GRID: Graph-Based Modelling Interface for Domain-Independent Dynamic Programming

Fabio Giordana
Department of Computer Science and Engineering, University of Bologna, Bologna, Italy

Zeynep Kiziltan
Department of Computer Science and Engineering, University of Bologna, Bologna, Italy

Ryo Kuroiwa
Principles of Informatics Division, National Institute of Informatics, Tokyo, Japan
The Graduate University for Advanced Studies, SOKENDAI, Kanagawa, Japan

Abstract

Domain-Independent Dynamic Programming (DIDP) is a general framework for solving combinatorial optimization problems using Dynamic Programming (DP), where search is separated from the problem specification. However, modelling directly in DIDP requires defining state variables, transitions, and dominance relations, which can be complex and error-prone. We introduce GRID, a modelling interface that enables high-level DP-oriented specifications. Users describe entities, relations, and resource attributes over a graph-based structure, from which GRID automatically compiles a DIDP model by deriving the components required by a DIDP solver. Common modelling elements are supported with built-in semantics, while additional constraints can be specified through user-defined variables and expressions. We use vehicle routing problems as a case study to present GRID and evaluate it on three variants: CVRP, PDPTW, and ECVRP. Results show that compilation overhead from GRID to DIDP is small and that generated models remain competitive with manually designed DIDP models while outperforming state-of-the-art CP and mixed-integer programming approaches.

2012 ACM Subject Classification Theory of computation → Dynamic programming; Computing methodologies → Modeling methodologies; Mathematics of computing → Combinatorial optimization

Keywords and phrases Modelling & Modelling Languages, Dynamic Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2026.26

Supplementary Material *Software*: `https://github.com/domain-independent-dp/grid`
archived at `swh:1:dir:f81cd8f5bd27a45fa3e524945e00a10aa0262f16`
Software: `https://github.com/Kurorororo/grid-models`
archived at `swh:1:dir:c0efd49c5a5d5b62e8726d891b60570d97d5d140`

Funding This work was supported by the NII-UniBo MoU Grant funded by NII.
Ryo Kuroiwa: JSPS KAKENHI grant number JP25K24378

1 Introduction

Dynamic Programming (DP) is a powerful paradigm for solving combinatorial optimization problems that can be formulated as sequential decision processes. A solution is constructed step by step: states summarize partial decisions, and transitions extend them while accumulating cost. This perspective provides strong pruning capabilities through dominance relations, often yielding highly effective exact algorithms. Domain-Independent Dynamic Programming (DIDP) [19] generalizes this idea by separating the search procedure from the problem specification. Rather than implementing a dedicated DP algorithm for each problem, the user models the problem by specifying state variables, transitions, and dominance rules, while a generic solver performs the exploration. This enables reusable solving technology across problem domains. However, modelling problems directly in DIDP remains difficult in practice. Users must manually design a state representation to summarize decisions and ensure the correctness of transitions and dominance relations, which limits accessibility.

In contrast, the CP community has developed high-level modelling features, such as structured decision variables [3, 4, 11, 20] and global constraints [34], which support intuitive problem descriptions and efficient exploitation of combinatorial substructures. Building on these features, solver-independent high-level modelling languages and libraries have emerged [8, 13, 22, 28] to ease the non-trivial modelling process. However, these tools target constraint-based formulations and offer limited support for state-based modelling in DP.

We bridge two modelling perspectives: lower-level DIDP formulations, which make state-space structure and dominance relations explicit but are labour-intensive to design, and high-level declarative models, which are concise but not expressed directly in terms of states and transitions. We introduce a GRaph-based modelling Interface for DIDP (GRID) that allows high-level, DP-oriented specifications of combinatorial optimization problems. Instead of manually defining DP components, users specify entities, relations, and resource attributes in a structured representation. GRID then automatically derives the state variables, transitions, feasibility conditions, objective accumulation rules, and dominance relations required for DIDP, and generates the corresponding DIDP model. In this initial version of GRID, we focus on Vehicle Routing Problems (VRPs) as a case study [1]. VRPs are a widely studied and flexible domain that naturally admits both declarative constraint-based formulations and DP-based incremental solution construction, and recent work has highlighted the interest in using DIDP for solving VRPs [19]. Because a route is a sequence that can be represented on a graph, GRID can potentially be extended to other closely related sequencing problems, such as job scheduling [2]. By contrast, it is not well suited to general graph optimization problems, such as vertex cover, facility location, or graph coloring, which do not involve constructing a sequence. We implement GRID as a Python library, inspired by recent solver-independent Python modelling libraries for CP [13, 22].

In what follows, after we review related work in Section 2, we introduce GRID in Section 3 using concrete VRPs as examples. We present native features useful for modelling common variants, as well as custom features for flexible, generic modelling: users can introduce variables, specify how decisions update them, and express constraints and the objective in terms of these variables. In Section 4, we describe GRID's underlying modelling formalism and how a GRID model is compiled into a DIDP model. In Section 5, we conduct an experimental study showing that GRID incurs small compilation overhead and that the resulting models perform comparably to manually designed DIDP models, while outperforming state-of-the-art CP and Mixed-Integer Programming (MIP) approaches. Finally, we conclude in Section 6.

2 Related Work

Modelling languages such as Essence [8] and MiniZinc [28] allow to specify high-level solver-independent CP models, thanks to rich decision variable types and global constraints to capture combinatorial substructures. Solver-independent modelling libraries have also emerged, including CPMPy [13] and PyCSP3 [22] in Python. These frameworks provide flexible modelling with access to multiple solvers and are widely used across combinatorial optimisation domains. Some CP solvers provide specialized modelling features tailored to particular problem classes. For example, IBM ILOG CP Optimizer [20] and Google OR-Tools CP-SAT [29] support interval variables and associated global constraints for scheduling. MaxiCP [31] provides sequence variables for VRPs [4, 3], allowing solvers to reason efficiently about tours or ordered tasks while handling precedence, cumulative resources, and other structural constraints. Despite these specialised features, these solvers remain generic and flexible: users can define additional variables, constraints, and objective functions as needed.

Unlike the CP approaches above, high-level modelling features for DP that capture problem-specific structure have not been developed. Gromicho et al. [12] view DP as a flexible algorithmic framework for solving VRPs, which is closely related to our setting. Our approach differs in three respects: we separate modelling and solving via DIDP, introduce a modelling formalism, and provide a practical high-level modelling interface. There have also been attempts to generate DP-oriented CP models or DP solving code from constraint-based specifications [24, 25]. In contrast, we develop a DP-oriented modelling interface by design.

For specific problems, solvers with high-level modelling interfaces have also been proposed. For VRPs, dedicated solvers have been developed [17], such as VRPSolver [30, 7], VRPy [27], Google OR-Tools' Vehicle Routing Solver (VRSolver) [5], PyVRP [36], and RoutingBlocks [16], all of which provides a Python interface. For jobshop scheduling variants, PyJobShop [21] offers high-level modelling in Python and solves models by compiling them into CP. The majority of these solvers target specific variants modelled as combinations of predefined features. Exceptionally, RoutingBlocks is more flexible at the cost of being less declarative: a problem is defined by procedures that extend a route and manage resources, implemented as a Python class and its methods. While GRID provides predefined native features to ease modelling of common variants, as shown in Section 3.2, it also offers custom features to model complex variants in a declarative way. Users can introduce variables, specify how decisions update them, and express constraints and the objective in terms of these variables. These capabilities allow us to model problems beyond the scope of existing VRP-specific solvers, as shown in Section 3.3.

3 GRID: A Graph-based Modelling Interface for DIDP

GRID is a modeling interface for high-level, DP-oriented specifications of combinatorial optimization problems on a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. In this initial version, we focus on VRPs as a natural and representative use case for this graph-based abstraction. As described in Section 4, a GRID model can be compiled automatically into a DIDP model with state variables, transitions, feasibility conditions, cost accumulation rules, and dominance relations.

3.1 VRP Modelling Elements

A VRP consists of a set of locations and vehicles operating on a graph: nodes represent locations and edges represent vehicle travel between them, yielding feasible routes that start and end at designated nodes while visiting mandatory or optional nodes. The interface for VRP is built around four main elements, implemented as Python classes: `RoutingModel`, `Node`, `Edge`, and `VehicleType`. `RoutingModel` acts as a container, defining the graph structure and managing all entities. `Node` objects correspond to graph nodes $\mathcal{N}$ and supports attributes such as demand and service time. `Edge` objects represent directed edges $\mathcal{E}$ connecting pairs of nodes, with attributes such as distance and travel time. `VehicleType` objects define a fleet of identical vehicles with shared characteristics, including capacity and fixed usage cost. Nodes, edges, and vehicle types are created through the `add_node`, `add_edge`, and `add_vehicle_type` methods of `RoutingModel` and can be stored in variables for future reference. The global objective is defined via the `set_objective` method of `RoutingModel`.

