Numerically safe lower bounds for the Capacitated Vehicle Routing Problem *

Ricardo Fukasawa[1] and Laurent Poirrier[1]

[1]Department of Combinatorics & Optimization, , University of Waterloo, Waterloo, ON, Canada N2L 3G1 , {rfukasawa,lpoirrier}@uwaterloo.ca

May 13, 2016

Abstract

The resolution of integer programming problems is typically performed via branch-and-bound. Nodes of the branch-and-bound tree are pruned whenever the corresponding subproblem is proven not to contain a solution better than the best solution found so far. This is a key ingredient for achieving reasonable solution times. However, since subproblems are solved in floating-point arithmetic, numerical errors can occur, and may lead to inappropriate pruning. As a consequence, optimal solutions may be cut off. We propose several methods for avoiding this issue, in the special case of a branch-cut-and-price formulation for the Capacitated Vehicle Routing Problem (CVRP). The methods are based on constructing dual feasible solutions for the LP relaxations of the subproblems and obtaining, by weak duality, bounds on their objective function value. Such approaches have been proposed before for formulations with a small number of variables (dual constraints), but the problem becomes more complex when the number of variables is exponentially large, which is the case in consideration. We show that, in practice, besides being safe, our bounds are stronger than those usually employed, obtained with unsafe floating-point arithmetic plus some heuristic tolerance, all of this at a negligible computational cost. We also discuss some potential advantages and other uses of our safe bounds derivation.

1 Introduction

Mixed-integer programming (MIP) is a fundamental tool in operations research. Great progress has been made in MIP solvers like CPLEX (see [20]), to the point that MIPs are nowadays often used as subroutines for other problems (as an example see [13]). It is now common for even problems with hundreds of thousands of variables and constraints to be considered easy (see [23]).

However, most MIP solvers work with floating-point (FP) arithmetic, which implies that some of the decisions that are taken by the algorithms implemented within them can be incorrect. This is due to the intrinsic numerical errors that accompany FP arithmetic ([17]).

Some recent works ([30, 12]) have shown that indeed these numerical errors can lead to commercial solvers returning incorrect solutions. They highlight some of the potential consequences that these can have. To give an idea of one such issue, when correctness of an

*Fukasawa was supported by NSERC Discovery Grant RGPIN-05623. Poirrier was supported by Early Researcher Award ER11-08-174

approach relies on obtaining a truly optimal solution (for instance generation of local cuts, see [4]), then the whole approach may be invalid if one does not have a true optimal solution.

Several approaches have been proposed to deal with these computational errors in different components of a MIP solver. For instance, [2] discuss the exact solution of linear programs, [7] and [10] address numerical safety in the context of cutting planes, while [29] and [27] tackle the issue of obtaining safe bounds for MIPs. The works in [9] and [8] deal with the design of a full exact MIP solver.

In this work we focus our attention on obtaining numerically safe dual bounds for linear programming relaxations of a MIP formulation of the Capacitated Vehicle Routing Problem (CVRP). In other words, we want to obtain dual bounds that are valid even in the presence of the numerical errors in FP arithmetic.

The main difference between this work and the previous ones cited above is that all those approaches have been proposed for MIPs with a fixed (and not too big) number of variables. Such approaches, however, are not applicable to formulations that are solved via column generation inside a branch-and-price or branch-and-cut-and-price framework, which are the most successful types of formulations for the CVRP and several other routing problems. The only other work that we are aware of that deals with numerically safe bounds within a column generation-based framework is the one in [18] in the context of the graph coloring problem.

We note that, in principle, to obtain a lower bound for a MIP, all one needs to do is obtain an optimal solution to its LP relaxation, which can be provided by any floating point LP solver. However, as mentioned before, in presence of numerical errors, optimality may be hard to verify. Nonetheless, a lower bound may still be obtained if the dual solution returned is at least feasible for the dual of the LP relaxation. One problem is that even dual feasibility is not guaranteed for the solution given by the solver. But more importantly, in the context of column generation, what the LP solver returns is a potentially optimal dual solution to a restricted version of the problem and one must use it to find violated dual constraints (columns with negative reduced costs) to add those to the LP and iterate. This is what makes the problem of obtaining safe dual bounds particularly challenging in the context of column generation.

After this discussion, we highlight the main contributions of our work, which are the following:

1. We show how the numerically safe bounds proposed in [18] can be computed within the context of the CVRP.
2. In addition to the approach proposed in [18], we propose several different approaches to computing such safe bounds, that are particular to the structure of the formulation of the CVRP (but can be nicely adapted to different variants of the problem as well). We show that these new safe bounds are in practice better than the safe bounds proposed by [18].
3. We discuss other applications of obtaining those safe bounds and perform extensive computational experiments to draw empirical conclusions.

The outline of the paper is as follows. In Section 2 we introduce the CVRP and the formulation under consideration, including the pricing used for column generation. Section 3 discusses how to obtain valid lower bounds given an approximately feasible dual solution. While all the above sections deal with results assuming no numerical errors occur, they build the basic necessary results that enable us to obtain numerically safe bounds even in the presence of numerical errors. Section 4 discusses such results. Computational experiments are presented in Section 5, and a conclusion is left for Section 6.

2 The Capacitated Vehicle Routing Problem formulation

Let $G = (V, E)$ be an undirected graph with vertices $V = \{0, 1, \ldots, n\}$. Vertex 0 represents the *depot*, and each remaining vertex i represents a *client* with an associated positive demand

d_i. The set of client vertices is denoted by $V_+ = \{1, \ldots, n\}$. Each edge $e \in E$ has a positive length ℓ_e. Given G and two positive integers (K and C), the *Capacitated Vehicle Routing Problem* (CVRP) consists of finding routes for K vehicles satisfying the following constraints: (i) each route starts and ends at the depot, (ii) each client is visited by a single vehicle, and (iii) the total demand of all clients in any route is at most C. The goal is to minimize the sum of the lengths of all routes. This classical NP-hard problem is a natural generalization of the Traveling Salesman Problem (TSP), and has widespread application itself. The CVRP was first proposed by [11] and has received close attention from the optimization community since then.

We will use the following notation throughout the paper. Whenever we refer to an undirected edge from i to j, we use ij. For any $S \subseteq V$, we use $\delta(S) := \{uv \in E : u \in S, v \notin S\}$. Also, we use $\delta(v)$ to represent $\delta(\{v\})$.

Given a set $S \subseteq V_+$, let $d(S)$ be the sum of the demands of all vertices in S. Also, let $r(S) = \lceil d(S)/C \rceil$. A classical formulation for the CVRP uses x_e to represent the number of times a vehicle traverses edge e. Then the CVRP can be formulated as

$$\begin{aligned}
\min \ & \sum_{e \in E} l_e x_e & & \\
\text{s.t.} \ & \sum_{e \in \delta(i)} x_e = 2 & , \forall\, i \in V_+ & \quad (1)\\
& \sum_{e \in \delta(0)} x_e = 2 \cdot K & & \quad (2)\\
& \sum_{e \in \delta(S)} x_e \geq 2 \cdot r(S) & , \forall\, S \subseteq V_+ & \quad (3)\\
& x_e \leq u_e & , \forall\, e \in E & \\
& x_e \geq 0 & , \forall\, e \in E & \\
& x \in \mathbb{Z}^E, & &
\end{aligned}$$

where $u_e = 2$ if $e \in \delta(0)$ and $u_e = 1$ otherwise. Constraints (1) state that each client is visited once by some vehicle, whereas (2) states that K vehicles must leave and enter the depot. Constraints (3) are rounded capacity inequalities, which require all subsets to be served by enough vehicles.

We will actually consider a more general formulation by replacing the constraints (3) by a generic set of cuts $(\alpha^k)^T x \geq \alpha_o^k$, for all $k \in \kappa$, which can include the cuts (3), but also many other cuts including framed capacity, strengthened comb, multistar, partial multistar, generalized multistar and hypotour cuts (see [24, 26]), or even branching constraints.

Alternatively, a formulation with an exponential number of columns can be obtained by defining variables (columns) that correspond to q-routes. A *q-route* is a walk $i_0 i_1 i_2 \ldots i_L$ with $i_0 = i_L = 0$, $i_1, i_2, \ldots, i_{L-1} \in V_+$ and $\sum_{k=1}^{L-1} d_{i_k} \leq C$. Let $q_1, \ldots, q_p$ be the set of all possible q-routes. Moreover, let q_j^e be the number of times edge e appears in the q-route q_j. Then we can define a variable λ_j for every q-route q_j. By using the equation $x_e = \sum_{j=1}^{p} q_j^e \cdot \lambda_j$ and substituting x_e in the above formulation, we get the following Dantzig-Wolfe formulation:

$$\begin{aligned}
\min \quad & \textstyle\sum_{j=1}^{p} \left(\sum_{e \in E} l_e q_j^e \right) \lambda_j & & \\
\text{s.t.} \quad & \textstyle\sum_{j=1}^{p} \left(\sum_{e \in \delta(i)} q_j^e \right) \lambda_j = b_i & , \forall i \in V & \\
& \textstyle\sum_{j=1}^{p} \left(\sum_{e \in E} \alpha_e^k q_j^e \right) \lambda_j \geq \alpha_0^k & , \forall k \in \kappa & \\
& \textstyle\sum_{j=1}^{p} q_j^e \lambda_j \leq u_e & , \forall e \in E & \\
& \lambda_j \geq 0 & , \forall j \in \{1, \ldots, p\}, &
\end{aligned} \qquad \text{(DWM)}$$

where $b_i = 2K$ if i is the depot and $b_i = 2$ otherwise, and $u_e = 2$ if $e \in \delta(0)$ and $u_e = 1$ otherwise. Note that we omit here the integrality constraints for the sake of conciseness. The

details on how a solution to this LP can be embedded within a branch-and-bound framework to obtain an exact algorithm are described in [14].

