CDMO project

Armando Renzullo: armando.renzullo@studio.unibo.it,
Cosimo Russo: cosimoemanuele.russo@studio.unibo.it,
Tommaso Perniola: tommaso.perniola@studio.unibo.it

June 2025

1 Introduction

Efficient and balanced allocation of delivery workloads is a cornerstone of modern logistics, especially in dense urban environments where fairness among couriers and adherence to service level agreements are critical. The challenge addressed in this work is a variant of the classical **vehicle routing problem** (**VRP**) in which the primary goal is not to minimise total cost, but rather to ensure that the longest route assigned to any courier is as short as possible. Our work develops a deterministic, end-to-end solution pipeline that handles these requirements by integrating lightweight heuristics, structural insights, and solver strategies, allowing us to consistently reach near-optimal or optimal solutions within a strict runtime budget. The following sections detail the design principles, algorithmic components, and empirical evaluation of the proposed solution.

All experiments were conducted in a Docker environment with a 3 GB memory limit, running on the same machine with a 2 GHz allocation to the container.

1.1 Model Foundations

This subsection covers the main ingredients shared by all models in this report. We describe the objective function, the preprocessing steps used to simplify instances, the bounding techniques that guide the solver, and the symmetry-breaking constraints that reduce redundant solutions. These elements form the baseline for all model variations and won't be repeated in later sections.

1.1.1 Objective function

We tackle a min–max vehicle-routing problem that seeks to minimise the longest (i.e., maximum) tour length among all couriers. Formally, let d_c denote the total distance travelled by courier $c \in C$. The optimization goal is:

$$\min_{x,f,\ldots} [\max_{c \in C}(d_c)]$$

Thus, the objective function is a **min–max criterion**:

$$\min z \;\text{ s.t. } d_c \leq z \;\forall c \in C$$

1.1.2 Preprocessing steps

To improve the efficiency of our approach, we applied a series of preprocessing and simplification steps. These included establishing tight upper and lower bounds for key variables, eliminating non-contributing arcs (such as self-loops), and ordering instance data to favour efficient model construction. The following sections describe some of these techniques in detail.

1.1.3 Bounding Heuristics: Lower and Upper Bounds

To guide the solver and restrict the feasible search space, we used a **radial lower bound** and a **warm-start-derived upper bound**, both efficiently computable from instance data.

Lower Bound. The lower bound is computed as the cost of the longest round trip from the depot to any customer and back:

$$\text{LB}_{\text{radial}} = \max_{j \in V} \left(D_{dj} + D_{jd} \right), \tag{1}$$

where D_{ij} is the travel cost between node i and node j, and d is the depot node. This value reflects the minimal unavoidable travel distance any courier must perform.

Upper Bound. To provide a feasible initial solution and determine an effective arc filtering threshold, we compute an upper bound using the best available warm-start solution. This upper bound is derived as follows:

1. **OR-Tools Heuristic**: We first attempt to generate a feasible seed using Google OR-Tools, invoking its routing solver with a time cap and using the PATH CHEAPEST ARC strategy. If successful, the resulting routes are used to compute:

$$\text{UB}_{\text{seed}} = \max_{c \in C} \left(\sum_{(i,j) \in \text{route}_c} D_{ij} \right).$$

2. **Clarke–Wright Fallback.** If OR-Tools fails (due to solver timeout or infeasibility), a Clarke–Wright savings heuristic is used to produce a guaranteed feasible set of routes.

The best incumbent is injected into the solver via a warm start when possible (i.e., in MIP). Simultaneously, the upper bound value is used to shrink the search space.

These simple bounding procedures have proven instrumental in reducing computation time and ensuring solver stability on larger instances.

1.1.4 Triangle Inequality and Symmetry

We safely assume that the triangle inequality always holds for all distances. This property ensures that, for any three points i, j, and k, the direct distance from i to k is not greater than the sum of the distances from i to j and from j to k:

$$D[i,k] \leq D[i,j] + D[j,k] \tag{2}$$

1.1.5 Courier Ordering

As part of the preprocessing, couriers are sorted in non-increasing order of their load capacities before being passed to the solver. This ordering is not only consistent with the structure of the instance files, but also enables effective symmetry breaking. In particular, by enforcing a non-increasing load order through constraints of the form:

$$\texttt{load}_c \geq \texttt{load}_{c+1} \quad \forall c \in 1, \ldots, m-1, \tag{3}$$

we eliminate symmetric solutions arising from permuting couriers with identical capacity. This reduces the size of the search tree and leads to faster convergence without affecting solution optimality.