We refer to the predefined set of modelling constructs whose semantics are built into the interface as *native features*. These features enable users to express many common VRP variants by setting object attributes (e.g., a node's demand or a vehicle's capacity) or by selecting a built-in objective via `set_objective`, without manually encoding their

effects on states, transitions, or costs. For example, to model the classical Capacitated VRP (CVRP), the interface supports node demands, vehicle capacities, travel costs, and the objective of minimising total travel distance. The native features and their semantics are inspired by common VRP variants in the literature [1]. If a VRP variant cannot be fully specified using the native features, *custom features* can be introduced via user-defined variables and expressions over them. These expressions define how the variables are updated, aggregated, or constrained when interacting with graph elements such as nodes and edges. For instance, as we will show in Section 3.3, an Electric Capacitated VRP (ECVRP) with battery constraints and charging stations requires user-defined rules to track battery levels and charging behaviour, which cannot be captured by native features.

User-defined variables are declared in `VehicleType`. Each variable has a single definition, and the interface assumes the same behaviour for every vehicle of that type. This design choice reflects the role of vehicles as the dynamic elements that construct solutions during search: `VehicleType` provides a natural scope for quantities that evolve along routes while remaining shared across identical vehicles. This level of abstraction is sufficient for our targeted routing models, while leaving open the possibility of supporting variables at a more general scope in future versions. The `VehicleType` class provides the `add_integer_var` and `add_float_var` methods to declare integer and floating-point variables, and the `add_nodes_set_var` method to declare set variables whose elements are graph nodes represented by their integer IDs. When defining integer or floating-point variables, users may specify preferences over their values; in this case, it must be guaranteed that, among two partial routes ending at the same node, the route with the preferred value leads to a better solution.

Expressions are attached to nodes or edges, and specify how user-defined variables are updated, constrained, or aggregated. For example, an expression can update a variable when a node or edge is included in a route, restrict the selection of certain nodes or edges, and compute accumulated costs. This is implemented via the `Expression` class and its subclasses for variables and constants, which overload Python operators for intuitive definition. Constraints can be applied globally or locally to nodes and edges via the `add_constraint` method, and nodes or edges can update variables when included in a route via the `add_transition` method. The global objective is defined via the `set_objective` method by either aggregating a built-in metric or user-defined variables, that can be minimized (by default) or maximized.

3.2 Modelling PDPTW using Native Features

We now present an example GRID model for the Pickup and Delivery Problem with Time Windows (PDPTW), which can be specified using only natively supported features.

Problem Definition [6]. Given a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $\mathcal{N} = \{0\} \cup \mathcal{I}$ is the set of nodes, where 0 is the depot housing a fleet of m vehicles with a capacity q, and $\mathcal{I} = \{1, \dots, 2n\}$ is the set of customers. We partition $\mathcal{I}$ into pickup locations $\mathcal{I}^+ \subset \mathcal{I}$ and delivery locations $\mathcal{I}^- \subset \mathcal{I}$ with $\mathcal{I} = \mathcal{I}^+ \cup \mathcal{I}^-$ and $|\mathcal{I}^+| = |\mathcal{I}^-| = n$. Each pickup location $i \in \mathcal{I}^+$ is coupled with a unique delivery location $j \in \mathcal{I}^-$ (and vice versa), representing the origin and destination of a specific task. Each node $i \in \mathcal{N}$ has an associated service time s_i (with $s_0 = 0$), time window $[a_i, b_i]$ and demand μ_i (with $\mu_0 = 0$). For a pickup location $i \in \mathcal{I}^+$, $\mu_i > 0$, and for a delivery location $i \in \mathcal{I}^-$, $\mu_i < 0$. If a vehicle reaches a node $i \in \mathcal{N}$ before its earliest arrival time a_i, it has to wait at the location. Each edge $(i, j) \in \mathcal{E}$ has a non-negative travel time c_{ij}, equivalent to the travel distance. The objective is to find a set of routes for the vehicles starting and ending at the depot, visiting all the customers within their time windows, finishing all tasks without exceeding the vehicle capacity, while minimizing the total travel distance.

Code Listing 1 GRID model for PDPTW.

```
1  # m: number of vehicles, q: vehicle capacity
2  # tasks: list of pickup and delivery tasks
3  # p[t], d[t]: pickup and delivery node indices of task t
4  # a[i], b[i], s[i]: time window and service time of node i
5  # mu[i]: demand of node i
6  # edges: set of edges (i,j) with time (distance) c[i][j]
7
8  model = RoutingModel()
9  model.add_vehicle_type(id=0, count=m, capacity=q)
10 model.add_node(id=0, tw_start=a[0], tw_end=b[0], depot=True)
11
12 for t in tasks:
13     model.add_node(
14         id=p[t],
15         pickup={f"{t}": mu[p[t]]},
16         tw_start=a[p[t]], tw_end=b[p[t]], service_t=s[p[t]],
17     )
18     model.add_node(
19         id=d[t],
20         delivery={f"{t}": mu[d[t]]},
21         tw_start=a[d[t]], tw_end=b[d[t]], service_t=s[d[t]],
22     )
23 for (i, j) in edges:
24     model.add_edges(
25         node_from=model.get_node(i), node_to=model.get_node(j),
26         distance=c[i][j],
27     )
28
29 model.set_objective(metric="distance")
```

GRID Model. In Listing 1, lines 1–6 show the input parameters; line 8 creates the `RoutingModel` container. Line 9 adds a vehicle type representing the homogeneous fleet with its ID, count, and capacity. Line 10 initializes the depot node with its ID, time window, and depot flag. Lines 12–22 then add the nodes associated with each task: for every task, two nodes are created, one with a pickup and one with a delivery attribute. These attributes link each node to the corresponding task identifier and demand. The `pickup` and `delivery` arguments are dictionaries mapping task IDs to pickup or delivery amounts. Before a delivery node is visited, the same vehicle must have already visited every pickup node whose `pickup` dictionary shares a task ID with the delivery node's `delivery` dictionary. This semantics allows tasks with multiple pickup or delivery points, and nodes that act as both a pickup for one task and a delivery for another. Time windows and service times are also specified during node creation. Lines 23–27 add edges for all the node pairs in the `edges` set and assign distances. Finally, line 29 defines the global objective by setting the objective metric to the distance of a travelled edge, which is by default aggregated by summing all the distances and minimized, yielding an objective that minimizes the total distance travelled by all vehicles. GRID also supports other built-in objective metrics and aggregation functions, as well as objective maximization, by initializing the related method parameters accordingly, e.g., `metric="num_vehicles"` for the number of used vehicles, `aggregation="max"`, and `maximization="True"`.

Code Listing 2 GRID model for ECVRP (I).

```
1  # m: number of vehicles, q: vehicle capacity
2  # b: maximum and initial battery level
3  # customers, stations: set of customer and station node indices
4  # mu[i]: demand of customer i
5
6  v = model.add_vehicle_type(id=0, count=m)
7  load = v.add_integer_var(initial_value=q)
8  battery = v.add_float_var(initial_value=b, preference="high")
9  depot = model.add_node(id=0, depot=True)
10 depot.add_constraint(load < q)
11
12 for i in customers:
13     customer = model.add_node(id=i)
14     customer.add_constraint(load - mu[i] >= 0)
15     customer.add_transition(var=load, expression=load - mu[i])
16 for s in stations:
17     station = model.add_node(id=s, optional=True, max_visits=None)
18     station.add_transition(var=battery, expression=b)
```

3.3 Modelling ECVRP using Custom Features

This second example illustrates how to model the Electric Capacitated Vehicle Routing Problem (ECVRP), a variant that cannot be specified using only natively supported features.

