CADERNOS DO LOGIS

Volume 2017, Number 4

A branch-and-price algorithm for the Minimum Latency Problem

Teobaldo Bulhões, Ruslan Sadykov, Eduardo Uchoa

July, 2017

A branch-and-price algorithm for the Minimum Latency Problem

Teobaldo Bulhões[a], Ruslan Sadykov[b], Eduardo Uchoa[c,*]

[a] *Instituto de Computação, Universidade Federal Fluminense, Brazil*
[b] *Inria Bordeaux - Sud Ouest, France*
[c] *Departamento de Engenharia de Produção, Universidade Federal Fluminense, Brazil*

Abstract

This paper deals with the Minimum Latency Problem (MLP), a variant of the well-known Traveling Salesman Problem in which the objective is to minimize the sum of waiting times of customers. This problem arises in many applications where customer satisfaction is more important than the total time spent by the server. This paper presents a novel branch-and-price algorithm for MLP that strongly relies on new features for the *ng*-path relaxation, namely: (1) a new labeling algorithm with an enhanced dominance rule named multiple partial label dominance; (2) a generalized definition of *ng*-sets in terms of arcs, instead of nodes; and (3) a strategy for decreasing *ng*-set size when those sets are being dynamically chosen. Also, other elements of efficient exact algorithms for vehicle routing problems are incorporated into our method, such as reduced cost fixing, dual stabilization, route enumeration and strong branching. Computational experiments over TSPLIB instances are reported, showing that several instances not solved by the current state-of-the-art method can now be solved.

Keywords: minimum latency, *ng*-paths, branch-and-price

1. Introduction

This paper deals with the Minimum Latency Problem (MLP). In MLP, we are given a complete directed graph $G = (V, A)$ and a time t_{ij} for each arc $(i, j) \in A$. Set V is composed of $n + 1$ nodes: node 0, representing a depot, and n nodes representing customers. The task is to find a Hamiltonian circuit $(i_0 = 0, i_1, \ldots, i_n, i_{n+1} = 0)$, a.k.a tour, in G that minimizes $\sum_{t=1}^{n+1} l(i_t)$, where the latency $l(i_t)$ is defined as the accumulated travel time from the depot to i_t. The MLP is related to the Time Dependent Traveling Salesman Problem (TDTSP), a generalization of the Traveling Salesman Problem (TSP) in which the cost for traversing an arc depends on its position in the tour. More precisely, MLP can be viewed as the particular case of the TDTSP where the cost of an arc (i, j) in position $p, 0 \leq p \leq n$, is given by $(n - p + 1)t_{ij}$.

*Corresponding author
Email addresses: tbulhoes@ic.uff.br (Teobaldo Bulhões), Ruslan.Sadykov@inria.fr (Ruslan Sadykov), uchoa@producao.uff.br (Eduardo Uchoa)

The MLP is also known in the literature as Delivery Man Problem (Roberti and Mingozzi, 2014), Traveling Repairman Problem (Afrati, Foto et al., 1986), Traveling Deliveryman Problem (Tsitsiklis, 1992) and Traveling Salesman with Cumulative Costs (Bianco et al., 1993). Although MLP seems to be a simple variant of TSP, some important characteristics are very different in those problems. First, two different viewpoints of a distribution system are considered: TSP is server oriented, since one wants to minimize the total travel time; on the other hand, MLP is customer oriented because the objective is equivalent to minimizing the average waiting time of customers (Silva et al., 2012; Sitters, 2002; Archer and Williamson, 2003). Customer satisfaction is the main objective in many applications, such as home delivery services (Méndez-Díaz et al., 2008), and has attracted the attention of researchers, as reflected by the considerable number of MLP variants studied in the very last years (see, for instance, (Lysgaard and Wøhlk, 2014; Rivera et al., 2016; Nucamendi-Guillén et al., 2016; Sze et al., 2017)). Second, in contrast to what happens in TSP, simple local changes may affect globally a MLP solution because the latency of subsequent customers may change (Silva et al., 2012; Sitters, 2002). This can make it more difficult to solve MLP both exactly and heuristically. For example, current state-of-the-art exact methods for MLP are not capable of solving consistently instances with 150 customers, whereas TSP instances with thousands of customers are solved routinely (Abeledo et al., 2013).

Many complexity results for MLP have been obtained. The problem is NP-Hard for general metric spaces (Sahni and Gonzalez, 1976), and remains NP-Hard even if the times correspond to euclidean distances (Afrati, Foto et al., 1986) or if they are obtained from an underlying graph that is a tree Sitters (2002). On the other hand, the problem is polynomial if the underlying graph is a path (Afrati, Foto et al., 1986; Garca et al., 2002), a tree with equal weights or a tree with diameter at most 3 (Blum et al., 1994). The MLP with deadlines, i.e., with upper bounds on $l(i_t)$, is NP-Hard even for paths (Afrati, Foto et al., 1986). In terms of approximation, hardness results show that one should not expect to attain arbitrarily good approximation factors for MLP (Blum et al., 1994). However, 3.59 and 3.03 approximations are known for general metric spaces and general trees, respectively (Chaudhuri et al., 2003; Archer and Blasiak, 2010). Moreover, a constant factor approximation is not likely to exist if times do not satisfy the triangle inequality, just as for TSP (Blum et al., 1994).

The first integer programming formulations were given in (Picard and Queyranne, 1978), where the authors stated TDTSP as a machine scheduling problem and solved instances with up to 20 jobs by means of a branch-and-bound method over lagrangian bounds. A new formulation with n constraints was presented in (Fox et al., 1980), but the authors did not report any computational results. Lucena (1990) and Bianco et al. (1993) followed the same approach as Picard and Queyranne (1978) and employed langragian bounds in experiments over MLP instances with up to 30 and 60 vertices, respectively. The latter authors also developed a dynamic programming method capable of attesting that the bounds obtained for 60-vertex instances were within 3% from optimality. Then, a series of enumerative strategies based on new formulations

was introduced in (Fischetti et al., 1993; Van Eijl, 1995; Méndez-Díaz et al., 2008; Bigras et al., 2008; Godinho et al., 2014), as well as cutting planes (Van Eijl, 1995; Méndez-Díaz et al., 2008; Bigras et al., 2008) and polyhedral studies (Méndez-Díaz et al., 2008). Instances with 60 vertices could already be solved by the algorithm of Fischetti et al. (1993). More recently, Abeledo et al. (2013) managed to solve almost all TSPLIB instances with up to 107 vertices using a branch-cut-and-price algorithm. The authors departed from a formulation by Picard and Queyranne (1978) and proposed new inequalities, that are proved to be facet-inducing. Roberti and Mingozzi (2014) implemented dual ascent and column generation techniques to compute a sequence of lower bounds associated with set partitioning formulations where a column represents a *ng*-path, which is a path relaxation introduced by Baldacci et al. (2011). A *ng*-path may contain cycles, but just those allowed by the so-called *ng*-sets. Such sets are iteratively augmented so that less cycles are allowed and improved bounds are obtained. The final lower bound is used in a dynamic programming recursion to compute the optimal solution. This method could solve some larger TSPLIB instances, with up to 150 vertices, and currently holds the status of state-of-the-art exact method for MLP. Finally, heuristic algorithms for MLP can be found in (Ngueveu et al., 2010; Salehipour et al., 2011; Silva et al., 2012; Mladenović et al., 2013).

This paper presents a novel branch-and-price algorithm for MLP that strongly relies on *ng*-paths. Following the directions of Roberti and Mingozzi (2014), our method works over a set partitioning formulation where columns represent *ng*-paths and the column generation bounds computed on each node of the tree are derived from dynamically defined *ng*-sets. However, we introduce the following improvements on the use of *ng*-paths.

- **Multiple Partial Label Dominance:** In the labeling algorithms used for pricing *ng*-paths, a partial path P is represented as a label $L(P)$. A key concept in this kind of algorithm is of dominance. A label $L(P_1)$ dominates a label $L(P_2)$ if the cost of P_1 is not larger than the cost of P_2 and every completion of P_2 is also a feasible completion of P_1. In this case, $L(P_2)$ can be safely eliminated. In this paper, we propose a stronger dominance rule by which some extensions for $L(P_2)$ can be avoided, even though this label can not be completely disregarded according to the classical dominance rule. We briefly discuss two alternative implementations of this new dominance rule, where the best one typically speeds the labeling algorithm by factors between 4 and 8.
- **Arc-Based *ng*-Path Relaxation:** *ng*-sets as originally defined by Baldacci et al. (2011) are a vertex-based memory mechanism. In this paper, we provide a generalized definition of them in terms of arcs. We show that this new definition is particularly useful in the context of dynamically defined *ng*-sets, allowing strong bounds to be obtained in more controlled pricing times.
- **Fully Dynamic *ng*-Path Relaxation:** We improve the dynamic *ng*-path relaxation of Roberti and Mingozzi (2014) by introducing a procedure for decreasing the *ng*-sets, without

changing the current bounds. Such reductions are beneficial for the pricing time and also help to refine the choice of *ng*-sets.

Also, other well-known elements of efficient exact algorithms for many other variants of the vehicle routing problem (VRP) are incorporated into our method, namely reduced cost fixing, dual stabilization, route enumeration and strong branching. Computational experiments over MLP instances derived from TSPLIB were conducted to attest the effectiveness of the new branch-and-price algorithm. The results show that better bounds can be obtained in less computational time when compared to the state-of-the-art algorithm, specially because of the new features for the *ng*-path relaxation. In particular, the branch-and-price solved all the 9 instances with up to 150 vertices not solved in Roberti and Mingozzi (2014). It could also solve 4 additional instances, with more than 150 vertices, never considered before by exact methods.

The remainder of this paper is organized as follows. Section 2 discusses the *ng*-path relaxation and labeling algorithms. Section 3 introduces the new features for the *ng*-path relaxation. The proposed branch-and-price algorithm is described in Section 4, where we also give implementations details. Computational experiments are presented in Section 5. Finally, concluding remarks are drawn in the last section.

2. Route Relaxations and Labeling Algorithms

This section reviews the route relaxations and labeling algorithms that are related to current state-of-the-art exact algorithms for VRPs, such as Capacitated VRP (CVRP), VRP with time windows (VRPTW), and the MLP itself. Such algorithms are based on a combination of column and cut generation over the following set-partitioning formulation.