The dual of (DWM) is

$$\begin{aligned}
\max \quad & \sum_{i\in V} b_i\omega_i + \sum_{k\in\kappa} \alpha_0^k\pi_k + \sum_{e\in E} u_e\rho_e \\
\text{s.t.} \quad & \sum_{i\in V}\left(\sum_{e\in\delta(i)} q_j^e\right)\omega_i + \sum_{k\in\kappa}\left(\sum_{e\in E}\alpha_e^k q_j^e\right)\pi_k + \sum_{e\in E} q_j^e\rho_e \le \left(\sum_{e\in E} l_e q_j^e\right), \quad \forall j\in\{1,\dots,p\} \qquad \text{(DW-DUAL)} \\
& \omega \text{ free}, \qquad \pi \ge 0, \qquad \rho \le 0.
\end{aligned}$$

Since (DWM) is an LP with an exponential number of variables, one needs to perform column generation. The pricing problem is solved by dynamic programming and is described in the next section.

We note that the most recent and most successful approaches to solving the CVRP (e.g. [3, 6, 28]) are based on using more complex structures instead of q-routes in (DWM). Here, we choose to use q-routes since:

1. it is simpler and easier to present in terms of notation,
2. the purpose of this article is not to solve larger instances of the CVRP, but to present a concept and development that can be applied to branch-cut-price approaches to it and other related problems, and
3. the results we present here can be easily generalized to those other more complex structures.

2.1 Dynamic programming

The pricing of q-routes is a thoroughly studied topic (see e.g. [21]). However, we describe its basics here since some of our results will be easier to understand based on this exposition. We also use it to discuss how to solve the pricing problem despite arithmetic errors later on.

Given a basis of (DWM), the task of the pricing step is to find columns with a negative reduced cost, or prove that all reduced costs are nonnegative. Equivalently, we want to find constraints of (DW-DUAL) that are violated by the dual solution associated to that basis, or prove that all constraints are satisfied. Besides bound constraints $\pi \ge 0$ and $\rho \le 0$, (DW-DUAL) has one constraint for every $j \in \{1,\dots,p\}$. Introducing slack variables $s_j \ge 0$, these constraints can be written

$$\sum_{i\in V}\left(\sum_{e\in\delta(i)} q_j^e\right)\omega_i + \sum_{k}\left(\sum_{e\in E}\alpha_e^k q_j^e\right)\pi_k + \sum_{e\in E} q_j^e\rho_e + s_j = \left(\sum_{e\in E} l_e q_j^e\right),$$

for all $j \in \{1,\dots,p\}$. Note that the slack s_j corresponds in (DWM) to the reduced cost of the variable λ_j. We can find the value of the smallest s_j by solving the problem $\min_{j\in\{1,\dots,p\}}\{s_j\}$, i.e.,

$$\begin{aligned}
& \min_{j\in\{1,\dots,p\}}\left\{\sum_{e\in E} l_e q_j^e - \sum_{i\in V}\left(\sum_{e\in\delta(i)} q_j^e\right)\omega_i - \sum_{k}\left(\sum_{e\in E}\alpha_e^k q_j^e\right)\pi_k - \sum_{e\in E} q_j^e\rho_e\right\} \\
= & \min_{j\in\{1,\dots,p\}}\left\{\sum_{e\in E}\left(l_e - \sum_{i:e\in\delta(i)}\omega_i - \sum_{k}\alpha_e^k\pi_k - \rho_e\right) q_j^e\right\} \\
= & \min_{j\in\{1,\dots,p\}}\left\{\sum_{e\in E}\bar{c}_e q_j^e\right\}, \qquad (4)
\end{aligned}$$

where we define the *reduced cost* $\bar{c}_e$ of an edge as

$$\bar{c}_e := l_e - \sum_{i:e\in\delta(i)} \omega_i - \sum_k \alpha_e^k \pi_k - \rho_e. \tag{5}$$

The expression (4) corresponds to finding a minimum cost q-route in a graph with edge costs $\bar{c}_e$, for $e \in E$. This problem is solved adopting a dynamic programming approach, storing, for every (i, d), a minimum cost route from the depot to node $i \in V_+$ with a vehicle carrying a load d. Routes are computed for increasing $d \in \{0, \ldots, C\}$. A minimum cost route is then found by taking a minimum over all (i, d), each time adding the return edge $i0$. In practice, all routes with a negative cost can be added as new columns.

Note that a similar approach can be used for different definitions of q-routes, like for instance if we forbid small cycles, or more generally for any state space relaxation of the elementary shortest path problem. Then, multiple routes are stored for each (i, d), along with some set U of vertices already visited (see [21]).

We denote by $-\Delta$ the cost of the q-route found above, i.e. the smallest (possibly most negative) reduced cost of a column of (DWM) for the current basis. We use this slightly counterintuitive notation for now to be compatible with later discussion where we will focus on Δ as being the maximum violation of a constraint in (DW-DUAL). Note that we can easily compute a slightly more fine-grained information. Specifically, for any node $i \in V_+$, we can compute a shortest q-route whose return edge is $i0$, by taking the minimum over all d for i fixed. Also, since the edge costs $\bar{c}_e$ are symmetric, a shortest q-route that ends with edge $i0$ is also a shortest q-route that starts with edge $0i$. In other words, we are computing

$$\Delta^e \quad := \quad - \min_{j\,:\,q_j^e \geq 1} \left\{ \sum_{f\in E} \bar{c}_f q_j^f \right\}, \tag{6}$$

for $e \in \delta(0)$, or in terms of the slack variables, $\Delta^e := -\min_{j:q_j^e\geq 1}\{\bar{s}_j\}$. We will see in Section 3.2 that this additional information can be exploited to compute stronger safe bounds.

3 Bound correctors

The pricing problem presented in Section 2.1 returns to us a most negative reduced cost column in (DWM). Equivalently, it provides a most violated constraint of (DW-DUAL) together with its violation $\Delta > 0$. In this section we will see how to use that information to directly obtain valid dual bounds for (DWM).

Consider a general primal-dual linear programming pair:

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Ax \geq b & \text{(P)} \\ & x \geq 0 & \end{array} \qquad\qquad \begin{array}{lll} \max & b^T y & \\ \text{s.t.} & A_j^T y \leq c_j, \forall j = 1, \ldots, p & \text{(D)} \\ & y \geq 0 & \end{array}$$

As mentioned in the introduction, a key requirement of any LP-based branch-and-bound method is to be able to obtain valid lower bounds for problems of type (P) by means of a dual feasible solution to (D). However, what modern LP solvers return as candidate dual variables may not even satsify that condition. In this section, we tackle the problem of obtaining lower bounds for (P) based on a solution $\bar{y}$ that is almost feasible for (D). This will be a key ingredient for us to be able to compute numerically safe lower bounds. Note, however, that throughout this section we will assume that all calculations are made without errors. Later on, we will describe how numerically safe lower bounds can be obtained even when arithmetic errors are present, based on the results of this section.

Formally, suppose that $\bar{y} \geq 0$ is such that $A_j^T \bar{y} \leq c_j + \Delta$ for some $\Delta \geq 0$ and all $j = 1, \ldots, p$. If $\Delta = 0$, then $b^T \bar{y}$ is a trivial lower bound for (P) and we are done. Now if $\Delta > 0$, since $\bar{y}$ is not necessarily dual feasible, then we cannot say that $b^T \bar{y}$ is a valid dual bound. However, under some conditions, one can construct from $\bar{y}$ a dual feasible solution y' and then compute a valid lower bound $b^T y'$.

We now present several different approaches to compute such valid lower bounds.

3.1 Scaling approach

[18] proposed the approach of setting $y' = \alpha \bar{y}$ for some $0 < \alpha < 1$ to guarantee that y' is dual feasible. One implicit requirement for this approach to work is that the constraints in the dual (D) are in $\leq$ format and that the right-hand-side is strictly positive for all dual constraints.

Suppose that, for some $\Delta > 0$, the dual vector $\bar{y} \geq 0$ is such that $A_j^T \bar{y} \leq c_j + \Delta$ for all $j = 1, \ldots, p$. Also, let $c_{min} = \min_{j=1,\ldots,p} c_j$. Then, we can set $y' = \frac{c_{min}}{c_{min}+\Delta} \bar{y}$ and we have, for all $j = 1, \ldots, p$, that

$$A_j^T y' = \frac{c_{min}}{c_{min} + \Delta} A_j^T \bar{y} \leq \frac{c_{min}}{c_{min} + \Delta}(c_j + \Delta) \leq \frac{c_j}{c_j + \Delta}(c_j + \Delta) = c_j.$$

The resulting dual bound will then be $\frac{c_{min}}{c_{min}+\Delta} b^T \bar{y}$. Note that the scaling factor does not affect the sign of any dual variable, so as long as the original dual variables have the correct signs, then the scaled dual variables also do. Using the same logic, this approach can be applied if there are other sign restrictions on the dual variables, provided that the original variables $\bar{y}$ satisfy those sign restrictions.

Note that for the case of the CVRP, (DW-DUAL) satisfies the required conditions that all constraints are in $\leq$ format and the right-hand side is strictly greater than zero. Furthermore, it is easy to compute c_{min} since the right-hand sides of the dual constraints are the costs of each q-route. The cheapest possible q-route consists of twice the cheapest edge out of the depot (going from the depot to that particular customer and back).

This leads to the following Proposition.