Problem Definition [26]. Given a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $\mathcal{N} = \{0\} \cup \mathcal{I} \cup \mathcal{F}$ is the set of nodes, where 0 is the depot housing a fleet of vehicles with a capacity q, $\mathcal{I} \subset \mathcal{N}$ is the set of customers, and $\mathcal{F} \subset \mathcal{N}$ is the set of charging stations. Each customer $i \in \mathcal{I}$ has a delivery demand $\mu_i > 0$. Each charging station $f \in \mathcal{F}$ can be visited by a vehicle to fully recharge its battery to the maximum battery level b. Multiple visits to the same station are allowed. Each edge $(i, j) \in \mathcal{E}$ has a non-negative travel distance d_{ij}. When an edge (i, j) is travelled, the battery charge decreases by hd_{ij}, where h is the energy consumption rate proportional to the load l carried by the vehicle, defined as $h = r + \frac{l}{q}$ with a given constant r. Vehicles leave the depot fully charged and loaded. The objective is to find a set of routes for the vehicles starting and ending at the depot, visiting all customers exactly once, without exceeding vehicle capacity and without having a negative battery value along a travelled edge, while minimizing the total distance travelled. Even though the number of vehicles m does not appear in the problem definition, its upper bound is $|\mathcal{I}|$.

GRID Model. Listing 2 initializes the fleet and the nodes, together with the user-defined variables. The fleet is created via `add_vehicle_type` (line 6), and two user-defined variables are declared and attached to the vehicle type, representing route-specific quantities: an integer for vehicle load (`load` in line 7) and a float for battery level (`battery` in line 8) with a preference for higher values. The depot node constrains the load to be less than its initial load at return (line 10), to ensure that any vehicle returning to the depot has served at least one customer, thereby preventing routes that visit only charging stations. Unused vehicles that never leave the depot are still permitted, since the constraint applies only at return. Customer nodes enforce that `load` never reaches a negative value (line 14) and update `load` by subtracting the customer's demand via the `add_transition` method (line 15), where the

Code Listing 3 GRID model for ECVRP (II).

```
19  # edges: set of edges (i,j) with distance d[i][j]
20  # r: fixed battery consumption
21  # min_from[i]: minimum distance from node i to any next node
22
23  for (i, j) in edges:
24      edge = model.add_edge(
25          node_from=model.get_node(i), node_to=model.get_node(j),
26          distance=d[i][j],
27      )
28      next_battery = battery - d[i][j] * (r + load / q)
29      if j not in customers:
30          edge.add_constraint(next_battery >= 0)
31      else:
32          edge.add_transition(var=battery, expression=next_battery)
33          edge.add_constraint(
34              next_battery >= min_from[j] * (r + (load - mu[j]) / q)
35          )
36
37  model.set_objective(metric="distance")
38
39  unv_stations = v.add_nodes_set_var(initial_value=stations)
40
41  for i in customers:
42      customer = model.get_node(i)
43      customer.add_transition(var=unv_stations, expression=stations)
44  for s in stations:
45      station = model.get_node(s)
46      station.add_constraint(unv_stations.contains(s))
47      station.add_transition(
48          var=unv_stations, expression=unv_stations.remove(s)
49      )
```

`expression` attribute specifies the new value assigned to the variable `load` when reaching the node. By default, `add_node` creates a mandatory node that must be visited exactly once. Stations are optional nodes and may be visited multiple times, achieved by setting the `optional` and `max_visits` attributes accordingly (line 17). When a station is visited, the battery level reaches full without any effect on its load through the `add_transition` method (line 18), where `battery` is updated to `b`.

Listing 3 defines the edges and the objective. Each edge is added with its distance (lines 24–27). Depending on the destination node, edges enforce battery constraints differently. When travelling to a station or the depot, the remaining battery level must be non-negative, ensuring that vehicles do not run out of battery along the edge (lines 29–30). When travelling to a customer, `battery` is updated via `add_transition` (line 32). Unlike the `load` update, the `battery` update is edge-specific because energy consumption depends on the edge distance. In this case, the battery constraint requires sufficient charge for subsequent customers, accounting for the reduced load after the customer visit (lines 33–35).

Visits to charging stations can be refined. Since the number of visits to each station is bounded by $2|\mathcal{I}|$ [26], we can set max_visits to this value in line 17. To further prune the search space, we avoid cycles consisting only of charging stations, as consecutive station visits

Code Listing 4 An example user-defined objective in GRID.

```
model.set_objective(
    variables=[var_on_type1, var_on_type2], aggregation="max"
)
```

do not change load or battery levels. This is enforced with a set variable initially containing all stations: a station can be visited only if it is in the set and is removed upon visit. The set is reset when a customer is visited or a new route starts, allowing stations to be revisited in later routes. The variable, updates, and constraint are shown in lines 39–49.

The ECVRP model minimizes the total travel distance (line 37), but GRID also supports user-defined objective metrics via a user-defined variable or a list of such variables. When a list is given, each variable must be associated with a distinct `VehicleType` variable, ensuring a one-to-one mapping between optimization variables and vehicle types in the fleet. Listing 4 shows an example with multiple variables in the objective; here, the `aggregation` attribute takes the maximum of their final values at the end of each route to produce the objective. The battery constraint in the ECVRP is not supported in some VRP-specific solvers such as VRPy [27] and PyVRP [36]. While VRPSolver [30, 7] and Google OR-Tools' VRSolver [5] allow to define resource changes along a route (called dimensions in OR-Tools), they assume that traversing an edge changes a resource by a constant and that each node constrains a resource between constant lower and upper bounds. However, battery consumption depends on the current load. The global constraints for sequence variables in MaxiCP impose similar restrictions. Thus, we were unable to model the ECVRP with them. In contrast, GRID's DP-oriented modelling allows variables to be updated and constrained based on the current values of other variables. Overall, GRID provides a powerful yet lightweight, high-level modelling approach that closely follows natural-language problem descriptions, in contrast to more cumbersome, lower-level DIDP formulations, as shown in the next section.

4 Compilation of GRID to DIDP

Compilation converts a GRID model into an executable DIDP model written in DyPDL [19]. Before describing this process, we introduce the notation used in DyPDL.

4.1 Compilation Target: DyPDL

In DyPDL, a model is defined by *states*, *transitions*, a *target state*, and *base cases*. A state S is a complete assignment to state variables $x_1, \dots, x_n$, where the domain of x_i depends on its type: *numeric* ($\mathbb{Q}$), *element* ($\mathbb{Z}_0^+$), or *set* ($2^{\mathbb{Z}_0^+}$). We denote the value of x_i in S by $S[x_i]$. The target state S^I is a particular state defined explicitly. A transition updates the state variables; for a state S, let $\mathcal{T}(S)$ denote the set of *applicable transitions*. Applying $\tau \in \mathcal{T}(S)$ to S yields the successor state $S[\![\tau]\!]$. A base case specifies conditions on state variables, and no transition is applicable once a base case is satisfied.

Given a state S, an *S-solution* is a sequence of transitions $\langle \tau_1, \dots, \tau_m \rangle$ inducing a sequence of states $\langle S^0, \dots, S^m \rangle$ with $S^0 = S$, $S^{i+1} = S^i[\![\tau_{i+1}]\!]$, and $\tau_{i+1} \in \mathcal{T}(S^i)$ for $i = 0, \dots, m-1$, and S^m satisfies a base case. A solution for a DIDP model is an S^I-solution starting from the target state S^I. The objective value of an S-solution is $w_{\tau_1}(S^0) \circ w_{\tau_2}(S^1) \circ \cdots \circ w_{\tau_m}(S^{m-1}) \circ w_{\text{base}}(S^m)$, where w_τ is the weight function for a transition τ, w_{base} is the weight function for a base case, and $\circ : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ is a binary operator (e.g., $+$) satisfying certain conditions

including associativity (see [19] for details). Throughout, we assume minimization of the objective value; maximization can be handled analogously. A DIDP model is described by the following recursive equation defining $V(S)$, the optimal objective value of an S-solution:

$$\text{compute } V(S^I) \tag{1a}$$

$$V(S) = \begin{cases} w_{\text{base}}(S) & \text{if } S \text{ satisfies a base case} \\ \min_{\tau \in \mathcal{T}(S)} w_\tau(S) \circ V(S[\![\tau]\!]) & \text{otherwise.} \end{cases} \tag{1b}$$

In DyPDL, state dominance can be defined by annotating numeric or element variables as *resource variables*. If a variable x_i is a resource variable, a preference (higher or lower) is specified. Resource variables must be defined so that, for any two states S and S', we have $V(S) \leq V(S')$ whenever $S[x_i] \geq S'[x_i]$ for each resource variable x_i where higher is preferred, $S[x_i] \leq S'[x_i]$ for each x_i where lower is preferred, and $S[x_i] = S'[x_i]$ for each non-resource variable x_i. In addition, a *dual bound function* h can be defined such that $V(S) \geq h(S)$.