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

$$\text{s.t.} \quad \sum_{r \in \Omega} a_r^i \lambda_r = 1, \qquad \forall i \in \mathcal{C}, \tag{2}$$

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

where $\mathcal{C}$, Ω, c_r and a_r^i denote, respectively, the set of customers, the set of feasible routes, the cost of route r, and the number of times route r visits customer i.

As the number of variables in Formulation (1)-(3) is exponential on $|\mathcal{C}|$, column generation is typically applied to solve its linear relaxation. The pricing subproblem depends on the considered variant, but it can often be modeled as the Elementary Resource Constrained Shortest Path Problem (ERCSPP). In ERCSPP, we are given a directed graph $G' = (V', A')$ with vertex set V' and arc set A'; source and sink nodes $s \in V'$ and $t \in V'$, respectively; and a set of resources $\mathcal{R}$. Each arc (i, j) has an associated cost $c_{ij} \in \mathbb{R}$ and consumes a predefined amount $w_{ij}^r \in \mathbb{R}_{>0}$ of resource r, for each $r \in \mathcal{R}$. Moreover, for each $i \in V'$ and $r \in \mathcal{R}$, let l_i^r and u_i^r be, respectively,

the minimum and the maximum consumption of resource r in any partial path from s to i. The task is to find the least-cost path from s to t satisfying the resource constraints and containing no cycles. Note that, in order to satisfy minimum consumption, it is possible to "drop resources" at no cost. See (Di Puglia Pugliese and Guerriero, 2012) for further details on ERCSPP.

From now on, for ease of presentation, we assume the case related to MLP, where a single discrete resource is present. The consumption of this single resource by an arc (i, j) will be denoted as $w_{ij} \in \mathbb{Z}_{>0}$ and the lower and upper bounds on its consumption when reaching vertex i as $l_i \in \mathbb{Z}_{\geq 0}$ and $u_i \in \mathbb{Z}_{\geq 0}$. In the graph G' for MLP, nodes in $\{1, \ldots, n\}$ represent customers, while source and sink nodes are associated with the depot. The single resource indicates the number of arcs in a partial path, and hence $w_{ij} = 1$ for any arc $(i, j) \in A'$. Finally, $(l_i, u_i) = (1, n)$ for a vertex $i \in \{1, \ldots, n\}$; $(l_s, u_s) = (0, 0)$ and $(l_t, u_t) = (n + 1, n + 1)$.

ERCSPP is NP-Hard in the strong sense (Dror, 1994) and also a difficult problem to solve in practice. The "offending" constraint is the one that imposes elementarity, since the Resource Constrained Shortest Path Problem (RCSPP) can be solved in pseudo-polynomial time. RCSPP is a relaxation of ERCSPP where a vertex can be visited more than once in an optimal path, as long as the resource constraints are satisfied. In view of this, several state-of-the-art algorithms employ some route relaxation as an alternative to elementary routes. The idea is to replace set Ω by some set also containing non-elementary routes so as to make the pricing subproblem easier. An ideal relaxation would provide the elementary route bound while keeping the pricing subproblem tractable. It is worth mentioning that such a relaxation is kind of mandatory in problems where the optimal solution has exactly one route (e.g., MLP), otherwise the pricing subproblem is as hard as the original problem, rendering column generation meaningless.

The first route relaxation is just to allow any non-elementary route, as long as it satisfies the resource constraints. As the elementarity is relaxed, the pricing corresponds to a RCSPP, which can be solved in pseudo-polynomial time. It was already observed in Christofides et al. (1981) that it is possible to eliminate routes with 2-cycles (subpaths like $i \rightarrow j \rightarrow i$) without increasing the complexity. The bounds obtained with 2-cycle elimination may be good in some cases, but are likely to be poor in other cases, specially in routing problems with many customers per route (Martinelli et al., 2014). This motivated Irnich and Villeneuve (2006) to propose an algorithm to forbid cycles of an arbitrary maximum size k. The pricing subproblem now is referred to as RCSPP with k-cycle elimination. Theoretically, the proposed algorithm can be used for pricing elementary routes, but only small values of k can be efficiently used in practice. This is because the complexity of the algorithm grows by a factor of up to $k^2.k!$. Nevertheless, k-cycle elimination for small values of k proved to be useful at that time, improving column generation based algorithms for several VRP variants. For example, the branch-cut-and-price for MLP in (Abeledo et al., 2013) uses $k = 5$.

Later, Baldacci et al. (2011) introduced a new kind of elementarity relaxation, the so-called *ng*-routes. Extensive experiments on several VRP variants show that *ng*-routes are almost al-

ways more efficient than routes without k-cycles, in the sense of providing better bounds in less computational time. In contrast to previous relaxations, cycles eliminated by *ng*-routes are not distinguished by size. Instead, a cycle $H = (i_0, i_1, \ldots, i_p = i_0)$ is forbidden if all vertices $i_1, \ldots, i_{p-1}$ are able to "remember" vertex i_0. Formally speaking, we define an *ng*-set N_i for each vertex i. We assume that $i \in N_i$. Typically, N_i contains the vertices that are likely to appear close to vertex i in low-cost paths, e.g., nearest customers in VRPs to take advantage of a locality principle that is often present in those problems. In case $j \in N_i$, we say that i remembers j or equivalently that j is remembered by i. Then, each path $P = (s = i_0, i_1, \ldots, i_p)$ has an associated set of forbidden extensions $\Pi(P)$ that is computed based on the *ng*-sets:

$$\Pi(P) = \left\{ i_k \in \mathcal{C}(P) \setminus \{i_p\} : i_k \in \bigcap_{s=k+1}^{p} N_{i_s} \right\} \cup \{i_p\} \tag{4}$$

where $\mathcal{C}(P)$ is the set of vertices visited by P. In words, a vertex $i_k \neq i_p$ belongs to $\Pi(P)$ if it is remembered by all vertices i_s with $k < s \leq p$. Therefore, path P is *ng*-feasible if $i_k \notin \Pi(P_{k-1} = (i_0, i_1, \ldots, i_{k-1})), 1 \leq k \leq p$. Alternatively, one can define the *ng*-path relaxation in terms of the *ng*-memory M_i of a vertex i, which is the set of vertices that remember i, i.e., $M_i = \{j \in V : i \in N_j\}$. In this case, the set of forbidden extensions for path P is computed as in Equation (5). Moreover, path P is *ng*-feasible iff between two visits to a vertex i, some vertex $j \notin M_i$ is visited. In this paper, both concepts (*ng*-sets and *ng*-memories) are used. Clearly, the greater the *ng*-memories, the closer to elementary are the *ng*-paths.

$$\Pi(P) = \left\{ i_k \in \mathcal{C}(P) \setminus \{i_p\} : i_s \in M_{i_k}, s = k+1, \ldots, p \right\} \cup \{i_p\} \tag{5}$$

The RCSPP with *ng*-routes is commonly solved by means of a labeling algorithm. In this kind of algorithm, a label $L(P) = (c(P), w(P), v(P), \Pi(P))$ represents a partial path P ending at vertex $v(P)$ with cost $c(P)$, resource consumption $w(P)$ (we are assuming a single-resource), and set $\Pi(P)$ of forbidden extensions. A single label representing the path defined only by the source vertex s is present in the beginning, and new labels are generated by extending existing ones along all possible arcs. In order to the algorithm to be correct, it is sufficient to process existing labels in increasing order of resource consumption. Dominance rules are applied to eliminate labels not leading to optimal paths, where a label $L(P_1)$ dominates a label $L(P_2)$ if the cost of P_1 is less than or equal to the cost of P_2 and any feasible completion to P_2 into a feasible $s-t$ path is also feasible to P_1. The general scheme of labeling algorithms with *ng*-paths is shown in Algorithm 1, where L_j and U_j denote, respectively, the set of processed and unprocessed labels that correspond to paths ending at vertex j.

The feasibility test in Line 5 of Algorithm 1 takes into account the resource constraints and the extensions forbidden by *ng*-sets. More precisely, the new path $P+k$ is feasible if $l_k \leq w(P)+w_{ik} \leq u_k$ and $k \notin \Pi(P)$. Regarding the dominance checks performed in lines 7 and 10, the following

Algorithm 1 General Labeling Algorithm with *ng*-paths

Initialize L_j and U_j as empty sets, for all $j \in V$
Add the initial label to U_s
while $\bigcup_{j \in V} U_j \neq \emptyset$ **do**
  Choose a label $L(P) \in \bigcup_{j \in V} U_j$ with minimum resource consumption and let $i = v(P)$
  for each $(i, k) \in A$ such that $P + k$ is feasible **do**
    Define label $L(P + k) = (c(P) + c_{ik}, w(P) + w_{ik}, k, (\Pi(P) \cap N_k) \cup \{k\})$
    if $L(P + k)$ is dominated by any label in $L_k \cup U_k$ **then**
      continue
    else
      Remove labels in U_k dominated by $L(P + k)$
      $U_k \leftarrow U_k \cup \{L(P + k)\}$
  $U_i \leftarrow U_i \setminus \{L(P)\}$
  $L_i \leftarrow L_i \cup \{L(P)\}$
return best label in L_t

conditions are sufficient and necessary to verify that a label $L(P')$ dominates a label $L(P)$.

(I) $v(P') = v(P)$
(II) $c(P') \leq c(P)$
(III) $w(P') \leq w(P)$
(IV) $\Pi(P') \subseteq \Pi(P)$

3. New Features for the *ng*-Path Relaxation

This section presents new contributions on the use of the *ng*-path relaxation. Such contributions are not built over any strong assumption regarding MLP and can be used in several other problems.

3.1. Multiple Partial Label Dominance

We will now introduce a stronger dominance rule called *multiple partial label dominance* (MPLD). Suppose that condition (IV) of the classical dominance rule discussed in Section 2 is the only one not satisfied by P and P'. Therefore, label $L(P)$ is not dominated by label $L(P')$ because there exists a vertex i such that $P + i$ is *ng*-feasible, whereas $P' + i$ is not. While completely disregarding label $L(P)$ because of label $L(P')$ is not correct, some extensions for the former may be unnecessary. That is, label $L(P)$ may be *partially dominated* by label $L(P')$.