Proposition 1. *Let $(\bar{\omega}, \bar{\pi}, \bar{\rho}) \in \mathbb{R}^V \times \mathbb{R}_+^\kappa \times \mathbb{R}_-^E$ and let $\Delta \geq 0$ be the largest violation of any constraint of* (DW-DUAL) *by $(\bar{\omega}, \bar{\pi}, \bar{\rho})$. Then,*

$$\frac{2l_f}{2l_f + \Delta} \left(\sum_{i \in V} b_i \bar{\omega}_i + \sum_{k \in \kappa} \alpha_0^k \bar{\pi}_k + \sum_{e \in E} u_e \bar{\rho}_e \right) \tag{7}$$

is a valid lower bound for (DWM)*, where $f := \arg\min\{l_e : e \in \delta(0)\}$.*

3.2 Using the dual variables corresponding to primal bound constraints

A second approach for correcting dual infeasibilites was proposed in [1]. When we have upper bounds on all primal variables, the primal/dual problems become:

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & Ax \geq b \\ & -x \geq -u \\ & x \geq 0 \end{array} \tag{PB}$$

$$\begin{array}{ll} \max & b^T y - u^T \rho \\ \text{s.t.} & A_j^T y - \rho \leq c_j, \forall j = 1, \ldots, p \\ & y \geq 0 \end{array} \tag{DB}$$

The approach in [1] notes that one can simply check if each constraint is violated, and if so, increase the corresponding ρ_j by the violation amount. The price we pay for this adjustment

is a reduction in the value of the dual bound by u_j times the violation amount. In particular, when $u_j = 1$, which is the case for several combinatorial optimization problems, the violation penalty with this approach is moderate.

However, when the number of columns is extremely large, checking the violation of each dual constraint is impractical. As an alternative, if we know that all constraints are violated by at most $\Delta > 0$, then we can uniformly reduce all ρ_j elements, incurring a reduction in the value of the dual bound of $p\Delta$ (if all $u_j = 1$). This constitutes a significant loss when p is huge, as is the case for (DWM). Therefore, this approach is not desirable. We propose here a variant based on the particular structure of the CVRP.

Recall that, for our formulation, the dual constraints take following form:

$$\sum_{i \in V} \left(\sum_{e \in \delta(i)} q_j^e \right) \omega_i + \sum_k \left(\sum_{e \in E} \alpha_e^k q_j^e \right) \pi_k + \sum_{e \in E} q_j^e \rho_e \leq \left(\sum_{e \in E} l_e q_j^e \right).$$

We can use the ρ_e variables, which correspond to the upper bound constraints on the original x_{ij} variables, to correct for any violation of the dual constraints. However, the problem is slightly different here because ρ_e variables are not slacks. We propose to build a dual feasible solution (ω, π, ρ') where $\rho' = \rho + \tau$. If $\Delta > 0$ is an upper bound on the violation of the dual constraints, τ can be found by solving exactly the following problem:

$$\begin{array}{lll} \min & \sum_{e \in E} u_e \tau_e & \\ \text{s.t.} & \sum_{e \in E} q_j^e \tau_e \quad \geq \quad \Delta & , \forall j \in \{1, \ldots, p\} \\ & \tau \in \mathbb{R}_+^{|E|}. & \end{array} \tag{8}$$

Note that any feasible solution to (8) provides a dual feasible solution to (DW-DUAL), while the objective function of (8) yields one that minimizes the cost of the correction in the objective function of (DW-DUAL).

By letting $t := \tau / \Delta$ and rewriting (8) as Δ times

$$\begin{array}{lll} \min & \sum_{e \in E} u_e t_e & \\ \text{s.t.} & \sum_{e \in E} q_j^e t_e \quad \geq \quad 1 & , \forall j \in \{1, \ldots, p\} \\ & t \in \mathbb{R}_+^{|E|}, & \end{array} \tag{9}$$

we observe that (9) is a fractional covering problem: find a nonnegative cost assigned to each edge so that every q-route has total cost at least 1. As shown by Lemma (1), this problem can be solved in closed form.

Lemma 1. *The optimal objective function value of* (9) *is* n.

Proof. Proof. We start by constructing a feasible solution to (9). We let $t_e := \frac{1}{2}$ if $e \in \delta(0)$, and $t_e := 0$ otherwise. Since every q-route includes exactly two edges in $\delta(0)$, the constraints are satisfied. The cost of this solution is $\sum_{e \in \delta(0)} 2t_e = n$.

Next, we construct a feasible solution for the dual of (9). The dual of (9) is given by

$$\begin{array}{lll} \max & \sum_{j \in \{1, \ldots, p\}} \gamma_j & \\ \text{s.t.} & \sum_{j \in \{1, \ldots, p\}} q_j^e \gamma_j \quad \leq \quad u_e & , \forall e \in E \\ & \gamma_j \in \mathbb{R}_+^p. & \end{array} \tag{10}$$

The optimization problem (10) is fractional packing problem: assign costs to each q-route such that the total cost of any edge is at most u_e. We set $\gamma_j := 1$ if route j goes from the depot to some node i and immediately back to the depot. There are exactly n such q-routes. Otherwise, $\gamma_j := 0$. For an edge $e \notin \delta(0)$, the left-hand side of the constraint will be zero. For an edge $e \in \delta(0)$, we have $u_e = 0$, and exactly two q-routes with $\gamma_j = 1$ will use that edge, yielding $\sum_{j\in\{1,\dots,p\}} q_j^e \gamma_j = 2 \leq u_e$.

We now have a primal feasible solution and a dual feasible solution to (9) with value n. By strong duality, those solutions are optimal. □ □

We can thus construct a feasible (and optimal) solution to (8) with value $n\Delta$, which leads to the following proposition.

Proposition 2. *Let* $(\bar{\omega}, \bar{\pi}, \bar{\rho}) \in \mathbb{R}^V \times \mathbb{R}_+^\kappa \times \mathbb{R}_-^E$ *and let* $\Delta > 0$ *be the largest violation of any constraint of* (DW-DUAL) *by* $(\bar{\omega}, \bar{\pi}, \bar{\rho})$.

Then,

$$\left(\sum_{i\in V} b_i\bar{\omega}_i + \sum_{k\in\kappa} \alpha_0^k \bar{\pi}_k + \sum_{e\in E} u_e\bar{\rho}_e\right) - n\Delta \tag{11}$$

is a valid lower bound for (DWM).

3.2.1 Tightening the bound in (11):

By analyzing the dynamic programming algorithm in a bit more detail, one slightly tighter lower bound is obtained through the following refinement. Recall that in (6), we defined $\Delta^e := -\min_{j:q_j^e\geq 1}\{s_j\}$, for all $e \in \delta(0)$. That is, Δ^e is the largest violation of any dual constraint corresponding to a q-route that includes the edge e.

We now have a relaxed version of (8), that describes exactly the problem of finding a minimal cost corrector τ

$$\begin{aligned} \min \quad & \sum_{e\in E} u_e\tau_e \\ \text{s.t.} \quad & \sum_{e\in E} q_j^e\tau_e \quad \geq \quad -s_j \quad ,\forall j \in \{1,\dots,p\} \\ & \tau \in \mathbb{R}_+^{|E|}. \end{aligned} \tag{12}$$

Guided by Lemma 1, we derive the following result:

Lemma 2. *There exists a feasible solution to* (12) *with objective function value* $\sum_{e\in\delta(0)} \Delta^e$.

Proof. Proof. We construct a feasible solution to (12) by letting $\tau_e := \frac{1}{2}\Delta^e$ if $e \in \delta(0)$, and $\tau_e := 0$ otherwise. For all j, the left-hand side of the constraints becomes

$$\sum_{e\in\delta(0)} q_j^e \frac{1}{2}\Delta^e = \frac{1}{2}\left(\Delta^d + \Delta^f\right),$$

for some $d, f \in \delta(0)$ such that $d \neq f$ and $q_j^d = q_j^f = 1$, or $d = f$ and $q_j^d = q_j^f = 2$. Using the definition of Δ^e, we get

$$\frac{1}{2}\left(\Delta^d + \Delta^f\right) = \frac{1}{2}\left(-\min_{k:q_k^d\geq 1}\{s_k\} - \min_{k:q_j^f\geq 1}\{s_k\}\right) \geq \frac{1}{2}\left(-s_j - s_j\right) = -s_j,$$

showing the feasibility of the solution τ. Its cost is $\sum_{e\in\delta(0)} u_e\frac{1}{2}\Delta^e = \sum_{e\in\delta(0)}\Delta^e$. □ □

This leads to the following proposition.

Proposition 3. *Let* $(\bar{\omega}, \bar{\pi}, \bar{\rho}) \in \mathbb{R}^V \times \mathbb{R}^\kappa_+ \times \mathbb{R}^E_-$ *and let* $\Delta > 0$ *be the largest violation of any constraint of* (DW-DUAL) *by* $(\bar{\omega}, \bar{\pi}, \bar{\rho})$.

Then

$$\left(\sum_{i \in V} b_i \bar{\omega}_i + \sum_{k \in \kappa} \alpha_0^k \bar{\pi}_k + \sum_{e \in E} u_e \bar{\rho}_e\right) - \sum_{e \in \delta(0)} \Delta^e \tag{13}$$

is a valid lower bound for (DWM).

3.3 Using dual variables corresponding to degree constraints

Having developed the approach in the previous section, we considered the fact that we could potentially do a similar strategy with other sets of dual variables. However, we note that due to the highly generic nature of the constraints $(\alpha^k)^T x \geq \alpha_o^k$, using variables π for this would be quite challenging and probably not such a good idea.

On the other hand, the variables ω could potentially be used for this purpose. The problem, as before, is to find a subset $S \subseteq V$ of the variables ω such that $\sum_{i \in S} \sum_{e \in \delta(i)} q_j^e \geq 1$ for every q-route j. Now, if $0 \notin S$ and $S \neq V_+$, then there is always a q-route for which that sum is 0, namely the q-route 0-k-0 for $k \notin S$. This means that the only two possible choices are $S = V_+$ or $S = \{0\}$.

If $S = V_+$, we can note that $\sum_{i \in S} \sum_{e \in \delta(i)} q_j^e = 2$, since every q-route must enter and leave S exactly twice. So we can decrease w_i by $\Delta/2$ for all $i \in S$, which would incur a decrease in the dual bound value of $|S|\Delta = n\Delta$. On the other hand, if we take $S = \{0\}$, then we still have $\sum_{i \in S} \sum_{e \in \delta(i)} q_j^e = 2$, which means we can decrease w_0 by $\Delta/2$, which would incur a decrease in dual bound value of $\Delta K \leq \Delta n$, so $S = \{0\}$ is clearly the better choice.