4.2 GRID Modelling Formalism

We define a generic VRP targeted by our interface. We are given a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and a set of fleets Θ, where each fleet $\theta \in \Theta$ is a set of identical vehicles of the same type. Each vehicle $v \in \theta$ starts at $\alpha^\theta \in \mathcal{N}$ and ends at $\omega^\theta \in \mathcal{N}$ and is assigned a route $\pi^v = (\pi^v_0, \dots, \pi^v_{m^v})$ with $\pi^v_0 = \alpha^\theta$ and $\pi^v_{m^v} = \omega^\theta$. We say that a route π^v or a vehicle v visits node π^v_i for $i \in \{0, \dots, m^v\}$. Nodes are classified as mandatory (visited by exactly one vehicle) or optional (may be visited multiple times by multiple vehicles or skipped). For example, in ECVRP, we have only one fleet ($\Theta = \{\theta\}$) starting and ending at the same location ($\alpha^\theta = \omega^\theta = 0$). The customers ($\mathcal{I}$) are mandatory, and the stations ($\mathcal{F}$) are optional.

More route constraints can be expressed via variables associated with each fleet. Let $x^\theta = (x^\theta_1, \dots, x^\theta_{n^\theta})$ be the vector of variables for a fleet $\theta \in \Theta$. As in DIDP, we consider numeric, element, and set variables with their respective domains, and denote the set of possible assignments by $\mathcal{D}^\theta$. The user specifies the initial assignment $\hat{x}^\theta \in \mathcal{D}^\theta$, an update function $e^\theta_{ij} : \mathcal{D}^\theta \to \mathcal{D}^\theta$ for each edge $(i,j) \in \mathcal{E}$, and constraints $\mathcal{C}^\theta_{ij} \subseteq \mathcal{D}^\theta$ for each $(i,j) \in \mathcal{E}$ that define the allowed assignments. A route $\pi^v = \langle \pi^v_0, \dots, \pi^v_{m^v} \rangle$ for vehicle $v \in \theta$ must satisfy $\pi^v_i[x^\theta] \in \mathcal{C}^\theta_{\pi^k_i, \pi^k_{i+1}}$ for $i = 0, \dots, m^v - 1$, where $\pi^v_{i+1}[x^\theta] = e^\theta_{\pi^v_i, \pi^v_{i+1}}(\pi^v_i[x^\theta])$ and $\pi^v_0[x^\theta] = \hat{x}^\theta$. In ECVRP, we have a numeric variable l for the load, a numeric variable y for the battery, and a set variable R for the set of unvisited stations since the last visit to a customer, with the initial values $l = q$, $y = b$, and $R = \mathcal{F}$. For edge (i,j) with $j \in \mathcal{I}$, the update function e^θ_{ij} maps l to $l - \mu_j$, y to $y - d_{ij}\left(r + \frac{l}{q}\right)$, and R to $\mathcal{F}$. For $j \in \mathcal{F}$, it maps l to l, y to b, and R to $R \setminus \{j\}$. The constraints $\mathcal{C}^\theta_{ij}$ are $l - \mu_j \geq 0$ and $y - d_{ij}\left(r + \frac{l}{q}\right) \geq \mathsf{min\_from}_j \times \left(r + \frac{l-\mu_j}{q}\right)$ for $j \in \mathcal{I}$, $y - d_{ij}\left(r + \frac{l}{q}\right) \geq 0$ and $j \in R$ for $j \in \mathcal{F}$, and $y - d_{i0}\left(r + \frac{l}{q}\right) \geq 0$ and $l < q$ for $j = 0$, where $\mathsf{min\_from}_j$ is the minimum travel time from j to any next node.

To define the objective, one variable o^θ is selected for each fleet θ. The objective is to minimize or maximize an aggregate of the final values $\pi^k_{m^k}[o^\theta]$, using either $\sum_{\theta \in \Theta, v \in \theta} \pi^v_{m^v}[o^\theta]$, $\max_{\theta \in \Theta, v \in \theta} \pi^v_{m^v}[o^\theta]$, or $\min_{\theta \in \Theta, v \in \theta} \pi^v_{m^v}[o^\theta]$. In ECVRP, we define a numeric variable o representing the travel cost, update it to $o + d_{ij}$ with edge (i,j) and minimize the sum.

As in DIDP, we consider dominance based on resource variables. We compare two partial routes of the same vehicle that do not necessarily end at the vehicle's end location. Consider $\pi^v = \langle \pi^v_0, \dots, \pi^v_j \rangle$ and $\pi'^v = \langle \pi'^v_0, \dots, \pi'^v_{j'} \rangle$ that visit the same set of mandatory nodes and end at the same node, i.e., $\pi^v_j = \pi'^v_{j'}$. We say that π^v dominates π'^v if, for any feasible

completion (a complete route) of π'^v with a sequence of nodes, the completion of π^v with the same sequence is feasible and yields a better objective value (smaller for minimization and larger for maximization). Let $\pi^v_j[x^\theta_i]$ (resp., $\pi'^v_{j'}[x^\theta_i]$) denote the value of x^θ_i in $\pi^v_j[x^\theta]$ (resp., $\pi'^v_{j'}[x^\theta]$) with $v \in \theta$. Resource variables must be defined so that π^v dominates π'^v whenever $\pi^v_j[x^\theta_i] \geq \pi'^v_{j'}[x^\theta_i]$ for each resource variable x^v_i where higher is preferred, $\pi^v_j[x^\theta_i] \leq \pi'^v_{j'}[x^\theta_i]$ for each x^v_i where lower is preferred, and $\pi^v_j[x^\theta_i] = \pi'^v_{j'}[x^\theta_i]$ for each non-resource variable x^θ_i. In ECVRP, the battery variable y is a resource variable where higher is preferred.

4.3 Mapping GRID to DyPDL

Our DP formulation follows the single-route representation [12], constructing vehicle routes sequentially. At each transition, we either visit a node with the current vehicle or switch to a new vehicle. We use three state variables: a set variable U for unvisited mandatory locations, an element variable i for the current location, and an element variable v for the index of the current vehicle. Vehicles of the same type are given consecutive indices, so v and $v+1$ belong to the same fleet unless v is the last vehicle of the fleet. In addition, we have a vector of state variables x^θ for each fleet $\theta \in \Theta$, corresponding to the GRID variables. In the target state, U contains all mandatory nodes, $i = \alpha^0$ (the starting depot of the first fleet), $v = 0$ (the first vehicle), and $x^\theta = \hat{x}^\theta$ (the initial values). We have the following transitions:

- Visit j, applicable only if $(i, j) \in \mathcal{E}$ and $x^\theta \in \mathcal{C}^\theta_{ij}$ for $v \in \theta$ and updating i to j and x^θ to $e^\theta_{ij}(x^\theta)$. If j is mandatory, it is applicable only if $j \in U$, and U is updated to $U \setminus \{j\}$. The objective weight is zero.
- Change vehicle, applicable only if $(i, \omega^\theta) \in \mathcal{E}$, $U \neq \emptyset$, and $x^\theta \in \mathcal{C}^\theta_{i\omega^\theta}$ for $v \in \theta$ and updating i to $\alpha^{\theta'}$ for $v+1 \in \theta'$, v to $v+1$, and x^θ to $\hat{x}^\theta$. The objective weight is the value of o^θ in $e^\theta_{i\omega^\theta}(x^\theta)$.
- Finish, applicable only if $(i, \omega^\theta) \in \mathcal{E}$, $U = \emptyset$, and $x^\theta \in \mathcal{C}^v_{i\omega^\theta}$ for $v \in \theta$ and updating i to ω^θ and x^θ to $e^\theta_{i\omega^\theta}(x^\theta)$. The objective weight is the updated value of o^θ.

The base case is defined over the core state variables: $U = \emptyset$ and $i = \omega^\theta$ for $v \in \theta$, with zero weight. Depending on the objective aggregation, the binary operator $\circ$ to combine transition weights is $+$, min, or max. The state variable v is a resource variable with lower values preferred. If a user-defined variable in x^θ (for some fleet θ) is declared as a resource variable, it is compiled into DyPDL as a resource variable with the corresponding preference.