Proposition 1. *Let $L(P)$ and $L(P')$ be two labels such that*

- $v(P') = v(P) = i$
- $c(P') \leq c(P)$

- $w(P') \leq w(P)$
- $\Pi(P') \not\subseteq \Pi(P)$

Then any extension to a vertex $j \notin \bigcup_{k\in\Pi'} M_k$ *can be safely disregarded for* $L(P)$*, where* $\Pi' = \Pi(P') \setminus \Pi(P)$.

Proof. Let j be a vertex such that $(i,j) \in A$ and $j \notin \bigcup_{k\in\Pi'} M_k$. Since we have assumed that $l \in M_l$ for every $l \in V$, we have that $j \notin \Pi'$. Hence, if $P + j$ is *ng*-feasible, so is $P' + j$ and clearly $v(P' + j) = v(P + j)$, $c(P' + j) \leq c(P + j)$ and $w(P' + j) \leq w(P + j)$. Furthermore, as $j \notin \bigcup_{k\in\Pi'} M_k$, any vertex $k \in \Pi'$ is forgotten by $P' + j$ and $P + j$, and thus $\Pi(P' + j) \subseteq \Pi(P + j)$. Therefore, label $L(P' + j)$ dominates label $L(P + j)$. □

In view of Proposition 1, a label $L(P)$ should also store a set $\xi(P)$ representing the extensions that can be avoided because of partial dominance, which we call *dominated extensions*. Let $\mathcal{L}(P)$ be the set of all labels $L(P')$ that together with label $L(P)$ satisfy conditions (I), (II) and (III). By applying the partial dominance from the multiple labels in $\mathcal{L}(P)$, the resulting set of dominated extensions for label $L(P)$ is:

$$\xi(P) = \bigcup_{L(P')\in\mathcal{L}(P)} \left\{ j \in V : j \notin \bigcup_{\substack{k\in\Pi(P') \\ k\notin\Pi(P)}} M_k \right\} \tag{6}$$

A small example of MPLD is presented in Figure 1. In this example, we have six nodes (besides source and sink nodes) and all labels ending at vertex 1 are depicted in the figure. Sets of dominated extensions for these labels are defined as $\xi(P_1) = \emptyset$, $\xi(P_2) = V \setminus M_2$, $\xi(P_3) = (V \setminus M_5) \cup (V \setminus (M_2 \cup M_5))$ and $\xi(P_4) = (V \setminus M_5) \cup (V \setminus M_3)$. All extensions for label $L(P_4)$ are either forbidden because of *ng*-sets or dominated because of labels $L(P_1)$, $L(P_2)$ and $L(P_3)$, therefore $L(P_4)$ can be removed. This illustrates a situation where MPLD results in complete dominance.

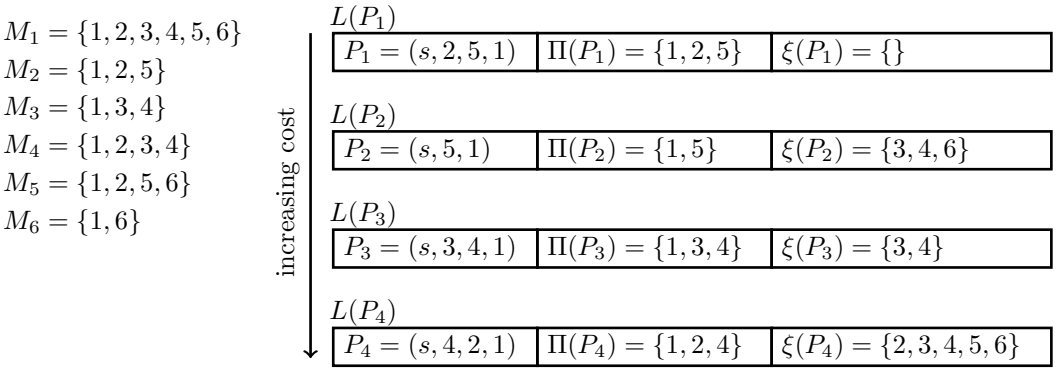

Figure 1: An example of MPLD. Left: *ng*-memories. Right: labels representing paths ending at vertex 1, all of them with the same resource consumption.

3.2. Arc-Based ng-Path Relaxation

The main idea exploited by the *ng*-path relaxation is that, in many problems, cycles are often confined to small "neighborhoods" of the graph: once a new visit to a node i is performed by a partial path P, any node j that is not a neighbor of i is forgotten. The set of nodes remembered by P is computed as a function of *ng*-memories defined over the set of vertices V. In this section, we show a generalized definition of the *ng*-path relaxation where *ng*-memories are defined in terms of arcs instead of nodes. The inspiration for this new definition comes from the arc-based limited memory technique developed by Pecin et al. (2017) in order to reduce the impact of the non-robust Rank-1 Chvátal-Gomory cuts on the labeling algorithm.

We denote the arc-based *ng*-memory of vertex j as $\vec{M}_j$, which is the set of arcs that remember vertex j. Let $P = (a_0, a_1, \ldots, a_p)$ be a partial path composed of arcs $a_0 = (s = i_0, i_1), a_1 = (i_1, i_2), \ldots, a_p = (i_p, i_{p+1})$. Similarly to Equation (5), we now define the set of forbidden extensions for path P as:

$$\vec{\Pi}(P) = \left\{ i_k \in \mathcal{C}(P) \setminus \{i_{p+1}\} : a_s \in \vec{M}_{i_k}, s = k, \ldots, p \right\} \cup \{i_{p+1}\} \tag{7}$$

Sets $\Pi(P)$ and $\vec{\Pi}(P)$ are equivalent if one defines $\vec{M}_k = \{(i,j) \in A : i \in M_k \wedge j \in M_k\}$ for every vertex $k \in V$. This generalized definition is particularly useful in the context of dynamically defined *ng*-memories. In this setting, one wants to augment current *ng*-memories in order to forbid a given cycle $H = (a_0 = (i_0, i_1), a_1 = (i_1, i_2), \ldots, a_p = (i_p, i_{p+1} = i_0))$. For doing so, vertices $i_1, \ldots, i_p$ should be added to the vertex-based *ng*-memory M_{i_0}. However, this forbids not only H, but any cycle $H' = (i_0, \ldots, i_0)$ passing through a subset of $\{i_0, i_1, \ldots, i_p\}$, which may represent a considerable impact on the labeling algorithm. On the other hand, the impact of adding $a_0, \ldots, a_{p-1}$ to $\vec{M}_{i_0}$ is much less considerable since only cycles $H' = (i_0, \ldots, i_q, i_0)$, with $q = 1, \ldots, p$, are forbidden — notice that it is not necessary to add (i_q, i_0) to $\vec{M}_{i_0}$ to forbid a cycle $H' = (i_0, \ldots, i_q, i_0)$ because $i_0 \in \vec{\Pi}((i_0, i_1, \ldots, i_q))$ if $a_0, \ldots, a_{q-1} \in \vec{M}_{i_0}$. We will show in our computational experiments that this reduced impact is crucial for solving hard MLP instances.

Hereafter, the term *ng*-memory refers to arc-based *ng*-memory and we will explicitly indicate if we refer to vertex-based *ng*-memory.

3.3. Fully Dynamic ng-Path Relaxation

Let us consider (arc-based) *ng*-memories $\vec{\mathcal{M}}$ and define $\Omega(\vec{\mathcal{M}})$ as the set of all feasible *ng*-routes w.r.t $\vec{\mathcal{M}}$. Notice that $\Omega \subseteq \Omega(\vec{\mathcal{M}})$. As we have discussed before, the following relaxation of formulation (1)-(3), hereafter denoted $LP(\vec{\mathcal{M}})$, is the basis of several state-of-the-art column generation based algorithms for vehicle routing problems.

$$LB(\vec{\mathcal{M}}) = \min \sum_{r \in \Omega(\vec{\mathcal{M}})} c_r \lambda_r \tag{8}$$

$$\text{s.t.} \quad \sum_{r \in \Omega(\vec{\mathcal{M}})} a_r^i \lambda_r = 1, \qquad \forall i \in \mathcal{C}, \tag{9}$$

$$\lambda_r \geq 0, \qquad \forall r \in \Omega(\vec{\mathcal{M}}). \tag{10}$$

The quality of the bound $LB(\vec{\mathcal{M}})$ depends on the *ng*-memories $\vec{\mathcal{M}}$. Ideally, one should define $\vec{\mathcal{M}}$ so as to guarantee that $LB(\vec{\mathcal{M}})$ corresponds to the bound attained if $\Omega(\vec{\mathcal{M}})$ is replaced by Ω in the formulation, i.e., if only elementary routes are generated in the pricing subproblem. This may require large *ng*-memories, and thus advanced techniques are needed for solving the pricing subproblem in reasonable computational times. For example, Martinelli et al. (2014) described an algorithm based on the decremental state-space relaxation (DSSR) technique suggested by Righini and Salani (2008) where *ng*-memories are iteratively augmented while the best column generated in the pricing subproblem is not *ng*-feasible w.r.t some large *ng*-memories.

Good bounds can also be obtained if *ng*-memories are very well chosen, but not necessarily large. In this regard, Roberti and Mingozzi (2014) introduced the *dynamic ng-path relaxation*, which is roughly a sequence of non-decreasing lower bounds $LB(\mathcal{M}_1), LB(\mathcal{M}_2), \ldots, LB(\mathcal{M}_k)$ associated with dynamically defined vertex-based *ng*-memories $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k$. Initial *ng*-memories $\mathcal{M}_1$ correspond to the Δ^1 nearest customers, and memories $\mathcal{M}_{t+1}$ are computed by extending memories $\mathcal{M}_t$, $t \in \{1, \ldots, k-1\}$, in order to forbid the smallest cycle of the column with the largest primal value in a near-optimal solution of the linear relaxation of $LP(\mathcal{M}_t)$. A limit of Δ^2 is imposed for the size of each *ng*-memory, and the method stops when no cycle can be forbidden. The interested reader should consult (Roberti and Mingozzi, 2014) for further details.

In this section, we propose improvements to the dynamic *ng*-path, the main difference being the possibility to also reduce the *ng*-memories. This reduction is carried out if the pricing subproblems in the previous iteration of the method have been considered expensive. However, the role of this reduction is not only to make the pricing subproblem easier, but also to allow one a better choice of *ng*-memories. Finally, instead of forbidding cycles of a single column, we employ a potentially more aggressive algorithm for augmenting the *ng*-memories. The *fully dynamic ng-path relaxation* is outlined in Algorithm 2, and its main ingredients are described next.