As before, given that we know Δ, all we need is to add $-K\Delta$ to the value of a dual (not necessarily feasible) solution to obtain a safe and valid dual bound.

Proposition 4. *Let* $(\bar{\omega}, \bar{\pi}, \bar{\rho}) \in \mathbb{R}^V \times \mathbb{R}^\kappa_+ \times \mathbb{R}^E_-$ *and let* $\Delta > 0$ *be the largest violation of any constraint of* (DW-DUAL) *by* $(\bar{\omega}, \bar{\pi}, \bar{\rho})$.

Then

$$\left(\sum_{i \in V} b_i \bar{\omega}_i + \sum_{k \in \kappa} \alpha_0^k \bar{\pi}_k + \sum_{e \in E} u_e \bar{\rho}_e\right) - K\Delta \tag{14}$$

is a valid lower bound for (DWM).

3.4 Using a Lagrangian relaxation

It is well-known that the Dantzig-Wolfe reformulation is equivalent to doing Lagrangian relaxation (see, e.g. [5]). We explore one last way to obtain safe dual bounds by exploiting this equivalence.

Recall that we are interested in obtaining valid lower bounds for the following Dantzig-Wolfe formulation of the CVRP,

$$\begin{array}{rll} \min & \sum_{j=1}^p \left(\sum_{e \in E} l_e q_j^e\right) \lambda_j & \\ \text{s.t.} & \sum_{j=1}^p \lambda_j = K & \\ & \sum_{j=1}^p \left(\sum_{e \in \delta(i)} q_j^e\right) \lambda_j = 2 & , \forall i \in V_+ \\ & \sum_{j=1}^p \left(\sum_{e \in E} \alpha_e^k q_j^e\right) \lambda_j \geq \alpha_0^k & , \forall k \in \kappa \\ & \sum_{j=1}^p q_j^e \lambda_j \leq u_e & , \forall e \in E \\ & \lambda_j \geq 0 & , \forall j \in \{1, \ldots, p\}\,, \end{array} \tag{DWM}$$

in which we explicitly rewrote the first constraint of the original formulation, since we always have $\sum_{e\in\delta(0)} q_j^e = 2$.

By letting $\mu = \lambda/K$, we observe that (DWM) can be rewritten as

$$\begin{array}{rl} \min & \left(\sum_{j=1}^{p}\left(\sum_{e\in E} l_e K q_j^e\right)\mu_j\right) \\ \text{s.t.} & \sum_{j=1}^{p}\mu_j = 1 \\ & \sum_{j=1}^{p}\left(\sum_{e\in\delta(i)} K q_j^e\right)\mu_j = 2 \quad ,\forall i\in V_+ \\ & \sum_{j=1}^{p}\left(\sum_{e\in E}\alpha_e^k K q_j^e\right)\mu_j \geq \alpha_0^k \quad ,\forall k\in\kappa \\ & \sum_{j=1}^{p} K q_j^e \mu_j \leq u_e \quad ,\forall e\in E \\ & \mu_j \geq 0 \quad ,\forall j\in\{1,\ldots,p\}\,. \end{array} \tag{DWM'}$$

Each variable in (DWM') now corresponds to K times the edge incidence vector of a q-route q_j (which we represent by Kq_j). Then, by the equivalence of the Dantzig-Wolfe and Lagrangian relaxations, we know that the optimal value for (DWM') is equal to the optimal value for the following Lagrangian dual

$$z_L = \max_{(\omega,\pi,\rho)\in\mathbb{R}^V\times\mathbb{R}^\kappa_+\times\mathbb{R}^E_-}\left\{z_L(\omega,\pi,\rho)\right\}, \tag{15}$$

where

$$\begin{array}{rll} z_L(\omega,\pi,\rho) := & \min & \sum_{e\in E} l_e x_e + \sum_{i\in V}\left(b_i - \sum_{e\in\delta(i)} x_e\right)\omega_i + \sum_k\left(\alpha_0^k - \sum_{e\in E}\alpha_e^k x_e\right)\pi_k \\ & & \quad + \sum_{e\in E}(u_e - x_e)\,\rho_e \\ & \text{s.t.} & x\in\operatorname{conv}(\{Kq_j : j=1,\ldots,p\}). \end{array} \tag{16}$$

We then know that given any Lagrange multipliers ω,π,ρ such that $\pi\geq 0$ and $\rho\leq 0$, the value of the minimization problem in (16) gives a lower bound on the optimal value of (DWM). Taking the multipliers ω,π,ρ as constants in the minimization problem, (16) can be rewritten

$$\sum_{i\in V} b_i\omega_i + \sum_k \alpha_0^k\pi_k + \sum_{e\in E} u_e\rho_e + \begin{array}{ll} \min & \sum_{e\in E}\left(l_e - \sum_{i:e\in\delta(i)}\omega_i - \sum_k \alpha_e^k\pi_k - \rho_e\right)x_e \\ \text{s.t.} & x\in\operatorname{conv}(\{Kq_j : j=1,\ldots,p\}). \end{array} \tag{17}$$

Note that for every extreme point x of $\operatorname{conv}(\{Kq_j : j=1,\ldots,p\})$, we have that x/K is a q-route, i.e. $x/K = q_j$ for some $j\in\{1,\ldots,p\}$. The problem (17) thus becomes

$$\sum_{i\in V} b_i\omega_i + \sum_k \alpha_0^k\pi_k + \sum_{e\in E} u_e\rho_e + K\min_{j=1,\ldots,p}\left\{\sum_{e\in E}\left(l_e - \sum_{i:e\in\delta(i)}\omega_i - \sum_k \alpha_e^k\pi_k - \rho_e\right)q_j^e\right\}. \tag{18}$$

Since the minimum in (18) measures the largest violation of any dual constraint, that minimum is actually equal to Δ, so

$$z_L(\omega,\pi,\rho) = \sum_{i\in V} b_i\omega_i + \sum_k \alpha_0^k\pi_k + \sum_{e\in E} u_e\rho_e - K\Delta \tag{19}$$

Thus, the following result follows.

Proposition 5. *Let $(\bar\omega,\bar\pi,\bar\rho)\in\mathbb{R}^V\times\mathbb{R}^\kappa_+\times\mathbb{R}^E_-$ and let $\Delta>0$ be the largest violation of any constraint of* (DW-DUAL) *by $(\bar\omega,\bar\pi,\bar\rho)$.*

Then

$$\left(\sum_{i\in V} b_i\bar\omega_i + \sum_{k\in\kappa}\alpha_0^k\bar\pi_k + \sum_{e\in E} u_e\bar\rho_e\right) - K\Delta \tag{20}$$

is a valid lower bound for (DWM).

Proposition 5 is exactly equivalent to Proposition 4. This section simply provides a different approach for obtaining the same corrector.

4 Obtaining numerically safe lower bounds

Note that, up until now, all the dual bounds were derived assuming no numerical errors occurred. From this point on, we will discuss how to deal with numerical errors in a safe way.

4.1 Computationally safe dynamic programming

We start by noting that all the propositions in Section 3 only required an upper bound $\bar{\Delta}$ on the violation of any dual constraint. In this section we focus on how to obtain such an upper bound even in the presence of arithmetic errors.

Specifically, given $(\bar{\omega}, \bar{\pi}, \bar{\rho})$, for every $i \in V_+$, we need to find $\bar{\Delta}^i$ such that $\bar{s}_j \geq -\bar{\Delta}^i$, for all j such that $q^i_j \geq 1$. In order to ensure that $-\bar{\Delta}^i$ is a safe lower bound, any loss of precision has to be controlled throughout the computation: (a) first, bound constraints are enforced on $(\bar{\omega}, \bar{\pi}, \bar{\rho})$, (b) $\bar{c}_e$ is computed with floating-point rounding towards minus infinity, (c) $\bar{c}_e$ is then converted to an integer value for passing as an input to the dynamic programming procedure, rounding again towards minus infinity, (d) dynamic programming is performed in integer arithmetic with no overflows, (e) the result is converted back to floating-point, rounding towards minus infinity.

As noted in the point (d), the dynamic programming procedure is implemented in integer arithmetic, and thus incurs no loss of precision. The reduced cost $\bar{c}_e$ are converted to integers $\tilde{c}_e$ by multiplying them by a large constant M, then rounding down to the closest integer, i.e. $\tilde{c}_e := \lfloor M\bar{c}_e \rfloor$. As the value of M is increased, we reduce the rounding error of the latter operation, and obtain tighter lower bound $-\bar{\Delta}^e$ on $-\Delta^e$. However, we use standard integer arithmetic with a fixed bit-width (64 bits), so if M is too large, integer overflows can occur while performing dynamic programming. Let T be the largest integer such that both T and $-T$ are representable integers, and $\tilde{c}(\mathcal{Q})$ be the cost of a q-route $\mathcal{Q}$. In order to avoid overflows, we seek a large value M such that $|\tilde{c}(\mathcal{Q})| \leq T$ for any q-route $\mathcal{Q}$.

A first bound can be obtained by observing that any valid q-route has length at most $C+1$, because any node has demand at least 1 except for the depot. We thus know that

$$|\tilde{c}(\mathcal{Q})| \leq (C+1) \max_{e \in E} |\tilde{c}_e|. \tag{21}$$

Note that we compute the bound in floating-point arithmetic, so the multiplication in the right-hand side must be computed with rounding towards plus infinity.

