.2**	**34,047**	498	**2.0**	**32,563**	26,721.7	95,535	**458**	2.2	33,154
C1_2_4	26,256.0	724	**283**	4.2	34,120	26,256.0	**626**	298	**3.6**	**32,310**
C1_2_9	26,396.0	**185**	55	11.1	8,836	26,396.0	191	**48**	**10.9**	**8,671**
C2_2_1	19,152.8	**413**	**70**	1.0	**455**	**19,156.4**	434	98	1.0	475
C2_2_2	**18,514.0**	12,368	1,415	1.2	5,863	18,495.7	**7,666**	**1,285**	1.2	**4,424**
C2_2_5	18,696.0	2,008	561	1.7	**1,602**	18,696.0	**1,913**	**523**	1.6	1,635
C2_2_6	18,448.0	4,111	628	**2.3**	3,102	18,448.0	**3,372**	**498**	2.5	**2,756**
C2_2_7	18,422.0	4,968	737	**2.2**	4,033	18,422.0	**4,234**	**600**	2.4	**3,850**
C2_2_8	18,137.0	4,633	577	2.7	5,321	18,137.0	**4,282**	**561**	2.6	**4,752**
C2_2_9	18,150.0	11,608	977	2.2	**10,730**	18,150.0	**10,600**	**837**	2.3	11,053
R1_210	**32,794.9**	2,140	417	2.2	29,849	32,793.1	**1,923**	**399**	2.2	**28,930**
R1_2_4	30,364.6	**18,415**	533	2.0	**127,556**	**30,367.1**	28,779	**526**	2.0	130,872
R1_2_5	**40,457.1**	427	253	1.1	7,141	40,451.6	**331**	**221**	1.1	**6,848**
R1_2_6	**35,456.9**	1,995	461	1.6	**29,240**	35,455.3	**1,835**	**426**	1.6	29,301
R1_2_7	31,392.8	**2,506**	**422**	2.1	**54,672**	**31,393.6**	3,797	442	**2.0**	55,468
R1_2_8	29,328.1	**4,897**	451	2.4	100,373	**29,328.5**	5,961	**447**	2.3	**99,820**
R1_2_9	**37,270.4**	750	308	1.5	13,958	37,267.6	**691**	**294**	1.6	**13,053**
R2_2_1	34,680.0	1,851	**230**	1.6	**452**	34,680.0	**1,794**	282	**1.5**	456
R2_2_2	30,082.0	5,271	594	**2.1**	**2,213**	30,082.0	**4,735**	**508**	2.2	2,229
R2_2_5	30,609.8	**4,836**	**1,302**	1.3	**2,128**	**30,611.0**	4,998	1,343	1.3	2,233
R2_2_6	26,748.4	**13,208**	**1,972**	1.5	6,390	**26,750.5**	14,960	2,536	**1.3**	**6,206**
R2_2_9	28,433.0	4,994	911	1.6	2,725	28,433.0	**4,747**	**847**	1.6	**2,583**
RC1_2_1	34,993.5	**634**	**312**	1.4	10,372	**34,994.6**	658	318	1.4	**10,135**
RC1_2_2	32,072.6	**28,285**	**495**	1.8	**41,646**	32,072.5	43,875	497	1.8	42,617
RC1_2_6	32,938.2	**5,123**	**503**	2.7	**36,610**	**32,941.2**	6,784	544	**2.5**	38,856
RC1_2_8	**30,499.5**	59,534	543	3.1	133,139	30,497.7	**57,887**	**534**	3.1	**130,149**
RC2_2_1	**27,966.5**	2,242	682	1.7	**909**	27,961.8	**2,194**	**571**	1.8	926
RC2_2_2	24,780.0	31,302	4,948	1.3	**6,565**	24,780.0	**25,755**	**4,177**	1.3	6,745
RC2_2_5	24,914.0	**7,532**	**1,817**	1.6	14,139	24,914.0	7,993	2,053	1.5	**13,719**
RC2_2_6	24,951.0	7,288	2,026	1.6	8,280	24,951.0	**6,431**	**1,920**	1.5	**7,976**
#Best	23/31	12/31	10/31	25/31	14/31	23/31	19/31	21/31	26/31	17/31

References

Baldacci, R., Mingozzi, A., Roberti, R., 2011. New route relaxation and pricing strategies for the vehicle routing problem. Operations Research 59, 1269–1283.

Boland, N., Dethridge, J., Dumitrescu, I., 2006. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. Operations Research Letters 34, 58–68.

Bulhões, T., Sadykov, R., Uchoa, E., 2018. A branch-and-price algorithm for the minimum latency problem. Computers & Operations Research 93, 66–78.

Contardo, C., Desaulniers, G., Lessard, F., 2015. Reaching the elementary lower bound in the vehicle routing problem with time windows. Networks 65, 88–99.