1.2 Project execution overview

Completion time and workload division The project spanned three months with intermittent work sessions. Experimental tasks involving different solvers and strategies were collectively discussed before assigning specific responsibilities to each team member. Any challenges or issues encountered were reviewed and addressed by the entire group to ensure comprehensive input and oversight. The development of each model was a collaborative effort, though each member contributed significantly to specific models:

- CP: Contributions were primarily from Renzullo.
- SAT: Each team member contributed equally.
- SMT: Perniola focused on this part.
- MIP: Russo was primarily responsible for developing this model.

Main difficulties The primary challenges revolved around the SAT workflow, particularly in developing an effective and efficient model capable of solving a range of instances. Additionally, the MiniZinc models were critical, as they laid the groundwork for subsequent models. The team's collaboration was essential in overcoming these difficulties and advancing the project.

2 CP Model

Using the MiniZinc language, we developed a constraint programming model which was subsequently tested with two different solvers: Gecode and Chuffed. We experimented with various problem encodings, particularly comparing those that utilize symmetry-breaking constraints with those that do not and we tested our models with different solving techniques, finally finding the best for our goal, which is optimization.

2.1 Decision Variables

The model is modelled as a Successor-based routing model. It is developed with the following decision variables:

- $x_{i,j}$: A decision variable representing the path of courier i. Specifically, it indicates the node that courier i will visit immediately after node j. The domain of $x_{i,j}$ is $\{1, 2, \ldots, n+1\}$.
  - If $x_{i,j} = j$, then courier i does not visit node j.
  - If $x_{i,n+1} \neq n+1$, then courier i must return to the depot after visiting node $n+1$.
- *max_distance_per_courier*$_i$: A variable representing the maximum distance traveled by courier i. This variable is defined over the range [lb ub], where total is the sum of all distances in the distance matrix D.
- *max_distance*: A decision variable representing the maximum distance among all couriers, which serves as the objective function.

2.2 Objective Function

The objective function is constrained by the equality constraint defining *max_distance* as the maximum value among all *max_distance_per_courier*[i] values. This is detailed in Section 1.1.1.

2.3 Constraints

The model includes the following constraints:

1. **Subcircuits:** $\forall i \in \{1, \ldots, m\}$, `subcircuit`$[x_{i,j}]_{j=1}^{n+1}$. Ensures each courier follows a valid tour covering assigned nodes [8].
2. **Unique delivery:** $\forall j \in \{1, \ldots, n\}, \sum_{i=1}^{m} [x_{i,j} \neq j] = 1$. Each package is delivered exactly once.
3. **Depot constraint:** $\forall i, x_{i,n+1} \neq n+1$, and $\sum_{j=1}^{n} [x_{i,j} = n+1] = 1$. Couriers must start and return to the depot.

4. **Capacity limit:** $\sum_{j=1}^{n}(x_{i,j} \neq j \rightarrow s_j) \leq l_i$. The load carried cannot exceed the courier's capacity.

5. **Distance limit:** $\sum_{j=1}^{n+1} D_{j,x_{i,j}} \leq \text{max\_distance}_i$. Limits each courier's travel distance.

Implied Constraints

1. $\sum_{j=1}^{n}[x[i,j] \neq j] \leq \lceil \frac{n}{m} \rceil + 1$ ensures workload balancing.

2. `alldifferent`$[x[i, n+1]]_{i=1}^{m}$ avoids overlapping return nodes.

Symmetry-breaking

1. **Lex-order:** If $l[i] = l[z]$, enforce lex_lesseq$([x[z,k]],[x[i,k]])$ to reduce equivalent permutations.

2. **Load pruning:** If $l[i] \geq l[z]$, then

$$\sum_{j}(x[i,j] \neq j \rightarrow s[j]) \geq \sum_{j}(x[z,j] \neq j \rightarrow s[j])$$

to prevent high-capacity couriers from being underloaded.