Although the objective value is typically treated as a state variable and updated only when changing vehicle type or finishing, we provide a specialised implementation for minimising the total travel distance, which is natively supported. We assign transition weights d_{ij} for visits to j and $d_{i\omega^\theta}$ (for $v \in \theta$) for transitions that change the vehicle or finish, and omit the state variable o^θ. We also introduce the following dual bound function:

$$V(U, i, v, (x^\theta)_{\theta\in\Theta}) \geq \max \left\{ \sum_{j \in (U \cup \{\omega^{\theta'}\}) \setminus \{i\}} \min_{k \in \mathcal{N}:(k,j)\in\mathcal{E}} d_{kj}, \sum_{j \in (U \cup \{i\}) \setminus \{\omega^{\theta'}\}} \min_{k \in \mathcal{N}:(j,k)\in\mathcal{E}} d_{jk} \right\} \quad (2)$$

where $v \in \theta'$. The first term sums the minimum travel distance $\min_{k\in\mathcal{N}:(k,j)\in\mathcal{E}} d_{kj}$ to each node j for all unvisited nodes and the end depot. The second term considers the minimum travel distance from each node. Similarly, for specific native features, GRID may introduce enhancements such as a dual bound function, resource variables, and state constraints.

As the above structure reflects a single fleet and a single depot, we further provide enhancements for more complex settings. For multiple fleets, we add a skip transition that moves directly to the next fleet, avoiding activation costs for intermediate unused vehicles.

For multi-depot instances, we introduce a virtual depot that serves as a unified starting point from which specific initial and final transitions originate, effectively selecting the correct and real physical depots for each vehicle's route.

Compiled DIDP Model for PDPTW. Let U be the set of unvisited customers, i the current location, and v the index of the current vehicle; these are the core state variables introduced earlier. Let l be the current load, t the current time, and R the set of delivery nodes that must be visited by the current vehicle; these additional variables are introduced by translating the selected native features. Let $V(U, i, v, l, t, R)$ denote the minimum cost to visit all customers in U starting from i with load l, time t, and delivery set R, using $m - v$ vehicles. To support more general pickup-and-delivery variants (e.g., nodes that are both pickup and delivery nodes with multiple commodities), we introduce two additional data structures for each node $j \in \mathcal{N} \setminus \{0\}$: pre_j, the nodes that must be visited before j by the same vehicle, and suc_j, the nodes that must be visited after j before returning to the depot.

$$\text{compute } V(N \setminus \{0\}, 0, 0, 0, 0, \emptyset) \tag{3a}$$

$$V(U,i,v,l,t,R) = \min \begin{cases} \min\limits_{j \in \mathcal{I}^{+'}(U,i,l,t)} c_{ij} + V(U \setminus \{j\}, j, v, l + \mu_j, t'_{ij}, (R \setminus \{j\}) \cup \text{suc}_j) \\ \min\limits_{j \in \mathcal{I}^{-'}(U,i,l,t)} c_{ij} + V(U \setminus \{j\}, j, v, l + \mu_j, t'_{ij}, R \setminus \{j\}) \\ c_{i0} + V(U, 0, v+1, 0, 0, \emptyset) & \text{if change}(U,i,v,t,R) \\ c_{i0} + V(U, 0, v, 0, t + s_i + c_{i0}, \emptyset) & \text{if finished}(U,i,v,t) \\ 0 & \text{if } U = \emptyset \wedge i = 0 \end{cases} \tag{3b}$$

$$V(U,i,v,l,t,R) \leq V(U,i,v',l',t',R) \quad \text{if } v \leq v' \wedge l \leq l' \wedge t \leq t' \tag{3c}$$

$$V(U,i,v,l,t,R) = \infty \quad \text{if } t + c^*_{i0} > b_0 \tag{3d}$$

$$V(U,i,v,l,t,R) = \infty \quad \text{if } \exists j \in R : t + c^*_{ij} > b_j \tag{3e}$$

$$V(U,i,v,l,t,R) \geq \max \left\{ \sum_{j \in (U \cup \{0\}) \setminus \{i\}} \min_{k \in \mathcal{N} : (k,j) \in \mathcal{E}} c_{kj}, \sum_{j \in (U \cup \{i\}) \setminus \{0\}} \min_{k \in \mathcal{N} : (j,k) \in \mathcal{E}} c_{jk} \right\} \tag{3f}$$

where we use the time update when moving from i to j $t'_{ij} = \max\{t + s_i + c_{ij}, a_j\}$, the set of pickup nodes that can be visited next $\mathcal{I}^{+'}(U,i,l,t)$, the set of delivery nodes that can be visited next $\mathcal{I}^{-'}(U,i,l,t)$, the condition to change a vehicle $\text{change}(U,i,v,t,R)$, and the condition to return to the depot $\text{finished}(U,i,v,R)$, defined as follows:

$$\mathcal{I}^{+'}(U,i,l,t) = \{j \in U \cap \mathcal{I}^+ \mid (i,j) \in \mathcal{E} \wedge l + \mu_j \leq q \wedge t + s_i + c_{ij} \leq b_j\}$$

$$\mathcal{I}^{-'}(U,i,l,t) = \{j \in U \cap \mathcal{I}^- \mid (i,j) \in \mathcal{E} \wedge l - \mu_j \geq 0 \wedge (U \cap \text{pre}_j) = \emptyset \wedge t + s_i + c_{ij} \leq b_j\}$$

$$\text{change}(U,i,v,t,R) = t + s_i + c_{i0} \leq b_0 \wedge (i,0) \in \mathcal{E} \wedge U \neq \emptyset \wedge v < m - 1 \wedge R = \emptyset$$

$$\text{finished}(U,i,v,t) = t + s_i + c_{i0} \leq b_0 \wedge (i,0) \in \mathcal{E} \wedge U = \emptyset$$

The first two lines of Equation (3b) describe the *Visit* j core transition: they share the same preconditions and effects on the core state variables U, i, and v, and differ only in how pickup and delivery nodes are handled based on l and R; time management is identical. The next three lines describe the *Change Vehicle* transition, the *Finish* transition, and the base case. Inequality (3c) defines the resource variables, automatically introduced for the load and time features specified via the native features. Similarly, Equation (3d) is introduced for the time attribute and (3e) for the pickup-and-delivery attribute. These state constraints use the shortest travel time c^*_{ij} between nodes i and j: at any time, it must remain possible

to return to the depot and to visit any mandatory delivery node in R without exceeding their latest arrival time. By default, GRID computes c^*_{ij} with the Floyd–Warshall algorithm, though users may provide the matrix directly. Inequality (3f) instantiates Inequality (2).

Compiled DIDP Model for ECVRP. Let U be the set of unvisited customers, i the current location, and v the index of the current vehicle; as for PDPTW, these are the core state variables. Let l be the current load, y the current battery level, and R the set of unvisited stations since the last visit to the depot or a customer, which are defined by custom features. Let $V(U, i, v, l, y, R)$ denote the minimum cost to visit all customers in U starting from i with load l, battery y, and station set R using $m - v$ vehicles.

$$\text{compute } V(\mathcal{I}, 0, 0, q, b, \mathcal{F}) \tag{4a}$$

$$V(U,i,v,l,y,R) = \min \begin{cases} \min\limits_{j \in \mathcal{I}'(U,i,l,y)} d_{ij} + V\left(U \setminus \{j\}, j, v, l - \mu_j, y - d_{ij}\left(r + \frac{l}{q}\right), \mathcal{F}\right) & \\ \min\limits_{j \in \mathcal{F}'(i,y,R)} d_{ij} + V(U, j, v, l, b, R \setminus \{j\}) & \\ d_{i0} + V(U, 0, v+1, q, b, \mathcal{F}) & \text{if change}(U,i,v,l,y) \\ d_{i0} + V(U, 0, v, q, b, \mathcal{F}) & \text{if finish}(U,i,v,l,y) \\ 0 & \text{if } U = \emptyset \wedge i = 0 \end{cases} \tag{4b}$$

$$V(U,i,v,l,y,R) \leq V(U,i,v',l,y',R) \quad \text{if } v \leq v' \wedge y \geq y' \tag{4c}$$

$$V(U,i,v,l,y,R) \geq \max \left\{ \sum_{j \in (U \cup \{0\}) \setminus \{i\}} \min_{z \in \mathcal{N} : (z,j) \in \mathcal{E}} d_{zj}, \sum_{j \in (U \cup \{i\}) \setminus \{0\}} \min_{z \in \mathcal{N} : (j,z) \in \mathcal{E}} d_{jz} \right\} \tag{4d}$$

where we use the set of customers that can be visited next $\mathcal{I}'(U, i, l, y)$, the set of stations that can be visited next $\mathcal{F}'(i, y, R)$, the condition to change the vehicle change(U, i, v, l, y), and the condition to return to the depot finished(U, i, l, y), defined as follows:

$$\mathcal{I}'(U,i,l,y) = \left\{ j \in \mathcal{I} \cap U \mid \begin{array}{l} l - \mu_j \geq 0 \wedge y - d_{ij}\left(r + \frac{l}{q}\right) \geq \mathsf{min\_from}_j \times \left(r + \frac{l-\mu_j}{q}\right) \\ \wedge (i,j) \in \mathcal{E} \end{array} \right\}$$

$$\mathcal{F}'(i,y,R) = \left\{ j \in \mathcal{F} \cap R \mid y - d_{ij}\left(r + \frac{l}{q}\right) \geq 0 \wedge (i,j) \in \mathcal{E} \right\}$$

$$\text{change}(U,i,v,l,y) = y - d_{i0}\left(r + \frac{l}{q}\right) \geq 0 \wedge l < q \wedge (i,0) \in \mathcal{E} \wedge U \neq \emptyset \wedge v < m - 1$$

$$\text{finished}(U,i,l,y) = y - d_{i0}\left(r + \frac{l}{q}\right) \geq 0 \wedge l < q \wedge (i,0) \in \mathcal{E} \wedge U = \emptyset.$$