A tighter bound can be obtained as follows. Assume that a q-route $\mathcal{Q}$ is composed of vertices $i_0, i_1, \ldots, i_L = i_0 = 0$. Let $e^k = i_{k-1}i_k$, for all $k = 1, \ldots, L$. The absolute value of its cost $\tilde{c}(\mathcal{Q})$ is given by

$$\begin{aligned} |\tilde{c}(\mathcal{Q})| \leq \sum_{k=1}^{L} |\tilde{c}_{e^k}| &= \sum_{k=1}^{L-1} d_{i_k} \frac{|\tilde{c}_{e^k}|}{d_{i_k}} + |\tilde{c}_{e^L}| \\ &\leq \sum_{k=1}^{L-1} d_{i_k} \cdot \max_{e \in E} \left\{ \frac{|\tilde{c}_e|}{d_{\min}(e)} \right\} + \max_{e \in \delta(0)} |\tilde{c}_e|. \end{aligned}$$

where

$$d_{\min}(e) := \min\{d_i : i \in V_+ \text{ and } e \in \delta(i)\},$$

i.e. $d_{\min}(e) := \min\{d_i, d_j\}$, for $e = ij \notin \delta(0)$ and $d_{\min}(e) := d_j$, for $e = 0j$. Recall that the total demand along the nodes visited by a q-route is at most C. Therefore, $\sum_{k=1}^{L-1} d_{i_k} \leq C$, and we obtain

$$|\tilde{c}(\mathcal{Q})| \leq C \max_{e \in E} \left\{ \frac{|\tilde{c}_e|}{d_{\min}(e)} \right\} + \max_{e \in \delta(0)} |\tilde{c}_e|. \tag{22}$$

Observe that the upper bound in (22) dominates the one in (21).

Taking into account that $|\lfloor\lambda\rfloor| < |\lambda| + 1$ for all $\lambda \in \mathbb{R}$, and that $d_{\min}(e) \geq 1$ for all $e \in E$, we replace $\tilde{c}_e$ with $\lfloor M\bar{c}_e \rfloor$ to obtain the expression

$$\begin{aligned}
|\tilde{c}(\mathcal{Q}, M)| &\leq C \max_{e\in E}\left\{\frac{|\lfloor M\bar{c}_e\rfloor|}{d_{\min}(e)}\right\} + \max_{e\in\delta^-(0)} |\lfloor M\bar{c}_e\rfloor| \\
&\leq C \max_{e\in E}\left\{\frac{M\,|\bar{c}_e| + 1}{d_{\min}(e)}\right\} + \max_{e\in\delta^-(0)} (M\,|\bar{c}_e| + 1) \\
&\leq C\left(M \max_{e\in E}\left\{\frac{|\bar{c}_e|}{d_{\min}(e)}\right\} + 1\right) + M \max_{e\in\delta^-(0)} |\bar{c}_e| + 1 \\
&\leq CM \max_{e\in E}\left\{\frac{|\bar{c}_e|}{d_{\min}(e)}\right\} + M \max_{e\in\delta^-(0)} |\bar{c}_e| + C + 1 \quad \leq \quad T.
\end{aligned}$$

It is thus safe to choose any M such that

$$M \leq (T - C - 1)/\left(C \max_{e\in E}\left\{\frac{|\bar{c}_e|}{d_{\min}(e)}\right\} + \max_{e\in\delta^-(0)} |\bar{c}_e|\right). \tag{23}$$

By computing M under the appropriate rounding modes (rounding up all operations on the denominator of the fraction and rounding down all other operations), one can guarantee the validity of such upper bound, even with FP arithmetic.

4.2 Tying it all together

The first immediate use of the correctors described in Propositions 1, 2, 3, 4 and 5 is, as previously stated, to compute numerically safe lower bounds for (DWM).

In order to do so, we use a numerically safe upper bound $\bar{\Delta}$ as described in Section 4.1 in place of Δ. Then, for individual arithmetic operations, we make sure to obtain appropriate bounds on the result: upper bounds if that result is subtracted or is in a denominator; lower bounds bounds otherwise. This is achieved by controlling the "rounding mode" of the processor, as detailed below in Section 4.3.

For example, to compute the bound (7), we first compute an upper bound on $2l_f + \bar{\Delta}$ by rounding errors up on the sum, then compute the lower bound on $\frac{2l_f}{2l_f+\bar{\Delta}}$ by performing the division while rounding errors down, and finally do all remaining sums and multiplication rounding down. In the end, we get a value guaranteed to be lower than the value in (7) and hence still a valid lower bound for (DWM). We refrain from detailing how safe bounds can be computed for all other propositions, since the operations are analogous. Using this approach, each of those bounds is easily computable in a numerically safe way to ensure it is valid, regardless of arithmetic errors.

In addition, there are other potential advantages to using those propositions. One well-known issue when dealing with column generation-type algorithms is dealing with tolerances to determine when the column generation process has converged. If these tolerances are not chosen well enough, one may end up with a suboptimal solution to (DWM) and thus with an invalid lower bound for the current node of the branch-and-bound tree. Furthermore, one may also be faced with *cycling*, a situation in which the pricing algorithm concludes that some columns have a negative reduced cost, but the LP solver decides that they should not enter the basis, due to rounding errors or tolerances. Then, the LP solver will not make any iteration and thus not change the dual variables, yielding exactly the same columns to be generated over and over. These issues create the need for a careful calibration of column generation tolerances, in conjunction with the tolerances of the LP solver. However, with the Propositions 1-5 in hand, one can simply abort column generation at any point and still get a valid dual bound, regardless of whether true convergence is attained.

This last point can be exploited even further and used to our advantage within a branch-and-bound context. For instance, even without the issues of tolerances, when performing

column generation, we need to wait until convergence occurs to be able to obtain a valid dual bound. However, it is often the case that the initial column generation iterations are very cheap (as they are usually performed by fast heuristics) and the final ones are more expensive. Therefore, if we can avoid the final column generation iterations, we would be able to save some time. In particular, if we compute the dual bound proposed in Propositions 1-5 and it tells us that the current node can be pruned, we can exit early; we do not need to run column generation until full convergence.

4.3 Implementation details

The result of an operation in floating-point arithmetic may not be *representable* in the chosen floating-point format, in our case 64-bits IEEE-754 double precision ([19]). Then, it is rounded to a nearby representable number. By default, it is rounded to the *nearest* representable floating-point value, which could be smaller or larger. But in order to perform safe arithmetic, we need to control the direction of the rounding. For example, we obtain safe lower bounds by always rounding towards minus infinity. Since C99, the standard C library provides a function `fesetround()` to change the rounding mode for the current thread ([22]). However, the support for changing the rounding mode is absent in LLVM ([25], as of version 3.5.0), and experimental in GCC (as of version 4.9.2). Specifically, the optimizer pass of GCC is not aware that the rounding mode can be changed. It thus allows arithmetic operations to be reordered around calls to `fesetround()` and other functions affecting floating-point arithmetic ([16, 15]). For instance, the following two snippets of code may (and, in our experience, will) compile into the same assembly output:

```
fesetround(FE_DOWNWARD);
x = 1.0 / a;
fesetround(FE_TONEAREST);
y = 1.0 / a;
```

```
fesetround(FE_DOWNWARD);
fesetround(FE_TONEAREST);
x = 1.0 / a;
y = 1.0 / a;
```

To the best of our knowledge, there is no reliable way to impose an ordering on arithmetic *operations* to the compiler. However, there exist a variety of tools to enforce an ordering on *memory access*. For instance, the following code

```
fesetround(FE_DOWNWARD);
compute_x_and_y();
fesetround(FE_TONEAREST);
```

behaves as expected as long as the function `compute_x_and_y()` is defined in a different compilation unit (thus does not risk being inlined). Indeed, the compiler must preserve the call order, in case `compute_x_and_y()` accesses memory that depends on the side effects of `fesetround(FE_DOWNWARD)`, or in case `fesetround(FE_TONEAREST)` has a dependency on the side effects of `compute_x_and_y()`. Since this method introduces the inconvenient need for having related code spread over different files, we describe several workarounds in Appendix A.

5 Computational experiments

Our code is based on the existing CVRP solver developed in [14], to which we added the safe bound computing methods described in Section 3. The only major modification to the previous code concerns the dynamic programming procedure. As noted in Section 4.1, we need dynamic programming to return exact results for an integer input. Consequently, the dynamic programming code was rewritten to use standard 64-bit integers in all computations. The linear programming problems are solved using CPLEX 12.6.

We present four different experiments. The first is intended to verify whether the new implementation of a safe dynamic programming procedure introduces any performance penalty. The second evaluates potential performance gains arising from the use of safe bounds. The third compares safe bounds with their unsafe counterparts, and tests whether unsafe or inefficient branching/pruning decisions were taken by the unsafe code. The fourth compares the strength of the three different bounds developed in Section 3.

5.1 Performance of safe dynamic programming

The dynamic programming routine implements the method from [21] for finding shortest q-routes, with elimination of short cycles. A traditional variant of the code uses floating point numbers for edge costs, and a safe variant uses integers. Two additional steps are necessary in its safe variant: conversion of the input edge costs from floating-point into integers, and the conversion the output q-route lengths from integer into floating-point values. In this section, we measure whether those conversions or the use of integers in computations presents any overhead. Both variants are run on an Intel Core i5 3210M processor with 8Gb of memory. The compiler is GCC 4.9.2 and the OS kernel is Linux 3.19.5. For this test, we use the same instances as [14], but limit to problems with less than 50 clients. This decision was made for practical reasons: we measure running times in this test, and running several instances in parallel would affect the accuracy of the measurements.

The results are presented in Table 1. Columns **cols** and **time** represent the total number of columns generated across all nodes of the BCP algorithm and the total running time, for either the *safe* or the *unsafe/traditional* variant of the code. The geometric means are reported for each column over all instances in the table. For some instances, we observe large variations between the two variants, both in number of columns generated and in running time. This is not surprising since slightly different solution paths may induce branch-and-bound trees of widely varying sizes. However, the numbers are very similar for most instances. In geometric mean, the safe code lead to the generation of 1051.1 columns per instance, while the unsafe code involved 1026.8 columns. The associated running times, also in geometric mean, are 5.008 seconds and 4.154 seconds, respectively. While it appears that the safe code is slightly slower on average, the difference in computational cost is certainly not prohibitive.