Algorithm 3 presents the procedure adopted to augment *ng*-memories. Up to two phases are executed for each *ng*-route $R \in \mathcal{R}$: first, the procedure attempts to forbid all cycles $H \in \mathcal{H}(R)$ with at most α vertices, where $\mathcal{H}(R)$ denotes the set of all cycles of R (Phase 1); if no cycle could be forbidden in Phase 1, then all cycles $H \in \mathcal{H}(R)$ are considered, regardless of size, and R is skipped after one cycle could be forbidden (Phase 2). We prioritize small cycles because they require less augmentations to be forbidden and many of them are likely to appear repeatedly in low-cost *ng*-routes. The *ng*-routes are sorted in a non-increasing order of primal value or non-decreasing order of reduced cost, depending on where the procedure is called from (Algorithm 2 or Algorithm 4, respectively). The procedure stops if β *ng*-routes have been explicitly forbidden. Notice that parameters α and β control the aggressiveness of the *ng*-memories augmentation. The

Algorithm 2 Fully Dynamic *ng*-Path Relaxation

1: **procedure** fullyDynNgPath($\vec{\mathcal{M}}$, Δ^{max}, α_I, β_I, α_R, β_R)
2: $k \leftarrow 1$, $\vec{\mathcal{M}}_1 \leftarrow \vec{\mathcal{M}}$
3: **repeat**
4: Compute $(\bar{\lambda}_k, \bar{\pi}_k)$, an optimal primal-dual solution pair of $LP(\vec{\mathcal{M}}_k)$
5: **if** *ng*-memories reduction condition is satisfied **then**
6: $\vec{\mathcal{M}}_k \leftarrow$ reduceNGMemories($\vec{\mathcal{M}}_k$, $\bar{\pi}_k$, α_R, β_R)
7: Let $\mathcal{R}$ be the set of *ng*-routes associated with the primal solution $\bar{\lambda}_k$
8: $\vec{\mathcal{M}}_{k+1} \leftarrow$ augmentNgMemories($\mathcal{R}$, $\vec{\mathcal{M}}_k$, $\emptyset$, Δ^{max}, α_I, β_I, `agressive`)
9: **until** stop condition is met

other parameters of this procedure — namely $\mathcal{A}$, Δ^{max} and `mode` — will be explained later in this section.

Quickly, some augmentations performed in previous iterations may become unnecessary to guarantee the bound of the current iteration k. Therefore, we propose Algorithm 4 to reduce *ng*-memories $\vec{\mathcal{M}}_k$. This algorithm is based on the same DSSR technique suggested by Righini and Salani (2008), and also implemented by Martinelli et al. (2014). A problem-dependent condition (pricing time, number of non-dominated labels, etc.) is used to trigger such reductions. Initially empty *ng*-memories are iteratively augmented in order to forbid the columns with the best reduced costs w.r.t $\bar{\pi}_k$. Of course, those improvements are confined to $\vec{\mathcal{M}}_k$ and new iterations are performed as long as the best column generated is not *ng*-feasible w.r.t $\vec{\mathcal{M}}_k$. The algorithm ends up with a set of reduced *ng*-memories $\vec{\mathcal{M}}$ such that $LB(\vec{\mathcal{M}}) = LB(\vec{\mathcal{M}}_k)$. Even though the final *ng*-memories are not necessarily minimal, in practice we have observed that Algorithm 4 often reduces significantly the size of the *ng*-memories.

Remark that we have adopted the same algorithm for augmenting *ng*-memories (i) in the main loop of proposed relaxation, and (ii) inside the *ng*-memories reduction algorithm, but with different parameters. In case (i), more aggressive augmentations are needed for the sake of convergence, whereas in case (ii) moderate augmentations are performed in order to get smaller final *ng*-memories. This is mainly controlled by parameter `mode` of procedure forbidCycle($\cdot$). Let us consider cycle $H = (a_0 = (i_0 = v, i_1), a_1 = (i_1, i_2), \ldots, a_p = (i_p, i_{p+1} = v))$. In moderate mode, only the cycles $H = (v, i_1, \ldots, i_k, v)$, with $k = 1, \ldots, p$, are explicitly forbidden, i.e., we add arcs $a_0, \ldots, a_{p-1}$ to $\vec{M}_v$. On the other hand, in aggressive mode, any cycle starting and ending at v and passing through a subset of $\{i_1, \ldots, i_p\}$ is explicitly forbidden, hence we add any arc between nodes in $\{v, i_1, \ldots, i_p\}$ to $\vec{M}_v$ (see Algorithm 6). We should also point out that even though arc-based *ng*-memories are used, the complexity of the labeling algorithm is still somewhat vertex-dependent. As usual, let $\delta^-(j)$ denote the set of arcs entering vertex j and let $\gamma(j)$ be the number of *ng*-memories arcs $\delta^-(j)$ belong to. The number of non-dominated labels representing paths ending at a vertex j may be exponential on $\gamma(j)$. Thus, procedure cycleCanBeForbidden($\cdot$),

Algorithm 3 *ng*-Memories Augmentation Algorithm

procedure augmentNgMemories($\mathcal{R}$, $\vec{\mathcal{M}}$, $\mathcal{A}$, Δ^{max}, α, β, `mode`)
  input $\mathcal{R}$: set of target *ng*-routes, $\vec{\mathcal{M}}$: current *ng*-memories, $\mathcal{A}$: set of forbidden augmentations, Δ^{max}: maximum value allowed for $\gamma(\cdot)$, α: maximum cycle size in Phase 1, β: maximum number of *ng*-routes explicitly forbidden, `mode`: mode of augmentation.
  output: new *ng*-memories

  $\mathcal{F}_H \leftarrow \emptyset$, $\mathcal{F}_R \leftarrow \emptyset$
  for each $R \in \mathcal{R}$ **do**
    phase $\leftarrow 1$
    for each $H = (v, \ldots, v) \in \mathcal{H}(R)$ in non-decreasing order of size **do**
      if $|H| > \alpha$ **and** phase $= 1$ **then**
        if $\mathcal{H}(R) \cap \mathcal{F}_H = \emptyset$ **then**
          phase $\leftarrow 2$
        else
          break
      if cycleCanBeForbidden(H, $\vec{\mathcal{M}}$, Δ^{max}, $\mathcal{A}$) **then**
        $\vec{\mathcal{M}} \leftarrow$ forbidCycle(H, $\vec{\mathcal{M}}$, `mode`)
        $\mathcal{F}_H \leftarrow \mathcal{F}_H \cup \{H\}$, $\mathcal{F}_R \leftarrow \mathcal{F}_R \cup \{R\}$
        if phase $= 2$ **then**
          break
    Stop if $|\mathcal{F}_R| \geq \beta$
  return $\vec{\mathcal{M}}$

described in Algorithm 5, considers a maximum value Δ^{max} allowed for $\gamma(\cdot)$. Additionally, it also considers a set $\mathcal{A}$ of forbidden augmentations, which is used in the context of the *ng*-memories reduction algorithm.

4. Branch-and-Price Algorithm

In this section, we describe the proposed branch-and-price algorithm (BP) for MLP. The solution of a node in our algorithm, outlined in Algorithm 7, is an iterative approach based on the fully dynamic *ng*-path relaxation described in Section 3.3. At each iteration k, we first solve $LP(\vec{\mathcal{M}}_k)$ by means of a two-stage column generation. Of course, branching constraints may also be present in this linear program. In Stage 1, dominance tests take into account only conditions (I), (II) and (III), thus a single *ng*-path is kept for a given vertex i and resource consumption w, which is the one with minimum reduced cost. This is a heuristic pricing intended to quickly generate good *ng*-paths. When Stage 1 fails to find a *ng*-path with negative reduced cost, we switch to Stage 2, where the exact pricing is solved. In both phases, the dual stabilization technique of Pessoa et al. (2013) is applied for the sake of convergence. Stage 1 (2) is solved by a mono-directional (bidirectional) labeling algorithm that returns at most 50 (300) *ng*-paths. Details on

Algorithm 4 *ng*-Memories Reduction Algorithm

procedure reduceNGMemories($\vec{\mathcal{M}}$, π^*, α, β)
  input: $\vec{\mathcal{M}}$: current *ng*-memories, π^*: dual solution of $LP(\vec{\mathcal{M}})$, and parameters to augmentNgMemories($\cdot$)
  output: new *ng*-memories

  $\vec{\mathcal{M}}_{ori} \leftarrow \vec{\mathcal{M}}$, $\vec{\mathcal{M}} \leftarrow \emptyset$, *ng*-feasible $\leftarrow$ false
  while not *ng*-feasible **do**
    $\mathcal{R} \leftarrow$ LabelingAlgorithm($\vec{\mathcal{M}}$, π^*)
    if best route in $\mathcal{R}$ is *ng*-feasible w.r.t $\vec{\mathcal{M}}_{ori}$ **then**
      ng-feasible $\leftarrow$ true
    else
      Define $\mathcal{A}$ as the set of augmentations that are not confined to $\vec{\mathcal{M}}_{ori}$
      $\vec{\mathcal{M}} \leftarrow$ augmentNgMemories($\mathcal{R}$, $\vec{\mathcal{M}}$, $\mathcal{A}$, ∞, α, β, `moderate`)
  return $\vec{\mathcal{M}}$

such algorithm will be given in Section 4.1.

If the node is the root, the first memories $\vec{\mathcal{M}}_1$ are equivalent to *ng*-sets of size 8 defined according to the classical distance-based rule — time-based for the case of MLP. Otherwise, they correspond to the final memories of the parent node, which are inherited by the child. In any node, a hybrid strategy for augmenting *ng*-memories may be used. Initially, we set `mode` $\leftarrow$ `agressive`. The method switches to `moderate` mode if the computational time of a single call to the labeling algorithm exceeds a threshold value $t^{red} = 100$ seconds. In this case, column generation is interrupted and the *ng*-memories reduction algorithm (see Section 3.3) is called with $\alpha_R = 5$ and $\beta_R = 200$; afterwards, the current iteration is restarted with the new augmentation mode. Regardless of mode, we have adopted $\alpha_I = 5$, $\beta_I = 200$ and $\Delta^{max} = 63$.