Note that $\mathsf{min\_from}_j$ is the minimum travel time from j to any next node given in a model. The first two lines of Equation (4b) describe the *Visit* j core transition: visiting a station does not change U as it can be visited multiple times. Updates to i and v remain the same. The first equation handles customer visits, decreasing both l and y and resetting R to its initial value. For station visits, j is removed from R to avoid station-only cycles and y is reset to its initial value, while the load is unchanged. The next three lines describe the *Change Vehicle* transition, the *Finish* transition, and the base case. While v is required as a resource variable for pruning, enforcing $v \leq m$ is redundant because it is the number of customers and each vehicle is already forced to have a non-empty path; the current implementation enforces it anyway. Inequality (4c) defines the resource variables. Since higher load enables more customer visits but also increases battery discharge, there is no clear preference, so it is not declared as a resource variable. Inequality (4d) instantiates Inequality (2).

4.4 Expression Aggregation

In DIDPPy, the existing Python interface for DIDP, each transition updates state variables via expressions built from a predefined set of operations. In the ECVRP model, the battery update depends on the traversed edge $(k, j) \in \mathcal{E}$. One could model this by defining a transition corresponding to visit location j from location i, applicable only if $i = k$ where i is the current location. However, this yields a quadratic number of transitions in the number of nodes. This can be mitigated using the constant-table feature in DIDPPy: users register a multi-dimensional array of constants and index it with state variables in expressions. In the ECVRP example, one could use a single transition to visit node j and update y as $y - d_{ij}\left(r + \frac{l}{q}\right)$, where i is the state variable, j is a constant, and d is a two-dimensional constant table. However, this is not straightforward in GRID, since users specify the update on b per edge rather than defining a constant table.

We develop a method that parses edge-specific update functions and constraints, identifies common structure across edges, and compiles them into a single expression indexed by a constant table. The interface parses the user constraints $\mathcal{C}^{\theta}_{ij}$ and update functions e^{θ}_{ij} into Abstract Syntax Trees (ASTs). Two expressions share the same pattern if they apply the same operations and differ only in constant values. We group expressions by edge (i, j); if multiple edges share the same set of patterns across their variables, we aggregate them into a single DIDP transition using a table of constants. A more fundamental solution is to extend DIDPPy to support tables of expressions rather than constants, so that edge updates and constraints can be expressed via a two-dimensional table indexed by the state variable i and a constant j. We leave this extension as future work and use DIDPPy as is in this paper.

5 Experimental Validation

In our experimental validation, we pursue three goals:

1. Evaluate the translation overhead introduced by GRID.
2. Assess how translation affects the resulting DIDP models and their solutions, compared to manually designed models.
3. Compare the solving performance of our approach with other model-based solvers applied successfully to VRPs in previous work: (i) Google OR-Tools CP-SAT [29], the winner of MiniZinc Challenge over years,[1] (ii) MaxiCP [31], a Java-based CP solver with sequence variables for routing, (iii) Gurobi [14], a commercial MIP solver. We also include model-based solvers specific to some VRPs: (iv) PyVRP [36], (v) OR-Tools VRSolver [5].

The aim of the third goal is not a comprehensive comparison with the state-of-the-art VRP solvers, but to show that GRID achieves reasonable performance with flexible modelling; improving solving performance is left for future work. We use three VRPs: PDPTW and ECVRP, for which DIDP approaches have not been investigated before, and CVRP, a simplification of ECVRP without battery constraints, for which prior work formulated a DIDP model [19]. We use the following metrics:

- **Model building time**: the wall-clock time to construct and initialize a model using the modelling API of the solvers before solving.
- **Primal gap**: the relative gap between the best-known objective value γ^* and the best objective value γ achieved by a solver, defined as $|\gamma - \gamma^*| / \max\{\gamma, \gamma^*\}$. We take the best-known objective value γ^* from the literature, as explained later.

[1] https://www.minizinc.org/challenge/

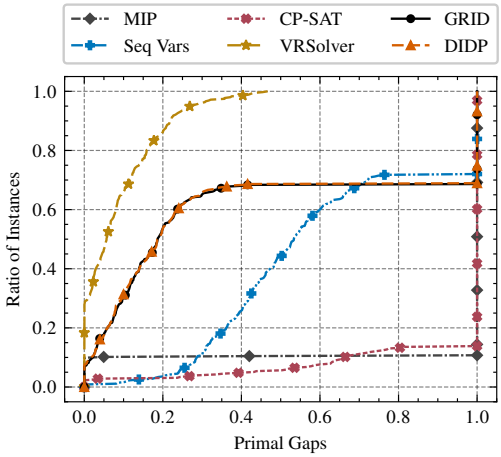
As software, we use DIDPPy 0.10.0, CP-SAT 9.15.6755, Gurobi Python Interface (gurobipy) 13.0.1, and PyVRP 0.13.3, all implemented with Python 3.11.2, and Java 21.0.10 for MaxiCP. For search, we use LNBS for GRID and DIDP models, which was reported to be effective in VRPs [18]. For CP-SAT, we enable Large Neighbourhood Search (LNS) [32] by setting `use_lns=True` because LNS is typically good in solving VRPs. For the OR-Tools VRSolver, we let the solver determine the strategy to find the first feasible solution and use guided local search [35] for improvement, as recommended in the documentation.[2] For MaxiCP, while the original paper designed problem-specific LNS on top of the sequence-variable model, we do not use it because we focus on fully model-based approaches, and the LNS code does not seem to have been published. For Gurobi, no well-known parameter tuning exists for VRP, so we use default parameters. Each run uses a single thread, an 8 GB memory limit for model building and solving, and a 5-minute time limit for solving, on a node with 8 cores provided by an AMD Athlon/Opetron CPU. The code used in our experiments is provided as supplementary material.

5.1 Models and Problem Instances

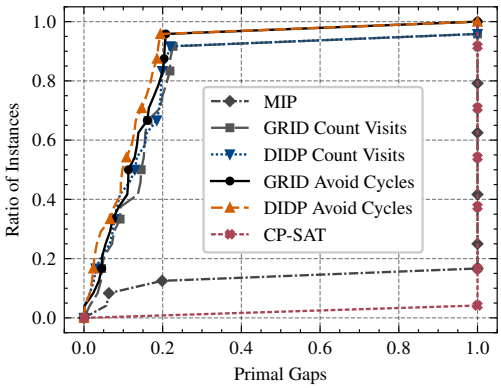
For PDPTW, we manually design a DIDP model simpler than the one generated by GRID by omitting pre_j and suc_j, which allow to model a general pickup-and-delivery version, as discussed in Section 4.3 (presented in Appendix A). The MIP model is a two-index vehicle-routing model proposed by [9]. In its original form, the objective jointly minimizes the number of used vehicles and the total travel distance. We modified it with a constraint that bounds the number of vehicles by the maximum specified in each instance. For CP, we adapted two Dial-a-Ride Problem formulations from [3]: the successor model based on the circuit global constraint [15] and the model using sequence variables. In the former, we introduced a symmetry-breaking constraint over the fleet of identical vehicles, which slightly improved performance. For the latter model, we adapt the code provided in the MaxiCP repository, which specifies search parameters.[3] We are not able to use PyVRP for PDPTW due to its modelling limitations: it does not support customer-to-customer pickup-delivery pairs. The instances are from [23] with 100 to more than 1000 nodes, and the best-known objective values are taken from a website.[4] We use the best objective value obtained by our solvers when it is better; the solutions on the website minimize the number of vehicles first, and then the total travel distance, whereas we directly minimize the total travel distance.