5.2 Safe bounds and column generation

The test in the previous section measures the compound effect of two differences between the safe and unsafe variants of our code. On the one hand, the safe variant has some overhead. On the other hand, as mentioned in Section 4.2, safe bounds let us interrupt column generation earlier in some cases. The test in this section intends to determine the effect of the latter. It is performed on the same computer and with the same instances as the previous test, and its results are shown in Table 2.

Here, whenever we determine that column generation could be interrupted early (i.e. before convergence), we start a timer and proceed until convergence is obtained. This way, we obtain an estimation of how many columns and how much time would be spared by interrupting column generation early, represented in the second and fourth columns of Table 2, respectively. Note that this is only an estimation, since columns that are avoided by early interruption may be needed later in other nodes of the branch-and-bound tree.

Note that in this experiment, column generation is interrupted only when the safe bound guarantees that the current node can be pruned. Our technique could let us stop the procedure at any earlier point with a valid (yet weaker) bound, in order to compute branch-and-bound nodes faster, at the risk of evaluating more nodes. However, this would require further fine tuning of our branch-and-cut-and-price framework, and is beyond the scope of this paper.

We observe that the early-exit strategy lets us avoid generating some columns in almost all instances. But the computing time associated with generating those columns remains marginal, around 1% for most instances. This is consistent with the corresponding gains in

		safe			unsafe				safe			unsafe	
Instance	nBB	cols	time (s)	nBB	cols	time (s)	Instance	nBB	cols	time (s)	nBB	cols	time (s)
A-n32-k5	1	817	0.405	1	819	0.378	B-n43-k6	13	1611	26.625	12	1858	30.614
A-n33-k5	1	715	0.303	1	715	0.251	B-n44-k7	1	1552	0.965	1	1634	1.049
A-n33-k6	5	666	6.721	4	695	4.434	B-n45-k5	7	2609	103.405	4	2545	69.317
A-n34-k5	3	949	3.627	3	943	3.152	B-n45-k6	5	1828	15.819	4	1847	14.720
A-n36-k5	11	1379	38.002	7	1197	15.279	E-n13-k4	1	51	0.027	1	51	0.012
A-n37-k5	32	3199	116.026	6	1668	29.394	E-n22-k4	1	240	0.088	1	240	0.073
A-n37-k6	50	1420	58.441	76	2673	93.742	E-n23-k3	1	925	5.029	1	943	5.576
A-n38-k5	19	2432	62.683	18	2760	55.337	E-n30-k3	1	2668	2.773	1	2583	2.915
A-n39-k5	7	1627	18.813	7	1668	17.514	E-n31-k7	3	631	3.045	3	619	3.067
A-n39-k6	9	1643	19.781	9	1480	19.069	E-n33-k4	1	1556	3.602	1	1534	3.555
A-n44-k6	3	1136	7.678	3	1126	6.903	F-n45-k4	1	6421	47.302	1	5840	39.677
A-n45-k6	70	3837	293.958	55	3426	176.184	P-n16-k8	4	65	0.170	4	65	0.167
A-n45-k7	50	2300	94.880	72	3563	154.411	P-n19-k2	1	506	0.231	1	506	0.228
A-n46-k7	2	1531	5.600	2	1572	11.525	P-n20-k2	3	625	4.113	3	592	4.065
A-n48-k7	168	5803	549.395	25	1733	60.152	P-n21-k2	1	520	0.227	1	520	0.269
B-n31-k5	1	1013	0.448	1	1013	0.400	P-n22-k2	2	615	5.369	2	615	5.344
B-n34-k5	14	1471	34.509	14	1298	39.387	P-n22-k8	1	109	0.039	1	109	0.027
B-n35-k5	1	1593	0.635	1	1655	0.623	P-n23-k8	1	143	0.032	1	143	0.029
B-n38-k6	7	1303	19.619	7	1284	19.590	P-n40-k5	2	1300	9.234	2	1300	9.194
B-n39-k5	2	2099	10.955	1	1978	1.271	P-n45-k5	21	1939	105.038	26	2367	177.198
B-n41-k6	3	1513	11.111	3	1452	12.186							
							geom. mean	3.8	1051.1	5.008	3.4	1026.8	4.154

Table 1: Computing time for safe and unsafe DP

the safe code being insufficient to compensate for the overhead, in Table 1. However, Table 2 confirms that the technique is useful in practice, although not much.

5.3 Safe bounds and unsafe bounds

The primary aim of this paper is to ensure that the pruning of branch-and-bound nodes is performed safely, i.e. without any risk of improper pruning, and without the need for arbitrary tolerances. We now examine the safe lower bounds that we obtain, in comparison to the unsafe bounds previously used in the code.

In order to obtain a fair comparison, for every branch-and-bound node, we first compute a bound z_{u} in floating-point arithmetic, then use the safe variant of the code to compute a safe bound z_{s}. For z_{u}, as is standard practice, a small value ε is subtracted from the LP bound computed in FP arithmetic, to account for FP rounding errors. For the safe bound, we use the maximum (i.e. strongest) of the safe bounds developed in Propositions 1-5.

In Table 3, we count for each of z_{s} and z_{u}, in how many nodes the bound was stronger. We break ties in favor of the unsafe bound since, as the safe bound incurs an overhead, we want to understand when our method is strictly preferable. Therefore, the number represented in the z_{s} column represents the number of nodes for which $z_{\mathrm{s}} > z_{\mathrm{u}}$, and the number in the z_{u} column represents the number of nodes for which $z_{\mathrm{s}} \leq z_{\mathrm{u}}$. The column "nBB" denotes the total number of branch-and-bound nodes. In the first group of columns, z_{u} is computed using a threshold of $\varepsilon = 10^{-6}$, the value used by [14]. In the second group of columns, z_{u} is computed with $\varepsilon = 10^{-9}$, in order to illustrate the importance of the choice of ε in an unsafe code. In some cases, the difference between z_{s} and z_{u} is large enough to result in different branching decisions being taken depending on which is used as a bound. If that is the case with $z_{\mathrm{s}} > z_{\mathrm{u}}$, it means that the unsafe code performed unnecessary branching. If it happens with $z_{\mathrm{u}} \geq z_{\mathrm{s}}$, the unsafe code might have performed unsafe pruning, although it may also be explained by the safe bound being too weak. The number of branch-and-bound nodes for which such event occurred is indicated between parenthesis when it is not zero.

The test was run on a computer with 48 AMD Opteron 6176 processor cores and 256Gb of memory. The compiler is GCC 4.4.7 and the OS kernel is Linux 2.6.32. For this test,

Instance	total cols	late cols	total time (s)	late time (s)	Instance	total cols	late cols	total time (s)	late time (s)
A-n32-k5	817	4	0.400	0.010	B-n43-k6	1611	336	26.322	0.932
A-n33-k5	715		0.301		B-n44-k7	1552	4	0.902	0.017
A-n33-k6	666	22	4.764	0.046	B-n45-k5	2609	145	93.644	0.601
A-n34-k5	949	22	3.662	0.072	B-n45-k6	1888	120	15.457	0.411
A-n36-k5	1610	48	46.791	0.189	E-n13-k4	51		0.024	
A-n37-k5	1614	98	18.500	0.289	E-n22-k4	240		0.073	
A-n37-k6	1391	383	56.109	0.963	E-n23-k3	925		5.042	
A-n38-k5	2445	161	50.018	0.460	E-n30-k3	2668	35	2.777	0.111
A-n39-k5	1627	167	18.162	0.314	E-n31-k7	631	16	3.031	0.043
A-n39-k6	1643	81	18.605	0.237	E-n33-k4	1556	10	3.403	0.204
A-n44-k6	1136	24	7.160	0.099	F-n45-k4	6421	35	45.725	1.526
A-n45-k6	3091	367	229.066	1.771	P-n16-k8	65		0.121	
A-n45-k7	2282	528	78.265	1.702	P-n19-k2	506		0.281	
A-n46-k7	1531	29	5.408	0.097	P-n20-k2	625	27	4.015	0.044
A-n48-k7	5049	1465	416.991	6.209	P-n21-k2	520		0.205	
B-n31-k5	1013	11	0.393	0.019	P-n22-k2	615	21	5.250	0.044
B-n34-k5	1484	189	34.382	0.470	P-n22-k8	109		0.028	
B-n35-k5	1593		0.601		P-n23-k8	143		0.024	
B-n38-k6	1428	212	23.984	0.521	P-n40-k5	1300	84	8.634	0.220
B-n39-k5	2099	46	10.816	0.120	P-n45-k5	1952	574	116.066	2.342
B-n41-k6	1513	39	10.731	0.135					

Table 2: Early-exit in column generation

we use all the instances from [14]. Some instances did not terminate within 500 hours: F-n135-k7.vrp, M-n200-k16.vrp, M-n200-k17.vrp, while others exhaust the available memory: B-n50-k8.vrp, B-n64-k9.vrp, B-n78-k10.vrp, E-n101-k8.vrp, M-n151-k12.vrp. Partial results are reported for those instances, on the branch-and-bound nodes that were processed.

Our results indicate that z_{s} is stronger than the z_{u} in the overwhelming majority of the nodes, despite being a safe bound. This is the case even when z_{u} is computed with the dangerously small tolerance $\varepsilon = 10^{-9}$. However, in our tests, the difference between z_{s} and z_{u} was never large enough to let the safe code prune nodes that the unsafe code could not prune. It must be noted that the objective function of most instances is integer, or can be made integer by multiplying it by a small power of ten. As a consequence, most LP bounds can be rounded up before being compared to the incumbent solution, rendering small errors harmless whenever the LP bound is far from an integral value.