2.4 Validation

The model is implemented in MiniZinc and it has been tested using two different solvers, Gecode and Chuffed. In addition, it has been made a comparison between different search strategies over the integer space.

Experimental Design The most effective search strategy turned out to be the one which implements:

- On x: "domWdeg" as variable selection heuristic with "indomain_random" as value selection heuristic.
- "restart_luby" as a restart strategy, in order to periodically restart the search.
- a "relax and reconstruct" heuristic over y to improve the efficiency of finding solutions by partially relaxing the current assignment and then reconstructing it.

We implemented a restart strategy and a "relax and reconstruct" heuristic to explore different regions of the search space, helping to escape local optima and improve search diversity, making it particularly useful for large instances. Being this search strategy available for Gecode it has shown way better results than Chuffed.

Instance	Gecode with SB	Gecode w/out SB	Chuffed with SB
1	15	15	**14**
2	**226**	**226**	**226**
3	**12**	**12**	**12**
4	**220**	**220**	**220**
5	**206**	**206**	**206**
6	**322**	**322**	**322**
7	**167**	**167**	**167**
8	**186**	**186**	**186**
9	**436**	**436**	**436**
10	**244**	**244**	**244**
11	512	515	851
12	**346**	**346**	522
13	518	508	918
14	716	738	-
15	761	611	-
16	**286**	**286**	**286**
17	934	875	-
18	559	601	1219
19	343	341	454
20	786	842	-
21	534	529	973

Table 1: While symmetry breaking constraints reduce the solution space, results without them could be slightly better. This is mainly due to the significantly smaller number of constraints, which leads to faster exploration—especially when using `restart_luby` as the search strategy.

3 SAT Model

We encode the courier routing problem as a pure SAT model using the Z3 SMT solver. All constraints are formulated exclusively with Boolean variables and propositional logic, avoiding pseudo-Boolean or high-level arithmetic constructs.

3.1 Decision Variables

The model uses the following Boolean variables:

- ***path*** $p_{c,p,s}$: true iff courier c visits node p at step s, encoding the route order.
- ***cw***: true iff courier c carries package p, representing assignment of packages to couriers.
- ***cl***, ***cd***: binary vectors encoding cumulative load and total traveled distance per courier, respectively, enabling bit-level arithmetic reasoning.
- ***c_step_d***: encodes the distance contribution from individual transitions in the route.
- ***max_d***: a binary vector representing the maximum distance traveled by any courier, serving as the optimization target.

Weights s_p and capacities $l[c]$ are pre-encoded in binary to support efficient bitwise operations within the SAT encoding.

3.2 Objective Function

The goal is to minimize the maximum traveled distance among all couriers:

$$\min \max_c \mathrm{cd}[c].$$

This is enforced by constraining max_d to be greater or equal than each courier's total distance cd[c], and by forcing equality with the distance of at least one courier c^*.

3.3 Constraints

All problem constraints are encoded as Boolean formulas:

1. **Unique Step Assignment:** Each courier visits exactly one node at each step s, ensuring valid routing sequences.
2. **Unique Package Assignment:** Every package (excluding the depot) is assigned to exactly one courier and one step, avoiding duplicate deliveries.
3. **Depot Fixation:** Routes start and end at the depot node, ensuring proper dispatch and return.

4. **Load Calculation:** For each courier c, the load $\text{cl}[c]$ is computed as the sum of the weights of assigned packages:

$$\text{cl}[c] = \sum_{p} \text{If}(\text{cw}[c][p], s_p, 0),$$

and constrained to not exceed the courier's capacity $l[c]$.

5. **Distance Calculation:** The total distance $\text{cd}[c]$ traveled by courier c is the sum over all consecutive transitions:

$$\text{cd}[c] = \sum_{i,j,s} \text{If}(p_{c,i,s} \wedge p_{c,j,s+1}, D_{i,j}, 0),$$

where $D_{i,j}$ is the distance between nodes i and j.

6. **Maximum Distance Enforcement:** max_d is constrained to be at least $\text{cd}[c]$ for all couriers c, with equality for some courier c^*, representing the bottleneck distance to minimize.