As we have already discussed, the pricing problem corresponds to finding a least-cost *ng*-path in the resource constrained network G' defined in Section 2. However, in practice, we work with an extended network $G^{ext} = (V^{ext}, A^{ext})$, where:

$$V^{ext} = \{(i,w) : i, w \in \{1,\ldots,n\}\} \cup \{(s,0),(t,n+1)\}$$
$$A^{ext} = \{((i,w),(j,w+1)) : (i,j) \in A, (i,w) \in V^{ext}, (j,w+1) \in V^{ext}\}$$

Network G^{ext} is defined in such a way that resource constraints are naturally satisfied by any *ng*-path. Once an optimal dual solution $\bar{\pi}_k$ is available, a reduced cost fixing procedure is used to remove from A^{ext} the arcs that can not participate in a solution that improves the current upper bound. For fixing an arc $((i,w),(j,w+1))$, one should prove that the minimum reduced cost of a *ng*-path traversing this arc is above a given threshold. To compute this cost, the minimum reduced cost of a partial path ending at (i,w) and of a partial path from $(j,w+1)$ to the sink node

Algorithm 5 Test for Cycle Elimination

procedure cycleCanBeForbidden(H, $\vec{\mathcal{M}}$, $\mathcal{A}$, Δ^{max})
  input: $H = (a_0 = (v = i_0, i_1), a_1 = (i_1, i_2), \ldots, a_p = (i_p, i_{p+1} = v))$: target cycle, $\vec{\mathcal{M}}$: current *ng*-memories, $\mathcal{A}$: set of forbidden augmentations, Δ^{max}: maximum value allowed for $\gamma(\cdot)$.
  output: a flag indicating whether H can be forbidden or not

  for $q = 0$ **to** $p - 1$ **do**
    if $a_q \notin \vec{M}_v$ **then**
      $\gamma(i_{q+1}) \leftarrow |\{j \in V \setminus \{v\} : \delta^-(i_{q+1}) \cap \vec{M}_j \neq \emptyset\}|$
      if $\gamma(i_{q+1}) \geq \Delta^{max}$ **or** $(\vec{M}_v, a_q) \in A$ **then**
        return false
  return true

Algorithm 6 Cycle Elimination

procedure forbidCycle(H, $\vec{\mathcal{M}}$, `mode`)
  input: $H = (a_0 = (v = i_0, i_1), a_1 = (i_1, i_2), \ldots, a_p = (i_p, i_{p+1} = v))$: target cycle, $\vec{\mathcal{M}}$: current *ng*-memories, `mode`: mode of augmentation.

  for $q = 0$ **to** $p - 1$ **do**
    if `mode` = `moderate` **then**
      $\vec{M}_v \leftarrow \vec{M}_v \cup \{(i_q, i_{q+1})\}$
    else
      for $t = q + 1$ **to** p **do**
        $\vec{M}_v \leftarrow \vec{M}_v \cup \{(i_q, i_t), (i_t, i_q)\}$

are computed by a forward and a backward labeling algorithms, respectively. Such a procedure is well-known in the literature and have been applied in many routing problems (see, for instance, (Roberti and Mingozzi, 2014; Pecin et al., 2016)). Dual solution $\bar{\pi}_k$ is also used in an enumerative procedure that tries to finish the node. As implemented by Roberti and Mingozzi (2014), the enumeration is performed by a mono-directional labeling that computes only elementary paths, using completion bounds associated with *ng*-paths to prune unpromising partial paths. However, we abort the enumeration if the number of non-dominated labels is greater than 10 millions. If enumeration finishes, either a single path representing the best integer solution for the node is returned, or all labels are dominated, meaning that this solution does not improve the current upper bound.

Besides the obvious stop conditions (node is solved or pruned), Algorithm 7 stops if:

- **Stop condition I:** Labeling algorithm time has exceeded the threshold value t^{red} twice.
- **Stop condition II:** No column could be forbidden by augmentNgMemories(·) because of

Algorithm 7 Solution of a node

procedure solveNode($\vec{\mathcal{M}}$, B)
  $k \leftarrow 1$, $\vec{\mathcal{M}}_1 \leftarrow \vec{\mathcal{M}}$
  `mode` $\leftarrow$ `agressive`
  repeat
    Compute $(\bar{\lambda}_k, \bar{\pi}_k)$, an optimal primal-dual solution pair of $LP(\vec{\mathcal{M}}_k)$ + branching constraints B
    if *ng*-memories reduction condition is satisfied **then**
      $\vec{\mathcal{M}}_k \leftarrow$ reduceNGMemories($\vec{\mathcal{M}}_k$, $\bar{\pi}_k$, 5, 200)
      `mode` $\leftarrow$ `moderate`
      continue
    Apply reduced cost fixing
    Try to finish the node by enumeration
    Let $\mathcal{R}$ be the set of *ng*-routes associated with the primal solution $\bar{\lambda}_k$
    $\vec{\mathcal{M}}_{k+1} \leftarrow$ augmentNgMemories($\mathcal{R}$, $\vec{\mathcal{M}}_k$, $\emptyset$, 63, 5, 200, `mode`)
    $k \leftarrow k + 1$
  until stop condition is met

Δ^{max}.

- **Stop condition III:** For 5 times, the gap of the current iteration is less than 2% smaller than the one of the previous iteration.

We branch on an edge $\{i, j\}$ defined by a 3-phase strong branching mechanism in the spirit of the works of Røpke (2012) and Pecin et al. (2016). Hence, vertices i and j must appear consecutively (either $i \rightarrow j$ or $j \rightarrow i$) in the solution of one child and must not be consecutive in the solution of the other child. In Phase 0, we define $\min\{100, TS(v)\}$ candidate edges, where $TS(v)$ is an estimation for the size of the tree rooted at node v. Such estimation takes into account the average bound improvement in the branch history, following the model of Kullmann (2009). $TS(v) = \infty$ if v is the root node. First, we select up to half of the edges from a pool containing the candidates evaluated in previous executions of Phase 2 in the whole branch history. The other candidates are the edges whose values are the closest to 0.5 in the current fractional solution. For each candidate selected in Phase 0, we perform a rough evaluation of both children by solving the master LP of node v with the corresponding branching constraint, but without any column generation. This is Phase 1, where the best $\min\{5, TS(v)/10\}$ candidates are selected, according to the product rule of (Achterberg, 2007), to go to the last phase. Phase 2 uses the same approach as for Phase 1, but heuristic column generation (Stage 1) is applied when evaluating the children. The selected edge is the one with the best score in Phase 2, also according to the product rule.

We now provide more information on the implementations of the labeling algorithm and of MPLD.

4.1. Labeling Algorithm

Following many related works in the literature — for instance, (Martinelli et al., 2014) and (Pecin et al., 2016) — we have adopted the concept of bucket in our implementation of the general method described in Algorithm 1. A *forward bucket* $\overrightarrow{B}(j, w)$ is data structure that stores all non-dominated labels associated with *forward* $s - j$ paths with resource consumption w. Therefore, for each vertex j we define buckets $\overrightarrow{B}(j, w)$, $\forall w \in \{l_j, \ldots, u_j\}$. To accelerate the algorithm, labels in a bucket are kept sorted by non-decreasing order of cost and dominance checks are performed only between labels of the same bucket, just as implemented by Pecin et al. (2016). Buckets are considered by the algorithm in a non-decreasing order of resource consumption. When a bucket $\overrightarrow{B}(j, w)$ is reached, we extend all labels in $\overrightarrow{B}(j, w)$ over all arcs $(j, k) \in A$. At the end, the optimal $s - t$ paths will be stored in the buckets associated with the sink node t.

The algorithm outlined above is called mono-directional since all labels kept correspond to partial paths from the source node to some node j and arcs are traversed in their regular directions. However, in our implementation we have used a bidirectional labeling algorithm, which generates more diversified set of $s - t$ paths and is typically faster than its mono-directional counterpart. A *backward path* corresponds to a partial path from the sink to some node j obtained by traversing arcs in their reverse directions. Labels corresponding to such paths are stored in *backward buckets* $\overleftarrow{B}(j, w)$. In the forward (backward) labeling algorithm, one computes non-dominated labels for buckets $\overrightarrow{B}(j, w)$ ($\overleftarrow{B}(j, w)$) such that $w \leq w^*$ ($w \geq w^*$), where w^* is a threshold value that ideally should be defined so as to balance the computational effort of the forward and the backward labeling algorithms. Then, a concatenation procedure is execute to build complete $s - t$ paths from the partial forward and backward paths. The reader is referred to (Righini and Salani, 2006) for further references on bidirectional labeling.

4.2. Multiple Partial Label Dominance

Let us consider again labels $L(P')$ and $L(P)$ such that conditions stated in Proposition 1 are satisfied, and define $\Pi' = \Pi(P') \setminus \Pi(P)$. As we have discussed, the extensions to vertices $j \in V \setminus \bigcup_{k \in \Pi'} M_k$ can be avoided for label $L(P)$. For example, in Figure 1, one can verify that any extension to vertices $j \notin M_3$ can be avoided for label $L(P_4)$ because of label $L(P_3)$, and thus set $\xi(P_4)$ is increased as $\xi(P_4) \leftarrow \xi(P_4) \cup \{2, 5, 6\}$. Such an operation is performed several times during the course of the labeling algorithm and an efficient implementation is required, otherwise the gains incurred by MPLD will not pay off. In what follows, we will describe two approaches that we have tried to take advantage of this new dominance rule.

4.2.1. Explicit Representation of $\xi(\cdot)$

We first tried an explicit representation of sets $\xi(\cdot)$ inside each label L.

- **Bitmap representation**: Sets $\xi(\cdot)$ are represented as bitmaps implemented over 64-bit integers. For each vertex $i \in V$, MPLD can avoid the extension of labels $L(P) \in U_i$ only

over a set E_i of at most 63 arcs. This set is composed of arcs $(i,j) \in A$ with the smallest costs, and the 64th bit of the bitmaps are always set to zero to indicate that arcs $(i,j) \notin E_i$ are not handled by MPLD. In practice, 63 is a reasonable number of arcs tracked and allows very quick operations since a single 64-bit integer is needed to represent a bitmap. The bit corresponding to arc (i,j) is checked in constant time through bitwise operations before extending labels $L(P) \in U_i$ over (i,j).