Our manually designed DIDP model for ECVRP matches the one generated by GRID, except that the manual formulation omits the redundant maximum vehicle precondition. The MIP model is taken from [26]. We formulate a CP model based on the MIP model, where battery-related parameters are scaled by the vehicle load capacity to avoid floating-point values. Both MIP and CP formulations require duplicating charging stations to allow multiple visits, which significantly increases the number of nodes in the model. As discussed earlier, ECVRP cannot be modelled in PyVRP and VRSolver, nor with sequence-variable global constraints in MaxiCP. In addition to the GRID and DIDP models using a set variable to avoid cycles of stations (Avoid Cycles), we also test models enforcing the total number of visits to a station to not exceed the upper bound $2|\mathcal{I}|$ (Count Visits), removing the need for the additional set variable, but enforcing a weaker condition (as presented in Section 3.3). The instances with their best-known objective values are from [26] and preprocessed as described in Appendix B. Instance size ranges from a few tens to over a thousand nodes.

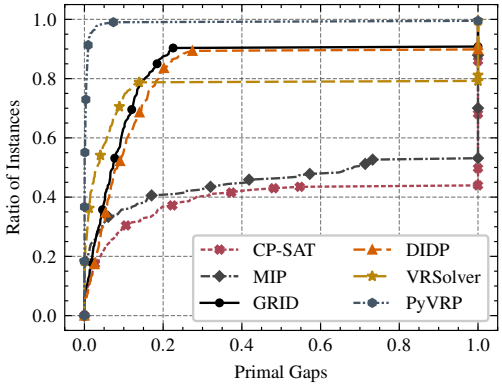
[2] https://developers.google.com/optimization/routing/routing_options
[3] https://github.com/aia-uclouvain/maxicp/blob/main/src/main/java/org/maxicp/cp/examples/modeling/DARPSeqVar.java
[4] http://combopt.org/tables/LiLim/



(a) PDPTW.



(b) ECVRP.



(c) CVRP.

Figure 1 Experimental results on PDPTW, ECVRP and CVRP.

Table 1 ECVRP model building times (s).

Size	Avoid Cycles		Count Visits		MIP	CP-SAT
	DIDP	GRID	DIDP	GRID		
[1, 200]	0.01	0.57	0.01	0.67	149.16	77.72
[201, 400]	0.07	5.69	0.07	8.03	882.87	352.65
[401, 600]	0.28	34.12	0.25	28.93	–	–
[601, 800]	0.62	45.86	0.49	45.55	–	–
[801, 1000]	0.90	68.98	0.91	70.30	–	–
[1001, ∞)	1.00	95.27	1.01	95.32	–	–

The GRID model for CVRP is a simplification of the ECVRP model, omitting battery constraints and implementing demands and capacity using native features. The DIDP model taken from [19] defines a transition to visit each customer after changing a vehicle, unlike the GRID model, which uses a single transition to change a vehicle. This finer-grained structure for GRID offers more flexibility, e.g. for multi-depot formulations (presented in Appendix A). The MIP model is from [10]. The CP model is a simplified version of the successor model of the PDPTW. We do not use MaxiCP because it does not directly support vehicle load tracking on sequence variables and assumes the triangle inequality of the travel distance, which is violated by some instances. Following [19], we take A, B, E, F, M, P, X, and DIMACS instances with the best-known objective value from CVRPLIB [33]. Instance size ranges from tens to over a thousand nodes.

5.2 Experimental Results

Table 1 shows the model building time for ECVRP. In larger instances, MIP and CP-SAT run out of memory before solving with a large number of variables and constraints, introduced by duplicated charging stations. GRID requires below two minutes even for the larger instances.

This overhead is mostly due to parsing and aggregating the update functions defined for edges, as described in Section 4.4. For CVRP and PDPTW, the overhead is much smaller, with an average of around 6 seconds with more than 1000 nodes, as reported in Appendix C.

In Figure 1, we evaluate the distribution of instances over the primal gap: a point (x, y) in each plot indicates that in $100 \times y$ percent of instances, the primal gap is at most x. While this metric evaluates solution quality, the strength of the bounds proven by the solvers is analysed separately using the optimality gap in the Appendix. Figure 1a shows the results for PDPTW: the VRSolver clearly outperforms all other approaches. GRID and the native DIDP model follow with very similar performance, their curves remaining close over the entire ratio of instances. The sequence-variable model achieves competitive results only on a limited portion of instances, running out of memory for higher size instances. The CP-SAT and MIP models exhibit significantly weaker performance.

In Figure 1b (ECVRP), the DIDP approaches are better than MIP and CP-SAT, with the best models being the ones enforcing the cycle avoidance, rather than just limiting the total number of visits to each station to the upper bound. This result suggests that user-defined expressions can be used to improve performance by injecting domain knowledge.

In Figure 1c (CVRP), the best performance is clearly achieved by PyVRP. VRSolver shows a strong initial performance, but its relative performance decreases as the ratio of considered instances increases, ultimately performing slightly below GRID and DIDP. Among the latter, the GRID model performs slightly better than the original DIDP formulation from previous work. The MIP and CP-SAT approaches are comparable to the others on a small fraction of instances, but overall remain below the leading methods.

Across the three problem classes, no single approach consistently dominates all scenarios. The VRP-specific solvers excel on classical variants, but they do not always support richer or non-standard variants. Conversely, general-purpose MIP and CP offer broad modelling flexibility but struggle with scalability. GRID matches the performance of the manually designed DIDP models – the minor formulation differences between them have little impact on solving performance – outperforming MIP and CP, while offering a higher-level, flexible, and declarative modelling interface. As we focus on a new modelling interface in this paper, addressing the performance gap relative to VRP-specific solvers is left for future work.

6 Conclusions

This paper introduced GRID, a graph-based modelling interface designed to bridge the gap between high-level declarative specifications and the execution requirements of DIDP. The interface provides Python abstractions to describe entities, resources, and constraints, which are automatically compiled into DyPDL models. Vehicle routing problems are used as a representative case study to evaluate the approach. The experimental results show that the interface preserves performance comparable to native DIDP models and to traditional modelling approaches, while substantially reducing modelling effort. The possibility to extend models through user-defined attributes and expressions further indicates that the framework is not restricted to built-in modelling primitives while being declarative.

In future work, we will focus on extending GRID for other problem classes, improving the compilation layer, designing generic solving algorithms (e.g., LNS) that exploit high-level modelling features, and investigating the compilation of GRID models to non-DP back-ends (e.g., CP, MIP, VRP solvers).

References

1 C. Archetti, L.C. Coelho, M.G. Speranza, and P. Vansteenwegen. Beyond fifty years of vehicle routing: Insights into the history and the future. *Eur. J. Oper. Res.*, 330(2):355–372, 2026. doi:10.1016/j.ejor.2025.06.014.

2 J. Beck, Patrick Prosser, and E. Selensky. Vehicle routing and job shop scheduling: What's the difference? In *ICAPS*, pages 267–276, 2003.

3 Augustin Delecluse, Pierre Schaus, and Pascal Van Hentenryck. Sequence variables: A constraint programming computational domain for routing and sequencing. *CoRR*, abs/2510.09373, 2025. doi:10.48550/arXiv.2510.09373.

4 Augustin Delecluse, Pierre Schaus, and Pascal Van Hentenryck. Sequence variables for routing problems. In *CP*, volume 235, pages 19:1–19:17, 2022. doi:10.4230/LIPIcs.CP.2022.19.

5 Frédéric Didier, Laurent Perron, Sarah Mohajeri, Steven Agostino Gay, Thibaut Cuvelier, and Vincent Furnon. OR-Tools' Vehicle Routing Solver: a generic constraint-programming solver with heuristic search for routing problems. In *ROADEF*, 2023.

6 Yvan Dumas, Jacques Desrosiers, and François Soumis. The pickup and delivery problem with time windows. *Eur. J. Oper. Res.*, 54(1):7–22, 1991. doi:10.1016/0377-2217(91)90319-Q.

7 Najib Errami, Eduardo Queiroga, Ruslan Sadykov, and Eduardo Uchoa. VRPSolverEasy: A Python library for the exact solution of a rich vehicle routing problem. *INFORMS J. Comput.*, 36(4):956–965, 2024. doi:10.1287/ijoc.2023.0103.

8 Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. doi:10.1007/s10601-008-9047-y.

9 Maria Gabriela S. Furtado, Pedro Munari, and Reinaldo Morabito. Pickup and delivery problem with time windows: A new compact two-index formulation. *Oper. Res. Lett.*, 45(4):334–341, 2017. doi:10.1016/j.orl.2017.04.013.