7. **Symmetry Breaking:** To reduce redundant symmetric solutions, for couriers $c_1 < c_2$ with equal capacity, we enforce a lexicographical ordering on their package assignments:

$$\text{cw}[c_1] \leq_{\text{lex}} \text{cw}[c_2].$$

3.4 Validation

Search Strategies We apply two complementary strategies to optimize max_d:

- **Branch and Bound:** Starting with an upper bound on max_d, iteratively tighten it by solving the SAT problem under decreasing bounds.
- **Binary Search:** Maintain lower and upper bounds $[lb, ub]$ and iteratively check satisfiability at the midpoint to refine bounds until convergence.

Implementation Details The model is implemented in Python using the Z3 SMT solver. Arithmetic operations on binary vectors are performed via bit-level logical formulas (e.g., `binary_adder`, conditional sums). The solver runs within subprocesses to enable timeout enforcement and parallel experimentation.

Experimental Results Both search strategies demonstrate effective convergence to optimal solutions in benchmark tests. Symmetry-breaking constraints significantly reduce solving times in symmetric scenarios. The solver successfully finds optimal solutions for the first 10 problem instances but fails to solve the remaining 10 within resource limits.

Instance	noSB_BB	SB_BB	noSB_bin	SB_bin
1	**14**	**14**	**14**	**14**
2	**226**	**226**	**226**	**226**
3	**12**	**12**	**12**	**12**
4	**220**	**220**	**220**	**220**
5	**206**	**206**	**206**	**206**
6	**322**	**322**	**322**	**322**
7	-	-	-	-
8	-	-	-	-
9	-	-	-	-
10	-	-	-	-

Table 2: Comparison of model variants: SB = symmetry breaking; BB = branch-and-bound; bin = binary search. Bold = certified optimum.

4 SMT Model

The SMT (Satisfiability Modulo Theories) framework solves optimization problems using theory-specific solvers. For the couriers problem, we used linear integer and array theories to model the problem compactly and enable expressive constraints.

4.1 Decision Variables

We define:

- *b_path[c][j]*: Boolean matrix indicating whether courier c takes item j (Array theory + Boolean logic).
- *path[c][j]*: Integer matrix representing the item sequence visited by courier c (Array + Arithmetic theory).
- *load[c]*, *size[j]*, *path_length[c]*, *total_distance[c]*: Integer arrays for courier capacity, item size, path length, and total distance (Arithmetic theory).
- *D_func(i,j)*: Uninterpreted function representing the distance between items i and j (EUF theory).
- *max_dist*: Integer variable representing the maximum courier distance, to be minimized.

4.2 Objective Function

Minimize the longest courier route:

$$\min \max_c D_c, \quad D_c = \sum_{j=1}^{k_c} D_{\text{func}}(p_{c,j-1}, p_{c,j})$$

where k_c is the number of nodes visited by courier c, and $p_{c,j}$ is the j-th location in their route.

4.3 Constraints

Constraints enforce solution validity:

1. **Path range:** $0 \leq \text{path}[c][j] \leq n + 1$
2. **Path length:** $3 \leq \text{path\_length}[c] \leq \text{MAX\_ITEMS}$
3. **Start/End at depot:** $\text{path}[c][1] = \text{path}[c][\text{path\_length}[c]] = n + 1$
4. **Trailing zeros:** $\text{path}[c][i] = 0$ if $i > \text{path\_length}[c]$
5. **Unique assignment:** $\sum_c \text{If}(b\_path[c][j], 1, 0) = 1$
6. **Item limit:** $\sum_j \text{If}(b\_path[c][j], 1, 0) \leq \text{MAX\_ITEMS}$
7. **No revisits:** Distinct visited nodes per courier (excluding 0)
8. **Channeling:** $b\_path[c][i] \iff i \in \text{path}[c]$
9. **Load capacity:**

$$\sum_j \text{If}(b\_path[c][j], \text{size}[j], 0) \leq \text{load}[c]$$

$$\sum_{j=2}^{\text{MAX\_ITEMS}} \text{If}(j < \text{path\_length}[c] - 1, \text{size}[\text{path}[c][j]], 0) \leq \text{load}[c]$$

Symmetry Breaking For couriers with equal loads $c_1 < c_2$, enforce:

$$\text{lexleq}([b\_path[c_1][j]], [b\_path[c_2][j]])$$

4.4 Validation

Experimental Design We implemented the model in Python using Z3's optimizer. To enhance scalability, we introduced *branch-and-bound* and *binary search* strategies (see Subsection **??**).