Costa, L., Contardo, C., Desaulniers, G., 2019. Exact branch-price-and-cut algorithms for vehicle routing. Transportation Science 53, 946–985.

Dantzig, G.B., Fulkerson, D.R., Johnson, S.M., 1954. Solution of a large-scale traveling salesman problem. Operations Research 2, 393–410.

Dantzig, G.B., Ramser, J.H., 1959. The Truck Dispatching Problem. Management Science 6, 80–91.

Desaulniers, G., Lessard, F., Hadjar, A., 2008. Tabu search, partial elementarity, and generalized k-path inequalities for the vehicle routing problem with time windows. Transportation Science 42, 387–404.

Desaulniers, G., Madsen, O.B.G., Ropke, S., 2014. The vehicle routing problem with time windows, in: Toth, P., Vigo, D. (Eds.), Vehicle Routing: Problems, Methods and Applications. 2nd ed.. Society for Industrial and Applied Mathematics, Philadelphia, PA. chapter 5, pp. 119–159.

Desaulniers, G., Pecin, D., Contardo, C., 2019. Selective pricing in branch-price-and-cut algorithms for vehicle routing. EURO Journal on Transportation and Logistics 8, 147–168.

Desrochers, M., Desrosiers, J., Solomon, M., 1992. A new optimization algorithm for the vehicle routing problem with time windows. Operations Research 40, 342–354.

Desrochers, M., Soumis, F., 1988. A generalized permanent labelling algorithm for the shortest path problem with time windows. INFOR: Information Systems and Operational Research 26, 191–212.

Desrosiers, J., Soumis, F., Desrochers, M., 1984. Routing with time windows by column generation. Networks 14, 545–565.

Dolan, E.D., More, J.J., 2002. Benchmarking optimization software with performance profiles. Mathematical Programming A 91, 201–213.

Dror, M., 1994. Note on the complexity of the shortest path models for column generation in vrptw. Operations Research 42, 977–978.

Gehring, H., Homberger, J., 2001. A parallel two-phase metaheuristic for routing problems with time windows. Asia-Pacific Journal of Operational Research 18, 35.

Irnich, S., 2008. Resource extension functions: Properties, inversion and generalization to segments. OR Spectrum 30, 113–148.

Irnich, S., Desaulniers, G., 2005. Shortest path problems with resource constraints, in: Desaulniers, G., Desrosiers, J., Solomon, M.M. (Eds.), Column Generation. 1st ed.. Springer US, Boston, MA. chapter 2, pp. 33–65.

Irnich, S., Desaulniers, G., Desrosiers, J., Hadjar, A., 2010. Path-reduced costs for eliminating arcs in routing and scheduling. INFORMS Journal on Computing 22, 297–313.

Irnich, S., Villeneuve, D., 2006. The shortest path problem with resource constraints and k-cycle elimination for $k \geq 3$. INFORMS Journal on Computing 18, 391–406.

Laporte, G., Nobert, Y., Desrochers, M., 1985. Optimal routing under capacity and distance restrictions. Operations Research 33, 1050–1073.

Martinelli, R., Pecin, D., Poggi, M., 2014. Efficient elementary and restricted non-elementary route pricing. European Journal of Operational Research 239, 102–111.

Pecin, D., Contardo, C., Desaulniers, G., Uchoa, E., 2017a. New enhancements for the exact solution of the vehicle routing problem with time windows. INFORMS Journal on Computing 29, 489–502.

Pecin, D., Pessoa, A., Poggi, M., Uchoa, E., 2017b. Improved branch-cut-and-price for capacitated vehicle routing. Mathematical Programming Computation 9, 61–100.

Poggi, M., Uchoa, E., 2014. New exact algorithms for the capacitated vehicle routing problem, in: Toth, P., Vigo, D. (Eds.), Vehicle Routing: Problems, Methods and Applications. 2nd ed.. Society for Industrial and Applied Mathematics, Philadelphia, PA. chapter 3, pp. 59–86.

Righini, G., Salani, M., 2006. Symmetry helps: bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. Discrete Optimization 3, 255–273.

Sadykov, R., Uchoa, E., Pessoa, A.A., 2017. A Bucket Graph Based Labeling Algorithm with Application to Vehicle Routing. Research Report Cadernos do LOGIS 2017/7. Universidade Federal Fluminense.

Solomon, M.M., 1987. Algorithms for the vehicle routing and scheduling problems with time window constraints. Operations Research 35, 254–265.

Toth, P., Vigo, D. (Eds.), 2014. Vehicle routing: problems, methods and applications. MOS/SIAM Series on Optimization.

Vidal, T., Laporte, G., Matl, P., 2019. A concise guide to existing and emerging vehicle routing problem variants. European Journal of Operational Research, forthcoming, 1–42.