- **Precomputed union of memories**: The number of different possible sets $\Pi(P)$ for a path P ending at vertex i is $2^{|N_i|}$. Thus, for two labels $L(P_1)$ and $L(P_2)$ such that $v(P_1) = v(P_2) = i$, the number of different possible sets $\Pi' = \Pi(P_1) \setminus \Pi(P_2)$ is also $2^{|N_i|}$. For reasonable values of $|N_i|$, it is practical to precompute sets $V \setminus \bigcup_{k \in \Pi'} M_k$ for every possible Π'. In fact, we have observed that gains of MPLD are diminished if these sets are not precomputed. When testing the dominance of label $L(P_2)$ by label $L(P_1)$, a bitmap representing $V \setminus \bigcup_{k \in \Pi'} M_k$ is retrieved and used to update $\xi(P_2)$ in constant time through bitwise operations. Notice that in our implementation such dominance checks are performed within a same bucket.

Computational experiments showed that the computational time speed-up incurred by this approach is not significant. The main limitation is that one needs to restrict the cardinality of sets $\xi(\cdot)$ to at most 63 in order to quickly manipulate the bitmaps representing them. In this case, MPLD is not completely explored. Moreover, we have adopted a distance-based criterion to decide the 63 extensions tracked, which may be a very crude criterion in some cases. For example, an extension over a *long arc* (i.e., an arc connecting two customers that are far away from each other) will probably generate a label $L(P)$ with a large cost, but with few customers in set $\Pi(P)$. This latter attribute makes very hard to dominate label $L(P)$, increasing the number of dominance checks in the destination bucket. A dynamic definition of tracked extensions would probably improve results, but the following approach is simpler and mitigates all aforementioned problems.

4.2.2. Implicit Representation of $\xi(\cdot)$

Recall that, in our implementation of the labeling algorithm, buckets are considered in a predefined order and labels of a same bucket are extended in a non-decreasing order of cost. Moreover, we extend all labels of a bucket over an arc, then over another arc and so on. Let us consider a (forward or backward) bucket $B(i,w)$ and an arc (i,j). Suppose that a label $L(P') \in B(i,w)$ has already been extended to a label $L(P'+j)$. Now let $L(P) \in B(i,w)$ be a label that has not yet been extended over (i,j) such that $\Pi(P'+j) \subseteq \Pi(P+j)$. Since its cost is larger, $L(P+j)$ will be dominated by label $L(P'+j)$ and thus one can avoid the extension of $L(P)$ over (i,j) — this is just an example of MPLD. In general, for each bucket $B(i,w)$ and arc (i,j), we keep a list of bitmaps representing sets $\Pi(\cdot)$ of already extended labels. Then, before extending a label $L(P) \in B(i,w)$ over (i,j), we first compute $\Pi(P+j)$ and check if it is a superset of some $\Pi(P'+j)$ contained in the list. If so, the extension is not necessary and we proceed to

the next label, otherwise we complete the extension of the label and update the list. Of course, if $\Pi(P+j) = \{j\}$, then we can stop extending labels in $B(i,w)$ over (i,j). Furthermore, there is no need to keep the list after considering the extension of all labels in $B(i,w)$ over (i,j). This approach better explores the potential of MPLD and typically yields a significant speed up in the pricing time, as will be seen in next section.

5. Computational Experiments

This section reports our computational experiments over 40 TSPLIB instances with up to 200 vertices. The proposed BP algorithm was implemented in C++ over the BaPCod platform of (Vanderbeck et al., 2017). The LP solver adopted is IBM CPLEX Optimizer version 12.6.0. All experiments were conducted on an Intel Xeon E5-2680 v3, running at 2.5 GHz with a single thread. We will compare our results to those obtained by the algorithm of Roberti and Mingozzi (2014) on an Intel Xeon X7350, running at 2.93 GHz. According to the CPU benchmark website `www.cpuboss.com`, the single thread performance of our CPU is about 1.5 times better than the one of Roberti and Mingozzi (2014). Therefore, our computational times are increased by this factor in Table 1.

5.1. Main Results

Table 1 presents the main results of the proposed BP and the results of Roberti and Mingozzi (2014) over instances with up to 152 vertices. In this case, we set a time limit of 2 days for BP. For what concerns the results of Roberti and Mingozzi (2014), we report the following data: lb, the final lower bound; t_{lb}, the computational time to obtain such lower bound; t_{tot}, the total computational time; and gap, the percentage gap before applying enumeration. A symbol "-" in column t_{tot} indicates that enumeration failed to find the optimal solution. The results of the proposed method are divided into root node and complete BP. For the root node, we report the first and last lower bounds obtained, lb^0 and lb^k, where k is the number of *ng*-memories augmentations performed. Further, we present the average and maximum values of $\gamma(\cdot)$ and the time spent at the root node. We also indicate if the method has switched to `moderate` mode in the root node. For the complete BP, we report the final bounds and gap, the total computational time and the number of nodes solved. All computational times in Table 1 are given in seconds.

We can observe in Table 1 that the proposed BP outperforms the method of Roberti and Mingozzi (2014) in most instances. In particular, 6 instances were solved for the first time and the BKS for instance kroA150 was improved from 1831766 to the optimal value 1825769. The main advantage of our method is the possibility of dealing with larger *ng*-memories without combinatorial explosion. For example, for the 6 instances solved only by BP, the maximum values of $\gamma(\cdot)$ are greater than 30, reaching 62 in eil101. This is possible only because of the proposed generalized definition of *ng*-sets in terms of arcs. Moreover, our computational experience shows that the hybridization of `agressive` and `moderate` modes is crucial for the robustness

Table 1: Results of BP over TSPLIB instances used by Roberti and Mingozzi (2014). Computational times are normalized according to the CPU benchmark website `www.cpuboss.com`.

		Roberti and Mingozzi (2014)				Proposed Method											
						Root Node							B&P				
Instance	z^*_{lit}	lb	t_{lb}	t_{tot}	gap	lb^0	lb^k	k	switched mode	Max. $\gamma(\cdot)$	Avg. $\gamma(\cdot)$	t_{root}	lb	ub	gap	nodes	t_{tot}
dantzig42	12528	12528	5	5	0.00	12528	12528	0	no	8	8.0	4.7	12528	12528	0.00	1	4.7
swiss42	22327	22327	3	3	0.00	22327	22327	0	no	8	8.0	2.1	22327	22327	0.00	1	2.1
att48	209320	209320	7	7	0.00	209320	209320	0	no	8	8.0	8.5	209320	209320	0.00	1	8.5
gr48	102378	102378	15	15	0.00	102378	102378	0	no	8	8.0	7.2	102378	102378	0.00	1	7.2
hk48	247926	247926	19	19	0.00	247926	247926	0	no	8	8.0	10.1	247926	247926	0.00	1	10.1
eil51	10178	10178	10	10	0.00	10178	10178	0	no	8	8.0	6.5	10178	10178	0.00	1	6.5
berlin52	143721	143721	10	10	0.00	143721	143721	0	no	8	8.0	10.1	143721	143721	0.00	1	10.1
brazil58	512361	512361	36	36	0.00	512361	512361	0	no	8	8.0	17.7	512361	512361	0.00	1	17.7
st70	20557	20557	61	61	0.00	20557	20557	0	no	8	8.0	24.0	20557	20557	0.00	1	24.0
eil76	17976	17976	91	91	0.00	17976	17976	0	no	8	8.0	26.6	17976	17976	0.00	1	26.6
pr76	3455242	3423016	169	223	0.93	3375941	3455242	2	no	11	8.6	169.1	3455242	3455242	0.00	1	169.3
gr96	2097170	2087792	365	369	0.45	2061228	2097170	2	no	11	8.7	215.3	2097170	2097170	0.00	1	215.3
rat99	57986	57541	540	-	0.77	57064	57623	17	yes	38	25.7	1467.9	57986	57986	**0.00**	3	3180.4
kroA100	983128	980428	679	688	0.27	968668	983128	3	no	15	10.3	337.2	983128	983128	0.00	1	337.2
kroB100	986008	986008	197	197	0.00	986008	986008	0	no	8	8.0	75.2	986008	986008	0.00	1	75.2
kroC100	961324	957564	409	428	0.39	939298	961324	5	no	22	14.7	815.7	961324	961324	0.00	1	815.9
kroD100	976965	972814	708	729	0.42	957467	976965	2	no	13	9.2	290.0	976965	976965	0.00	1	290.0
kroE100	971266	971266	266	266	0.00	971266	971266	0	no	8	8.0	87.7	971266	971266	0.00	1	87.7
rd100	340047	339919	317	317	0.04	337230	340047	1	no	10	8.2	213.1	340047	340047	0.00	1	213.3
eil101	27513	27235	630	-	1.01	26892	27319	17	yes	62	35.1	4943.9	27513	27513	**0.00**	3	6238.9
lin105	603910	603910	154	154	0.00	603910	603910	0	no	8	8.0	85.5	603910	603910	0.00	1	85.5
pr107	2026626	2026626	246	246	0.00	2007255	2026626	2	no	12	9.1	470.9	2026626	2026626	0.00	1	470.9
gr120	363454	360460	4685	-	0.82	353054	359970	18	yes	41	25.2	10532.0	363454	363454	**0.00**	11	79351.6
pr124	3154346	3154346	14080	14080	0.00	3001043	3154346	21	yes	31	20.2	58283.6	3154346	3154346	0.00	1	58283.6
bier127	4545005	4545005	723	723	0.00	4513768	4545005	1	no	12	8.3	307.7	4545005	4545005	0.00	1	307.7
ch130	349874	348015	2385	-	0.53	341809	348013	16	yes	43	25.7	3767.8	349874	349874	**0.00**	3	7810.0
pr136	6199268	6160253	17190	-	0.63	5908131	6155720	23	yes	31	22.7	15845.4	6175226	6199268	0.39	7	259200.0
gr137	4061498	4025358	7844	-	0.89	3952022	4017150	15	yes	33	19.8	12140.2	4061498	4061498	**0.00**	15	65614.2
pr144	3846137	3841847	8043	8061	0.11	3671878	3846137	4	no	13	9.9	6101.5	3846137	3846137	0.00	1	6101.6
ch150	444424	444118	2585	2588	0.07	438450	444424	6	no	22	12.6	1742.6	444424	444424	0.00	1	1743.0
kroA150	1831766	1818024	5187	-	0.75	1794351	1813870	13	yes	33	17.4	7250.5	1825769	**1825769**	**0.00**	15	93566.8
kroB150	1793204	1775914	20120	-	0.96	1720498	1752709	17	yes	42	24.7	12265.3	1766343	1793204	1.52	19	259200.0
pr152	5064566	4931382	31218	-	2.63	4803249	4984428	15	yes	18	13.2	127472.1	4988337	5064566	1.53	3	259200.0

of proposed method. Even though some instances are better solved if only slower arc-based *ng*-memories augmentations are performed (e.g. eil101 and gr120), others instances, such as pr124, could not be solved without `agressive` mode. Instances pr136, pr152 and kroB150 could not be solved, but we remark that "pr" instances have a special structure with a lot of symmetry (see Figure 2). However, we will show in the next section that a specific parameterization of our method is capable of solving those 3 instances.