Results Initial 10 instances were solved optimally with all methods and are omitted. The remaining results are summarized in Table 3.

Instance	noSB_BB	SB_BB	noSB_bin	SB_bin
11	925	1040	-	1066
12	490	504	506	659
13	1188	1234	1088	1648
14	-	-	-	-
15	-	-	-	-
16	**286**	**286**	**286**	295
17	-	-	-	-
18	1090	1195	1230	-
19	427	419	570	554
20	-	-	-	-
21	1006	-	1144	-

Table 3: Comparison of model variants: SB = symmetry breaking; BB = branch-and-bound; bin = binary search. Bold = certified optimum.

5 MIP Model

After initial experimentation with PuLP, we found that its limited solver access and lack of flexibility hindered performance on our benchmarks. This motivated a shift to PySCIPOpt, which offers deeper control over model construction, warm starts, and solver behaviour. With this transition, we were able to implement stronger constraints, inject high-quality incumbents, and significantly reduce runtime and optimality gaps. The following sections detail the core components of our final solver. We first introduce the decision variables that define the structure of the model. Then, we present the full set of constraints, including routing logic and flow conservation. Finally, we report the solver's performance across benchmark instances, highlighting optimality when reached.

5.1 Decision Variables

The model uses the following decision variables:

- **Routing variables** $x_{cij} \in \{0, 1\}$
  Binary variables equal to 1 if courier $c \in C$ travels directly from node i to node j, and 0 otherwise. These define the structure of each route over the filtered edge set.

- **Flow variables** $f_{cij} \in [0, \ell_c]$
  Continuous variables representing the remaining load carried by courier c when traversing arc (i, j). They implement a single-commodity flow model to ensure connectivity and eliminate subtours.

- **Courier activation variables** $y_c \in \{0, 1\}$
  Binary variables indicating whether courier c is used in the solution. These are used to conditionally activate load and distance constraints.

- **Load variables** $\texttt{load}_c \geq 0$
  Continuous variables representing the total demand assigned to courier c, computed from the flow sent out from the depot.

- **Distance variables** $\texttt{dist}_c \geq 0$
  Continuous variables equal to the total travel distance of courier c, computed as the sum of the distances of all active arcs.

- **Objective variable** $\texttt{max\_dist} \geq 0$
  A continuous variable representing the maximum route length among all couriers. The model minimises this value to balance courier workloads.

5.2 Problem Modeling

The constraints below specify the feasible region of our single–commodity flow formulation[6, 2]. Strengthening cuts and symmetry–breaking rules follow the guidelines of Toth and Vigo[9]. A high-quality warm start, obtained with a Clarke–Wright seed[3] polished by Google OR-Tools[7], is injected into SCIP[1]. This incumbent tightens dual bounds, accelerates the RINS[4] and Local Branching[5] heuristics, and lets the open-source solver match the primal performance of premium engines such as Gurobi.

- **Customer visit constraints:** Each customer must be visited exactly once by one courier:

$$\sum_{c \in C} \sum_{i \in \texttt{IN}(j)} x_{cij} = 1 \quad \forall j \in V \tag{4}$$

- **Flow conservation constraints:** For each courier and customer node, the incoming and outgoing flow values must reflect the demand:

$$\sum_{i \in \texttt{IN}(j)} f_{cij} - \sum_{k \in \texttt{OUT}(j)} f_{cjk} = s_j \cdot \sum_{i \in \texttt{IN}(j)} x_{cij} \quad \forall c, j \in V \tag{5}$$

- **Routing balance constraints:** Ensure that if a courier enters a customer node, it must also leave it:

$$\sum_{i \in \texttt{IN}(j)} x_{cij} = \sum_{k \in \texttt{OUT}(j)} x_{cjk} \quad \forall c, j \in V \tag{6}$$

- **Depot flow constraint:** The total load of a courier is equal to the outgoing flow from the depot:

$$\sum_{j \in \texttt{OUT}(d)} f_{cdj} = \texttt{load}_c \quad \forall c \in C \tag{7}$$