On the other hand, with $\varepsilon = 10^{-6}$, z_{u} was rarely stronger than z_{s}, and also never led to the unsafe code performing dangerous pruning. It is thus unlikely that the optimality of solutions reported in [14] is affected by FP errors. The conclusions differ for $\varepsilon = 10^{-9}$ however, with unsafe pruning occurring in four distinct instances (A-n60-k9, B-n68-k9, B-n78-k10, M-n121-k7). This tends to validate the choice of 10^{-6} for ε.

5.4 Comparison of safe bounds

In this section, we compare the bound correctors developed in Section 3. We reuse the data generated in the experiment of the previous section, but this time count for the number of nodes in which each was the strongest. Table 4 shows such numbers. Again, "nBB" denotes the total number of branch-and-bound nodes. We denote by z_{rho} the bound resulting from Proposition 3, by z_{sc} the one from Proposition 1, and by z_{Lag} the one from Propositions 5 and 4. Whenever a bound is *strictly* stronger than the others for a number of nodes, that

Instance	nBB	$\varepsilon = 10^{-6}$ z_s	z_u	$\varepsilon = 10^{-9}$ z_s	z_u	Instance	nBB	$\varepsilon = 10^{-6}$ z_s	z_u	$\varepsilon = 10^{-9}$ z_s	z_u
A-n32-k5	0	0	0	0	0	B-n68-k9	4428	4428	0	4345	83 (1)
A-n33-k5	0	0	0	0	0	B-n78-k10	3449	3449	0	3185	264 (4)
A-n33-k6	3	3	0	3	0	E-n101-k14	4331	4329	2	4247	84
A-n34-k5	4	4	0	4	0	E-n101-k8	3190	3177	13	3088	102
A-n36-k5	6	6	0	6	0	E-n13-k4	0	0	0	0	0
A-n37-k5	4	4	0	4	0	E-n22-k4	0	0	0	0	0
A-n37-k6	62	62	0	62	0	E-n23-k3	0	0	0	0	0
A-n38-k5	30	30	0	30	0	E-n30-k3	0	0	0	0	0
A-n39-k5	4	4	0	4	0	E-n31-k7	1	1	0	1	0
A-n39-k6	8	8	0	8	0	E-n33-k4	0	0	0	0	0
A-n44-k6	2	2	0	2	0	E-n51-k5	2	2	0	2	0
A-n45-k6	10	10	0	10	0	E-n76-k10	1726	1726	0	1693	33
A-n45-k7	59	59	0	59	0	E-n76-k14	4381	4381	0	4356	25
A-n46-k7	1	1	0	1	0	E-n76-k7	1093	1092	1	1089	4
A-n48-k7	165	165	0	160	5	E-n76-k8	311	311	0	301	10
A-n53-k7	9	9	0	9	0	F-n135-k7	3	2	1	2	1
A-n54-k7	45	45	0	44	1	F-n45-k4	0	0	0	0	0
A-n55-k9	9	9	0	9	0	F-n72-k4	14	14	0	12	2
A-n60-k9	615	615	0	600	15 (1)	M-n101-k10	1	1	0	1	0
A-n61-k9	174	174	0	166	8	M-n121-k7	70	69	1	59	11 (2)
A-n62-k8	85	85	0	83	2	M-n151-k12	585	582	3	307	278
A-n63-k10	377	377	0	370	7	M-n200-k16	58	58	0	6	52
A-n63-k9	1083	1083	0	1035	48	M-n200-k17	554	550	4	94	460
A-n64-k9	304	304	0	304	0	P-n101-k4	30	30	0	29	1
A-n65-k9	10	10	0	10	0	P-n16-k8	3	3	0	3	0
A-n69-k9	517	516	1	513	4	P-n19-k2	0	0	0	0	0
A-n80-k10	84	84	0	80	4	P-n20-k2	2	2	0	2	0
B-n31-k5	0	0	0	0	0	P-n21-k2	0	0	0	0	0
B-n34-k5	10	10	0	10	0	P-n22-k2	1	1	0	1	0
B-n35-k5	0	0	0	0	0	P-n22-k8	0	0	0	0	0
B-n38-k6	3	3	0	3	0	P-n23-k8	0	0	0	0	0
B-n39-k5	0	0	0	0	0	P-n40-k5	1	1	0	1	0
B-n41-k6	2	2	0	2	0	P-n45-k5	23	23	0	23	0
B-n43-k6	12	12	0	12	0	P-n50-k10	172	172	0	172	0
B-n44-k7	0	0	0	0	0	P-n50-k7	4	4	0	4	0
B-n45-k5	6	6	0	6	0	P-n50-k8	264	264	0	264	0
B-n45-k6	2	2	0	2	0	P-n51-k10	22	22	0	22	0
B-n50-k7	1	1	0	1	0	P-n55-k10	950	950	0	945	5
B-n50-k8	10969	10968	1	10774	195	P-n55-k15	697	697	0	690	7
B-n51-k7	214	214	0	214	0	P-n55-k7	1110	1110	0	1104	6
B-n52-k7	8	8	0	8	0	P-n55-k8	104	104	0	104	0
B-n56-k7	3	3	0	3	0	P-n60-k10	40	40	0	40	0
B-n57-k7	11	11	0	10	1	P-n60-k15	20	20	0	20	0
B-n57-k9	7	7	0	7	0	P-n65-k10	23	23	0	23	0
B-n63-k10	14900	14895	5	14785	115	P-n70-k10	1475	1475	0	1447	28
B-n64-k9	7998	7998	0	151	7847	P-n76-k4	85	84	1	77	8
B-n66-k9	88	88	0	85	3	P-n76-k5	1670	1665	5	1645	25
B-n67-k10	142	142	0	135	7						

Table 3: Comparison of safe and unsafe bounds

number is indicated between parenthesis.

As shown in Table 4, z_{sc} was never strictly stronger than the other safe bounds in our experiments. The strongest bound was most often z_{rho}, but z_{Lag} was strictly stronger than z_{rho} in many cases. It is thus clear that none of z_{Lag} and z_{rho} dominates the other. They are both useful and complement each other in a safe code.

6 Conclusion

We propose several methods to obtain safe lower bounds for LP relaxations of the Capacitated Vehicle Routing Problem (CVRP). We then describe how these methods can be exploited in practice. In particular, since we solve CVRP instances by column generation, we describe a safe implementation of the dynamic programming procedure employed in the pricing step. Finally, we perform computations to compare the resulting bounds to those traditionally computed with floating-point arithmetic and arbitrary tolerances. The first method is based on a scaling approach proposed by [18] for graph coloring problems. However, the resulting bound is weak in practice. Three other methods are derived from ideas by [1] for the Travelling Salesman Problem, and we adapt them to accommodate for column generation in the CVRP. The last method is simply an alternate derivation of one of the bounds obtained previously, and is based on a specific Lagrangian relaxation of the CVRP formulation.

In practice, the bounds obtained with our methods are in most cases stronger than those computed with traditional floating-point arithmetic, in addition to being safe. Moreover, they can be computed even when column generation has not converged yet. As a result, we can interrupt column generation earlier in some cases. Our computations show that this technique yields a modest reduction in the cost of column generation, compensating in part for the small overhead of safe computations.

ACKNOWLEDGMENT: We would like to thank Laura Sanità and Ahmad Abdi for their assistance with the problem (9).

References

[1] David Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem, A computational study.* 2006.

[2] David Applegate, William J. Cook, Sanjeeb Dash, and Daniel Espinoza. Exact solutions to linear programming problems. *Operations Research Letters*, 35(6):693–699, 2007.

[3] Roberto Baldacci, Aristide Mingozzi, and Roberto Roberti. New route relaxation and pricing strategies for the vehicle routing problem. *Operations Research*, 59(5):1269–1283, 2011.

[4] Vasek Chvátal, William Cook, and Daniel Espinoza. Local cuts for mixed-integer programming. *Mathematical Programming Computation*, 5(2):171–200, 2013.

[5] Michele Conforti, Gerard Cornuéjols, and Giacomo Zambelli. *Integer Programming.* Springer, 2014.

[6] Claudio Contardo and Rafael Martinelli. A new exact algorithm for the multi-depot vehicle routing problem under capacity and route length constraints. *Discrete Optimization*, 12:129–146, 2014.

[7] William J. Cook, Sanjeeb Dash, Ricardo Fukasawa, and Marcos Goycoolea. Numerically safe gomory mixed-integer cuts. *INFORMS Journal of Computing*, 21(4):641–649, 2009.

[8] William J. Cook, Thorsten Koch, Daniel Steffy, and Kati Wolter. An exact rational mixed-integer programming solver. In Oktay Günlük and Gerhard Woeginger, editors, *IPCO, Lecture notes in computer science*, volume 6655, pages 104–116, 2011.