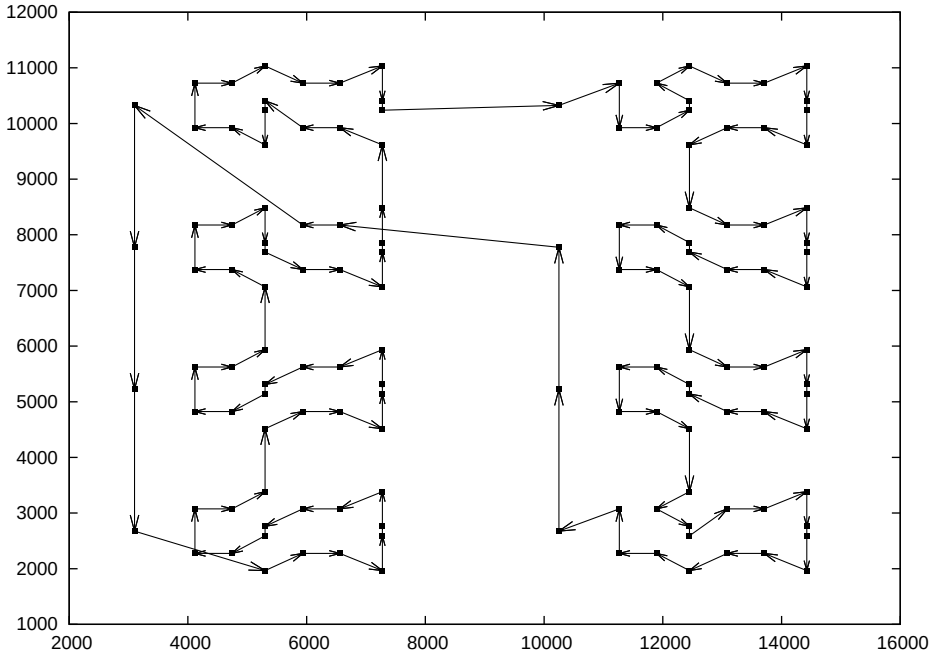

Figure 2: Optimal solution of pr136.

Table 2 shows the results of BP over large TSPLIB instances with up to 200 vertices. To our knowledge, this is the first time that such large instances are considered by an exact method for MLP. The upper bounds were computed by the method of Silva et al. (2012). Instances u159, si175, brg180 and rat195 were solved in reasonable computational times and small BP trees, but BP finished with considerable gaps for the other instances. For rat195, BP found an optimal solution with cost 218632, an improvement of 43 units over the heuristic solution found by the method of Silva et al. (2012). In spite of the new contributions presented in this paper, MLP instances with about 200 vertices still seem to be very challenging.

5.2. Longer runs with `moderate` *mode*

Here we show that our method can solve some more hard instances by using a different parameterization. The idea of this parameterization is to augment *ng*-memories very slowly and over a large number of iterations. More precisely, we adopt `moderate` mode in all iterations of Algorithm 7, set a time limit of 5 days for BP and change stop conditions I and III as below.

Table 2: Results of BP over large TSPLIB instances

		Root Node							B&P				
Instance	z^*_{lit}	lb^0	lb^k	k	switched mode	Max. $\gamma(\cdot)$	Avg. $\gamma(\cdot)$	t_{root}	lb	ub	gap	nodes	t_{tot}
u159	2972030	2892904	2960764	25	yes	43	26.65	19881.7	2972030	2972030	**0.00**	3	21437.5
si175	1808532	1801860	1808195	20	yes	38	22.04	22874.3	1808532	1808532	**0.00**	3	24004.1
brg180	174750	164672	174750	8	no	18	13.42	3353.6	174750	174750	**0.00**	1	3353.7
rat195	218675	216725	217830	13	yes	50	25.50	3529.6	218632	**218632**	**0.00**	11	55814.8
d198	1186049	1128131	1144047	12	yes	23	10.97	30457.7	1147364	1186050	3.37	7	172800.0
kroA200	2672437	2600209	2631523	15	yes	45	24.39	20536.9	2637610	2672438	1.32	17	172800.0
kroB200	2669515	2580374	2636041	15	yes	30	15.08	13809.2	2643616	2669516	0.98	13	172800.0

Table 3: Results of BP over hard instances with a different parameterization

		Root Node						B&P				
Instance	z^*_{lit}	lb^0	lb^k	k	Max. $\gamma(\cdot)$	Avg. $\gamma(\cdot)$	t_{root}	lb	ub	gap	nodes	t_{tot}
pr136	6199268	5908131	6182837	125	52	37.43	45071.2	6199268	6199268	**0.00**	15	187356.3
kroB150	1793204	1720498	1760182	103	63	44.98	13371.8	1786546	**1786546**	**0.00**	23	293479.1
pr152	5064566	4803249	4980497	110	26	17.22	38839.8	5064566	5064566	**0.00**	3	281061.9
d198	1186049	1128131	1145300	101	31	17.68	27433.4	1152702	1186050	2.89	17	432000.0
kroA200	2672437	2600209	2639043	100	63	48.40	32154.6	2651933	2672438	0.77	18	432000.0
kroB200	2669515	2580374	2646679	115	63	39.91	61444.4	2653034	2669516	0.62	15	432000.0

- **Stop condition I:** Labeling algorithm time has exceeded the threshold value $\mathbf{t^{red} = 150}$.
- **Stop condition III:** For **100** times, the gap of the current iteration is less than 2% smaller than the one of the previous iteration.

Table 3 shows the results obtained. For the first time, instances pr136, pr152 and kroB150 were solved to optimality. Therefore, we solved all instances that were not solved by Roberti and Mingozzi (2014). Furthermore, the BKS for kroB150 was improved from 1793204 to the optimal value 1786546, an improvement of 0.03%. Still, we should point out that such a parameterization is useful only for hard instances. In general, BP has a worse performance if a weaker tailing off condition is used.

5.3. Multiple Partial Label Dominance

Table 4 shows the performance of the labeling algorithm with and without MPLD. The average results presented in that table concern the exact calls to the labeling algorithm in the first descent of column generation with *ng*-sets of a given fixed size in $\{8, 12, 16\}$. A representative subset of the instances was used. With *ng*-sets of size 16, instances pr124, pr144 and pr152 exceeded a time limit set for this experiment, thus no results are reported in those cases. It can be seen in Table 4 that the new dominance rule has a significant impact on the labeling algorithm. The average number of extensions decreased by an order of magnitude for any tested instance. This allowed the algorithm to achieve remarkable average speedups of 4.37, 6.01 and 7.28 in computational

time for *ng*-sets of sizes 8, 12 and 16, respectively. Of course, with good lower and upper bounds and some rounds of reduced cost fixing, the pricing networks gets sparser and those speedups are smaller, but still considerable. Surprisingly, for most instances, the percentage of non-dominated labels that are never extended — column **MPLD (%)** — is significantly large. This means that not rarely a label is completely dominated by a set of two or more labels, but not by a single label as required by the classical dominance rule.

Table 4: Performance of the labeling algorithm with/without multiple partial label dominance

		Avg. time (s)			**Avg. number of extensions**			**MPLD**
Instance	$\|N_i\|$	without	with	factor	without ($\times 10^6$)	with ($\times 10^6$)	factor	(%)
gr120	8	1.16	0.33	3.51	17.90	1.97	9.08	24.54
pr124	8	1.83	0.48	3.82	30.38	2.60	11.68	7.76
bier127	8	1.78	0.42	4.23	28.37	2.56	11.07	11.56
ch130	8	1.78	0.42	4.25	27.06	2.58	10.47	16.83
pr136	8	2.35	0.55	4.26	36.67	3.08	11.89	19.58
gr137	8	1.63	0.46	3.53	28.54	2.99	9.55	17.73
pr144	8	8.51	1.28	6.64	108.24	5.38	20.11	2.01
ch150	8	2.26	0.61	3.71	36.25	3.75	9.66	24.69
kroA150	8	2.74	0.69	3.98	42.51	3.90	10.89	18.29
kroB150	8	2.96	0.73	4.07	44.23	3.97	11.14	17.64
pr152	8	7.77	1.27	6.11	108.04	5.87	18.41	5.08
Average for ng 8				**4.37**			**12.18**	**15.06**
gr120	12	4.74	0.82	5.76	53.77	2.98	18.04	19.40
pr124	12	38.77	15.66	2.48	203.96	9.73	20.96	4.03
bier127	12	8.40	1.46	5.77	85.20	4.24	20.09	11.58
ch130	12	7.60	1.14	6.68	81.84	3.98	20.58	16.33
pr136	12	11.40	1.80	6.32	120.52	5.44	22.17	13.50
gr137	12	7.87	1.41	5.59	95.40	4.93	19.33	13.96
pr144	12	174.41	33.38	5.23	555.34	19.80	28.05	1.62
ch150	12	8.55	1.27	6.72	102.46	5.09	20.11	22.03
kroA150	12	12.12	1.62	7.48	129.82	5.82	22.32	16.56
kroB150	12	14.06	2.04	6.90	140.31	6.37	22.02	15.14
pr152	12	212.33	29.70	7.15	731.47	23.54	31.08	2.28
Average for ng 12				**6.01**			**22.25**	**12.40**
gr120	16	28.68	5.21	5.50	158.58	6.84	23.18	12.30
pr124	16	-	-	-	-	-	-	-
bier127	16	57.66	8.99	6.42	256.19	9.41	27.22	9.46
ch130	16	35.69	4.35	8.21	210.17	7.62	27.58	13.70
pr136	16	65.27	8.96	7.29	334.53	13.12	25.51	10.41
gr137	16	48.09	8.84	5.44	286.45	11.56	24.78	10.08
pr144	16	-	-	-	-	-	-	-
ch150	16	40.89	5.12	7.98	279.85	9.47	29.55	16.99
kroA150	16	66.00	6.69	9.86	360.41	11.21	32.16	13.62
kroB150	16	89.06	11.86	7.51	409.30	13.98	29.28	12.40
pr152	16	-	-	-	-	-	-	-
Average for ng 16				**7.28**			**27.41**	**12.37**

6. Conclusions

This paper dealt with the Minimum Latency Problem (MLP), a variant of the Traveling Salesman Problem (TSP) where the objective is to minimize the sum of waiting times of customers.