- **Courier activation constraints:** A courier may leave and return to the depot only if it is activated:

$$\sum_{j \in V} x_{cdj} = y_c \sum_{i \in V} x_{cid} = y_c \quad \forall c \in C \tag{8}$$

- **Capacity constraints:** Enforce load and distance limits only if the courier is used:

$$\texttt{load}_c \leq \ell_c \cdot y_c \; \texttt{dist}_c = \sum_{(i,j)} D_{ij} \cdot x_{cij} \; \texttt{dist}_c \quad \leq \texttt{max dist} \quad \forall c \in C \tag{9}$$

- **Symmetry-breaking constraints:** Couriers are ordered by descending load to reduce equivalent solutions, as we discussed in 1.1.5 This constraint follows the principles outlined in [9] to reduce redundant branches in the search tree.

- **Radial lower bound:** As we discussed in 1.1.5

- **Static subset capacity cuts:** For all customer subsets $S \subseteq V$ with $|S| \leq 3$ whose combined demand exceeds the maximum capacity, at least one arc must cross from S to its complement:

$$\sum_{c \in C} \sum_{i \in S,\ j \notin S \cup d} x_{cij} \geq 1 \quad \text{if } \sum_{j \in S} s_j > \max_c \ell_c \tag{10}$$

  These cuts help eliminate infeasible subtours early and are commonly used in CVRP formulations [9].

5.3 Validation

Experimental Results Table 4 summarises the MIP results. While our top-performing configuration outpaces every alternative, space constraints prevent us from listing the full experimental grid; instead, we show a representative subset to highlight the modelling levers—such as warm starts and symmetry-breaking—that proved decisive.

6 Conclusions

In conclusion, our project successfully utilized Constraint Programming (CP) with MiniZinc, Satisfiability Modulo Theories (SMT) using Z3, and Mixed-Integer Programming (MIP) with Gurobi to optimize routing for multiple couriers across 21 instances. Notably, our results were promising, demonstrating effective solutions across diverse scenarios. Surprisingly, we achieved improved outcomes without relying on symmetry breaking constraints, highlighting the robustness and efficiency of our approach. This project underscores the versatility and effectiveness of combining different optimization techniques to tackle complex logistics and routing challenges.

Inst	Gurobi+SB	GurobyPy w/out SB	PuLp (CBC)	SCIP+W.S.
1	**14**	**14**	**14**	**14**
2	**226**	**226**	**226**	**226**
3	**12**	**12**	**12**	**12**
4	**220**	**220**	**220**	**220**
5	**206**	**206**	**206**	**206**
6	**322**	**322**	**322**	**322**
7	**167**	**167**	**167**	**167**
8	**186**	**186**	**186**	**186**
9	**436**	**436**	**436**	**436**
10	**244**	**244**	**244**	**244**
11	N/A	N/A	N/A	**304**
12	541	N/A	N/A	**346**
13	420	474	800	398
14	N/A	N/A	N/A	**332**
15	N/A	N/A	N/A	**350**
16	**286**	**286**	N/A	**286**
17	N/A	N/A	N/A	387
18	N/A	N/A	N/A	302
19	N/A	N/A	N/A	**334**
20	N/A	N/A	N/A	348
21	N/A	N/A	N/A	**374**

Table 4: Comparison of results from different MIP models. The bold means that the optimal solution was reached.

References

[1] Tobias Achterberg. Scip: Solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009.

[2] Roberto Baldacci, Elias Hadjiconstantinou, and Andrea Mingozzi. An exact algorithm for the capacitated vehicle routing problem based on a two-commodity flow formulation. *Operations Research*, 52(5):723–738, 2004.

[3] Geoff Clarke and John W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964.

[4] E. Danna, E. Rothberg, and C. Le Pape. Exploring relaxation-induced neighbourhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2005.

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

[6] Bezalel Gavish and Stephen C. Graves. The travelling salesman problem and related problems. *Naval Research Logistics Quarterly*, 25(1):61–68, 1978.

[7] Laurent Perron and Vincent Furnon. OR-Tools: Google optimization tools. urlhttps://developers.google.com/optimization, 2019.

[8] Nicolas Briot Philippe Vismara. *A Circuit Constraint for Multiple Tours Problems*. Conference paper, 2018.

[9] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. SIAM, 2002.