Instance	nBB	$z_{\text{best}} =$ z_{rho}	z_{sc}	z_{Lag}	Instance	nBB	$z_{\text{best}} =$ z_{rho}	z_{sc}	z_{Lag}
A-n32-k5	0	0	0	0	B-n68-k9	4428	(3806) 3857	0	(571) 622
A-n33-k5	0	0	0	0	B-n78-k10	3449	(2991) 3000	0	(449) 458
A-n33-k6	3	(1) 2	0	(1) 2	E-n101-k14	4331	(4152) 4162	0	(169) 179
A-n34-k5	4	(1) 1	0	(3) 3	E-n101-k8	3190	(2077) 2080	1	(1110) 1113
A-n36-k5	6	(3) 4	0	(2) 3	E-n13-k4	0	0	0	0
A-n37-k5	4	(2) 2	0	(2) 2	E-n22-k4	0	0	0	0
A-n37-k6	62	(29) 36	0	(26) 33	E-n23-k3	0	0	0	0
A-n38-k5	30	(16) 20	0	(10) 14	E-n30-k3	0	0	0	0
A-n39-k5	4	2	0	(2) 4	E-n31-k7	1	(1) 1	0	0
A-n39-k6	8	(5) 7	0	(1) 3	E-n33-k4	0	0	0	0
A-n44-k6	2	(1) 1	0	(1) 1	E-n51-k5	2	1	0	(1) 2
A-n45-k6	10	(6) 7	0	(3) 4	E-n76-k10	1726	(1396) 1412	0	(314) 330
A-n45-k7	59	(31) 37	0	(22) 28	E-n76-k14	4381	(4143) 4152	0	(229) 238
A-n46-k7	1	(1) 1	0	0	E-n76-k7	1093	(705) 719	0	(374) 388
A-n48-k7	165	(145) 151	0	(14) 20	E-n76-k8	311	(222) 234	0	(77) 89
A-n53-k7	9	(5) 6	0	(3) 4	F-n135-k7	3	(1) 1	0	(2) 2
A-n54-k7	45	(31) 40	1	(5) 14	F-n45-k4	0	0	0	0
A-n55-k9	9	(4) 7	0	(2) 5	F-n72-k4	14	0	0	(14) 14
A-n60-k9	615	(488) 493	0	(122) 127	M-n101-k10	1	(1) 1	0	0
A-n61-k9	174	(145) 152	0	(22) 29	M-n121-k7	70	(33) 35	0	(35) 37
A-n62-k8	85	(64) 69	0	(16) 21	M-n151-k12	585	(503) 504	0	(81) 82
A-n63-k10	377	(299) 301	0	(76) 78	M-n200-k16	58	(54) 54	0	(4) 4
A-n63-k9	1083	(937) 941	0	(142) 146	M-n200-k17	554	(538) 538	0	(16) 16
A-n64-k9	304	(249) 255	1	(49) 55	P-n101-k4	30	(2) 2	0	(28) 28
A-n65-k9	10	(8) 9	0	(1) 2	P-n16-k8	3	(1) 3	0	2
A-n69-k9	517	(432) 443	0	(74) 85	P-n19-k2	0	0	0	0
A-n80-k10	84	(75) 78	0	(6) 9	P-n20-k2	2	1	0	(1) 2
B-n31-k5	0	0	0	0	P-n21-k2	0	0	0	0
B-n34-k5	10	(3) 3	0	(7) 7	P-n22-k2	1	0	0	(1) 1
B-n35-k5	0	0	0	0	P-n22-k8	0	0	0	0
B-n38-k6	3	(2) 2	0	(1) 1	P-n23-k8	0	0	0	0
B-n39-k5	0	0	0	0	P-n40-k5	1	0	0	(1) 1
B-n41-k6	2	(2) 2	0	0	P-n45-k5	23	(15) 15	0	(8) 8
B-n43-k6	12	(1) 4	0	(8) 11	P-n50-k10	172	(151) 164	0	(8) 21
B-n44-k7	0	0	0	0	P-n50-k7	4	(2) 3	0	(1) 2
B-n45-k5	6	(2) 2	0	(4) 4	P-n50-k8	264	(231) 242	1	(22) 33
B-n45-k6	2	1	0	(1) 2	P-n51-k10	22	(20) 20	0	(2) 2
B-n50-k7	1	0	0	(1) 1	P-n55-k10	950	(889) 916	0	(34) 61
B-n50-k8	10969	(7821) 7898	1	(3071) 3148	P-n55-k15	697	(685) 691	0	(6) 12
B-n51-k7	214	(137) 144	0	(70) 77	P-n55-k7	1110	(869) 900	0	(210) 241
B-n52-k7	8	(4) 5	0	(3) 4	P-n55-k8	104	(83) 95	0	(9) 21
B-n56-k7	3	(3) 3	0	0	P-n60-k10	40	(30) 36	0	(4) 10
B-n57-k7	11	(8) 8	0	(3) 3	P-n60-k15	20	(12) 18	0	(2) 8
B-n57-k9	7	(7) 7	0	0	P-n65-k10	23	(16) 19	0	(4) 7
B-n63-k10	14900	(13908) 13977	0	(923) 992	P-n70-k10	1475	(1346) 1364	0	(111) 129
B-n64-k9	7998	(7055) 7061	0	(937) 943	P-n76-k4	85	(13) 14	0	(71) 72
B-n66-k9	88	(76) 76	0	(12) 12	P-n76-k5	1670	(669) 686	1	(984) 1001
B-n67-k10	142	(115) 118	0	(24) 27					

Table 4: Strongest safe bounds

[9] William J. Cook, Thorsten Koch, Daniel Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5:305–344, 2013.

[10] Gerard Cornuéjols, François Margot, and Giacomo Nannicini. On the safety of gomory cut generators. *Mathematical Programming Computation*, 5:345–395, 2013.

[11] George Dantzig and John Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 1959.

[12] Daniel G. Espinoza. *On Linear Programming, Integer Programming and Cutting Planes*. PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, March 2006.

[13] Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical Programming*, 98(1–3):23–47, 2003.

[14] Ricardo Fukasawa, Humberto Longo, Jens Lysgaard, Marcus Poggi de Aragão, Marcelo Reis, Eduardo Uchoa, and Renato F. Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical Programming*, 106(3):491–511, 2006.

[15] GCC Bugzilla. Optimization breaks floating point exception flag reading. Accessed April 15, 2015. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=29186`, 2006.

[16] GCC Bugzilla. Optimization generates incorrect code with -frounding-math option. Accessed April 15, 2015. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678`, 2008.

[17] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[18] Stephan Held, William J. Cook, and Edward C. Sewell. Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation*, 4:363–381, 2012.

[19] IEEE. *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Standard 754–1985.

[20] IBM ILOG. CPLEX 12.6, 2013. http://www.ilog.com/products/cplex.

[21] Stefan Irnich and Daniel Villeneuve. The shortest-path problem with resource constraints and k-cycle elimination for $k \geq 3$. *INFORMS Journal on Computing*, 18(3):391–406, January 2006.

[22] ISO/IEC. *Programming languages – C*. International Organization for Standardization, 1999. Standard ISO/IEC 9899:1999.

[23] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[24] Adam Letchford, Richard Eglese, and Jens Lysgaard. Multistars, partial multistars and the capacitated vehicle routing problem. *Mathematical Programming*, 94:21–40, 2002.

[25] LLVM Bugzilla. Clang/LLVM don't support C99 FP rounding mode pragmas. Accessed May 24, 2015. `https://llvm.org/bugs/show_bug.cgi?id=8100`, 2010.

[26] Jens Lysgaard, Adam Letchford, and Richard Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming*, 100:423–445, 2004.

[27] Arnold Neumaier and Oleg Shcherbina. Safe bounds in linear and mixed-integer lineear programming. *Mathematical Programming*, 99:283–296, 2004.

[28] Diego Pecin, Artur Pessoa, Marcus Poggi, and Eduardo Uchoa. Improved branch-cut-and-price for capacitated vehicle routing. In *Integer programming and combinatorial optimization*, pages 393–403. Springer, 2014.

[29] Daniel Steffy and Kati Wolter. Valid linear programming bounds for exact mixed-integer programming. *INFORMS Journal on Computing*, 25(2):271–284, 2013.

[30] Daniel E. Steffy. *Topics in exact precision mathematical programming*. PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, 2008.

A Implementation details for setting rounding mode

Technically, the workaround developed in Section 4.3 amounts to introducing a *compiler memory barrier* between the call to `fesetround()` and the subsequent arithmetic operations to enforce a specific ordering. In C, function calls act as compiler memory barriers as long as the compiler cannot see the function definition. They present two advantages over manually inserting a memory barrier (e.g. with `asm volatile("" : : : "memory");` in GCC). Firstly, function calls are portable and specified by the standard. Secondly, memory barriers have no impact on local variables (even if they are stored on the stack) unless a pointer to them is taken previously in the execution flow. Implementing arithmetic operations in a separate function ensures that no local variables could escape our memory ordering constraints.

Several alternatives to a function call are possible.

One may create a no-operation function `touch(double *)` defined outside of the compilation unit, to which we pass pointers to the local variables involved. Equivalently, we can create a wrapper `fesetround_wrapper()` for `fesetround()` that takes a variable number of arguments, to which we pass those same pointers.

```
int touch(double *v)
{
}
```

```
int fesetround_wrapper(int mode, ...)
{
        return(fesetround(mode));
}
```

In both cases, the function does not modify its arguments, but this method forces the compiler to store them to memory. Since the call to `fesetround()` acts as a compiler memory barrier, it produces the desired outcome. In the case of the wrapper function, the overhead could be limited to the storage of local variables to memory by inlining the code of `fesetround()` in the wrapper function. Examples follow.

```
fesetround(FE_DOWNWARD);
touch(&a)
x = 1.0 / a;
touch(&x)
fesetround(FE_TONEAREST);
touch(&a)
y = 1.0 / a;
```

```
fesetround_wrapper(FE_DOWNWARD, &a);
x = 1.0 / a;
fesetround_wrapper(FE_TONEAREST, &x, &a);
y = 1.0 / a;
```

A second possibility is to use an identity function `self(double)` which returns its argument. In this case, local variables may not need to be stored to memory if the ABI permits argument passing through registers (as is the case in x86_64), but the function calls to `self()` cannot be avoided.

```
fesetround(FE_DOWNWARD);
x = self(1.0 / self(a));
fesetround(FE_TONEAREST);
y = self(1.0 / self(a));
```

Finally we can create a `safe_double` class and define all its operator methods in a separate compilation unit. This approach has most overhead (constructors and destructors will be called as well as operator methods), but it requires less caution from its user.

```
safe_double b = a;
fesetround(FE_DOWNWARD);
safe_double x = 1.0 / b;
fesetround(FE_TONEAREST);
safe_double y = 1.0 / b;
```

It is not perfect however, as care must be taken with implicit casts. In the following example indeed,

```
fesetround(FE_DOWNWARD);
safe_double x = 1.0 / (double)a;
```

the division may be performed before the call to `fesetround()`, since only the result of `1.0 / (double)a` is cast to `safe_double`.