A branch-and-price (BP) algorithm over a set partitioning formulation was introduced, where columns are *ng*-paths. As implemented by Roberti and Mingozzi (2014), our algorithm is based on dynamically defined *ng*-memories. The proposed BP benefits from well-known elements of efficient exact method for routing problems, such as dual stabilization, reduced cost fixing, route enumeration and strong branching.

Although those elements are very important for the good performance of BP, the main sources of efficiency of our method are the new features for the *ng*-path relaxation. For example, the new dominance rules typically speeds the labeling algorithm by factors between 4 and 8. Furthermore, the proposed generalized definition of *ng*-sets in terms of arcs opened the way for less harmful augmentations of *ng*-memories, in contrast to the vertex-based augmentations introduced by Roberti and Mingozzi (2014). Nevertheless, our experiments showed that the best strategy for augmenting *ng*-memories is instance-dependent: some of them are better solved with arc-based augmentations, others with vertex-based augmentations. For example, pr124 could not be solved in 2 days using only arc-based augmentations, while pr136, pr144 and p152 were solved for the first time mainly because of them. Hence, for the sake of robustness, the solution of a node in our BP starts with vertex-based augmentations, switching to arc-based ones if the computational time of the labeling algorithm is too large. The proposed *ng*-memories reduction algorithm is crucial in this transition, reverting previous augmentations that are not needed to attain the current bound. We should mention that we did try to use this reduction of *ng*-memory more frequently, but it is time-consuming for large instances and affects the convergence of Algorithm 7. Still, we believe that the combination of augmentations and reductions of *ng*-memories deserves further investigation.

All the 9 instances not solved by Roberti and Mingozzi (2014) were solved. Also, BP could solve the larger instances u159, si175, br180 and rat195, but could not solve d198, kroA200 and kroB200. In general, MLP instances with about 150 vertices (or more) are still very challenging.

Acknowledgments

Experiments presented in this paper were carried out using the PlaFRIM (Federative Platform for Research in Computer Science and Mathematics), created under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux, CNRS and ANR in accordance to the "Programme d'Investissements d'Avenir" (see `www.plafrim.fr/en/home`). TB and EU visited Bordeaux with funding from project FAPERJ E26/110.075/2014 (International Cooperation FAPERJ/INRIA). We also thank Marcos Melo Silva and Anand Subramanian for computing upper bounds for the large instances tested only in this paper.

References

Abeledo, H., Fukasawa, R., Pessoa, A., Uchoa, E., 2013. The time dependent traveling salesman problem: polyhedra and algorithm. Mathematical Programming Computation 5 (1), 27–55.

Achterberg, T., 2007. Constraint integer programming. Ph.D. thesis, Technische Universitat Berlin.

Afrati, Foto, Cosmadakis, Stavros, Papadimitriou, Christos H., Papageorgiou, George, Papakostantinou, Nadia, 1986. The complexity of the travelling repairman problem. RAIRO-Theor. Inf. Appl. 20 (1), 79–87.

Archer, A., Blasiak, A., 2010. Improved approximation algorithms for the minimum latency problem via prize-collecting strolls. In: Proceedings of the 21th Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 429–447.

Archer, A., Williamson, D. P., 2003. Faster approximation algorithms for the minimum latency problem. In: Proceedings of the 40th Annual ACM-SIAM Symposium on Discrete algorithms. pp. 88–96.

Baldacci, R., Mingozzi, A., Roberti, R., 2011. New route relaxation and pricing strategies for the vehicle routing problem. Operations Research 59 (5), 1269–1283.

Bianco, L., Mingozzi, A., Ricciardelli, S., 1993. The traveling salesman problem with cumulative costs. Networks 23 (2), 81–91.

Bigras, L.-P., Gamache, M., Savard, G., 2008. The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times. Discrete Optimization 5 (4), 685–699.

Blum, A., Chalasanit, P., Pulleyblankt, B., Raghavan, P., Sudan, M., 1994. The minimum latency problem. In: Proceedings of the 26th Annual ACM Symposium on Theory of Computing. pp. 163–171.

Chaudhuri, K., Godfrey, B., Rao, S., Talwar, K., 2003. Paths, trees, and minimum latency tours. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2003. pp. 36–45.

Christofides, N., Mingozzi, A., Toth, P., 1981. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. Mathematical Programming 20 (1), 255–282.

Di Puglia Pugliese, L., Guerriero, F., 2012. A computational study of solution approaches for the resource constrained elementary shortest path problem. Annals of Operations Research 201 (1), 131–157.

Dror, M., 1994. Note on the complexity of the shortest path models for column generation in vrptw. Operations Research 42 (5), 977–978.

Fischetti, M., Laporte, G., Martello, S., 1993. The delivery man problem and cumulative matroids. Operations Research 41 (6), 1055–1064.

Fox, K. R., Gavish, B., Graves, S. C., 1980. Technical note-an n-constraint formulation of the (time-dependent) traveling salesman problem. Operations Research 28 (4), 1018–1021.

Garca, A., Jodr, P., Tejel, J., 2002. A note on the traveling repairman problem. Networks 40 (1), 27–31.

Godinho, M. T., Gouveia, L., Pesneau, P., 2014. Natural and extended formulations for the time-dependent traveling salesman problem. Discrete Applied Mathematics 164, Part 1, 138 – 153, combinatorial Optimization.

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 (3), 391–406.

Kullmann, O., 2009. Fundaments of branching heuristics. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (Eds.), Handbook of Satisfiability. Vol. 185 of Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 205–244.

Lucena, A., 1990. Time-dependent traveling salesman problem - The deliveryman case. Networks 20 (6), 753–763.

Lysgaard, J., Wøhlk, S., 2014. A branch-and-cut-and-price algorithm for the cumulative capacitated vehicle routing problem. European Journal of Operational Research 236 (3), 800 – 810.

Martinelli, R., Pecin, D., Poggi, M., 2014. Efficient elementary and restricted non-elementary route pricing. European Journal of Operational Research 239 (1), 102 – 111.

Méndez-Díaz, I., Zabala, P., Lucena, A., 2008. A new formulation for the traveling deliveryman problem. Discrete Applied Mathematics 156 (17), 3223–3237.

Mladenović, N., Urošević, D., Hanafi, S., 2013. Variable neighborhood search for the travelling deliveryman problem. 4OR 11 (1), 57–73.

Ngueveu, S., Prins, C., Wolfler Calvo, R., 2010. An effective memetic algorithm for the cumulative capacitated vehicle routing problem. Computers & Operations Research 37 (11), 1877–1885.

Nucamendi-Guillén, S., Martínez-Salazar, I., Angel-Bello, F., Moreno-Vega, J. M., 2016. A mixed integer formulation and an efficient metaheuristic procedure for the k-travelling repairmen problem. Journal of the Operational Research Society 67 (8), 1121–1134.

Pecin, D., Contardo, C., Desaulniers, G., Uchoa, E., 2017. New enhancements for the exact solution of the vehicle routing problem with time windows. To appear in INFORMS Journal on Computing.

Pecin, D., Pessoa, A., Poggi, M., Uchoa, E., 2016. Improved branch-cut-and-price for capacitated vehicle routing. Mathematical Programming Computation, 1–40.

Pessoa, A., Sadykov, R., Uchoa, E., Vanderbeck, F., 2013. In-Out Separation and Column Generation Stabilization by Dual Price Smoothing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 354–365.

Picard, J.-C., Queyranne, M., 1978. The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling. Operations Research 26 (1), 86–110.

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

Righini, G., Salani, M., May 2008. New dynamic programming algorithms for the resource constrained elementary shortest path problem. Networks 51 (3), 155–170.

Rivera, J. C., Afsar, H. M., Prins, C., 2016. Mathematical formulations and exact algorithm for the multitrip cumulative capacitated single-vehicle routing problem. European Journal of Operational Research 249 (1), 93 – 104.

Roberti, R., Mingozzi, A., 2014. Dynamic ng-path relaxation for the delivery man problem. Transportation Science 48 (3), 413–424.

Røpke, S., 2012. Branching decisions in branch-and-cut-and-price algorithms for vehicle routing problems. Presentation in Column Generation 2012.

Sahni, S., Gonzalez, T., July 1976. P-complete approximation problems. Journal of The ACM 23 (3), 555–565.

Salehipour, A., Sörensen, K., Goos, P., Bräysy, O., 2011. Efficient GRASP+VND and GRASP+VNS metaheuristics for the traveling repairman problem. 4OR 9 (2), 189–209.

Silva, M. M., Subramanian, A., Vidal, T., Ochi, L. S., 2012. A simple and effective metaheuristic for the minimum latency problem. European Journal of Operational Research 221 (3), 513 – 520.

Sitters, R., 2002. The minimum latency problem is NP-hard for weighted trees. In: Proceedings of the 9th International Conference on Integer Programming and Combinatorial Optimization, IPCO 2002. pp. 230–239.

Sze, J. F., Salhi, S., Wassan, N., 2017. The cumulative capacitated vehicle routing problem with min-sum and min-max objectives: An effective hybridisation of adaptive variable neighbourhood search and large neighbourhood search. Transportation Research Part B: Methodological 101, 162 – 184.

Tsitsiklis, J. N., 1992. Special cases of traveling salesman and repairman problems with time windows. Networks 22 (3), 263–282.

Van Eijl, C. A., 1995. A polyhedral approach to the delivery man problem. Tech. Rep. COSOR 95-19, Eindhoven University of Technology.

Vanderbeck, F., Sadykov, R., Tahiri, I., 2017. Bapcod — a generic branch-and-price code. Tech. rep.
URL https://realopt.bordeaux.inria.fr/?page_id=2