10 S.L. Gadegaard and J. Lysgaard. A symmetry-free polynomial formulation of the capacitated vehicle routing problem. *Discrete Appl. Math.*, 296:179–192, 2021. doi:10.1016/j.dam.2020.02.012.

11 Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997. doi:10.1007/BF00137870.

12 J. Gromicho, J.J. van Hoorn, A.L. Kok, and J.M.J. Schutten. Restricted dynamic programming: A flexible framework for solving realistic VRPs. *Comput. Oper. Res.*, 39(5):902–909, 2012. doi:10.1016/j.cor.2011.07.002.

13 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *ModRef*, 2019.

14 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL: https://www.gurobi.com.

15 Philip Kilby and Paul Shaw. Vehicle routing. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 801–836. Elsevier, 2006. doi:10.1016/S1574-6526(06)80027-1.

16 Patrick S. Klein and Maximilian Schiffer. RoutingBlocks: An open-source Python package for vehicle routing problems with intermediate stops. *INFORMS J. Comput.*, 36(4):966–973, 2024. doi:10.1287/IJOC.2023.0104.

17 Nicholas D. Kullman, Jorge E. Mendoza, and Ted K. Ralphs. Introduction to the special section on software tools for vehicle routing. *INFORMS J. Comput.*, 36(4):941–942, 2024. doi:10.1287/ijoc.2024.ed.v36.n4.2.

18 Ryo Kuroiwa and J. Christopher Beck. Large neighborhood beam search for domain-independent dynamic programming. In *CP*, volume 280, pages 23:1–23:22, 2023. doi:10.4230/LIPIcs.CP.2023.23.

19 Ryo Kuroiwa and J. Christopher Beck. Domain-independent dynamic programming. *Artif. Intell.*, 354:104506, 2026. doi:10.1016/j.artint.2026.104506.

20 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling. *Constraints*, 23(2):210–250, 2018. doi:10.1007/s10601-018-9281-x.

21 Leon Lan and Joost Berkhout. PyJobShop: Solving scheduling problems with constraint programming in Python. *CoRR*, abs/2502.13483, 2025. doi:10.48550/arXiv.2502.13483.

22 Christophe Lecoutre and Nicolas Szczepanski. PyCSP3: Modeling combinatorial constrained problems in Python. *CoRR*, abs/2009.00326, 2020. arXiv:2009.00326.

23 H. Li and A. Lim. A metaheuristic for the pickup and delivery problem with time windows. In *ICTAI*, pages 160–167, 2001. doi:10.1109/ICTAI.2001.974461.

24 Shu Lin, Na Meng, and Wenxin Li. Optimizing constraint solving via dynamic programming. In *IJCAI*, pages 1146–1154, 2019. doi:10.24963/ijcai.2019/160.

25 Shu Lin, Na Meng, and Wenxin Li. Generating efficient solvers from constraint models. In *ESEC/FSE*, pages 956–967, 2021. doi:10.1145/3468264.3468566.

26 Michalis Mavrovouniotis, Charalambos Menelaou, Stelios Timotheou, Georgios Ellinas, Christoforos Panayiotou, and Marios M. Polycarpou. A benchmark test suite for the electric capacitated vehicle routing problem. In *CEC*, pages 1–8, 2020. doi:10.1109/CEC48606.2020.9185753.

27 Romain Montagné, David Torres Sanchez, and Halvard Olsen Storbugt. VRPy: A Python package for solving a range of vehicle routing problems with a column generation approach. *J. Open Source Softw.*, 5(55):2408, 2020. doi:10.21105/JOSS.02408.

28 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, volume 4741, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

29 Laurent Perron, Frédéric Didier, and Steven Gay. The CP-SAT-LP solver. In *CP*, volume 280, pages 3:1–3:2, 2023. doi:10.4230/LIPIcs.CP.2023.3.

30 Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck. A generic exact solver for vehicle routing and related problems. *Math. Prog.*, 183(1):483–523, 2020. doi:10.1007/s10107-020-01523-z.

31 Pierre Schaus, Guillaume Derval, Augustin Delecluse, Laurent Michel, and Pascal Van Hentenryck. MaxiCP: A constraint programming solver for scheduling and vehicle routing, 2024. URL: https://github.com/aia-uclouvain/maxicp.

32 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP*, pages 417–431, 1998. doi:10.1007/3-540-49481-2_30.

33 Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the Capacitated Vehicle Routing Problem. *Eur. J. Oper. Res.*, 257(3):845–858, 2017. doi:10.1016/j.ejor.2016.08.012.

34 Willem-Jan van Hoeve and Irit Katriel. Global constraints. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi:10.1016/S1574-6526(06)80010-6.

35 Christos Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *Eur. J. Oper. Res.*, 113(2):469–499, 1999. doi:10.1016/S0377-2217(98)00099-X.

36 Niels A. Wouda, Leon Lan, and Wouter Kool. PyVRP: A high-performance VRP solver package. *INFORMS J. Comput.*, 36(4):943–955, 2024. doi:10.1287/ijoc.2023.0055.

A GRID-generated vs Manually-designed DIDP Models

In this section, we present the DIDP models omitted from the main text and compare the GRID-generated models with those manually-designed for the PDPTW and CVRP. We skip the ECVRP, as the generated model is identical to the manual one, except a redundant precondition, as mentioned in Section 5.

A.1 PDPTW

Manually-designed Model. Let U be the set of unvisited customers, i the current location, v the index of the current vehicle, l the current load, t the current time, and R the set of delivery nodes we must visit with the current vehicle. Let $V(U, i, v, l, t, R)$ be the minimum cost to visit all customers in U from i with load l, time t and delivery nodes R using $m - v$ vehicles. For each pickup node $j \in \mathcal{I}^+$, we indicate with $d_j \in \mathcal{I}^-$ the correspondent delivery node.

$$\text{compute } V(N \setminus \{0\}, 0, 0, 0, 0, \emptyset) \tag{5a}$$

$$V(U,i,v,l,t,R) = \min \begin{cases} \min\limits_{j \in \mathcal{I}^{+'}(U,i,l,t)} c_{ij} + V(U \setminus \{j\}, j, v, l + \mu_j, t'_{ij}, (R \cup \{d_j\}) \\ \min\limits_{j \in \mathcal{I}^{-'}(U,i,t,R)} c_{ij} + V(U \setminus \{j\}, j, v, l + \mu_j, t'_{ij}, R \setminus \{j\}) \\ c_{i0} + V(U, 0, v+1, 0, 0, \emptyset) & \text{if change}(U,i,v,t,R) \\ c_{i0} + V(U, 0, v, 0, t + s_i + c_{i0}, \emptyset) & \text{if finished}(U,i,v,t) \\ 0 & \text{if } U = \emptyset \wedge i = 0 \end{cases} \tag{5b}$$

$$V(U,i,v,l,t,R) \leq V(U,i,v',l',t',R) \quad \text{if } v \leq v' \wedge l \leq l' \wedge t \leq t' \tag{5c}$$

$$V(U,i,v,l,t,R) = \infty \quad \text{if } t + c^*_{ij} > b_0 \tag{5d}$$

$$V(U,i,v,l,t,R) = \infty \quad \text{if } \exists j \in R : t + c^*_{ij} > b_j \tag{5e}$$

$$V(U,i,v,l,t,R) \geq \max \left\{ \sum_{j \in (U \cup \{0\}) \setminus \{i\}} \min_{k \in \mathcal{N}:(k,j) \in \mathcal{E}} c_{kj}, \sum_{j \in (U \cup \{i\}) \setminus \{0\}} \min_{k \in \mathcal{N}:(j,k) \in \mathcal{E}} c_{jk} \right\} \tag{5f}$$

where we use the time update when moving from i to j $t'_{ij} = \max\{t + s_i + c_{ij}, a_j\}$, the set of pickup nodes that can be visited next $\mathcal{I}^{+'}(U,i,l,t)$, the set of delivery nodes that can be visited next $\mathcal{I}^{-'}(U,i,t,R)$, the condition to change a vehicle change(U,i,v,t,R), and the condition to return to the depot finished(U,i,v,R), defined as follows:

$$\mathcal{I}^{+'}(U,i,l,t) = \{j \in U \cap \mathcal{I}^+ \mid (i,j) \in \mathcal{E} \wedge l + \mu_j \leq q \wedge t + s_i + c_{ij} \leq b_j\}$$
$$\mathcal{I}^{-'}(U,i,t,R) = \{j \in U \cap \mathcal{I}^- \mid (i,j) \in \mathcal{E} \wedge j \in R \wedge t + s_i + c_{ij} \leq b_j\}$$
$$\text{change}(U,i,v,t,R) = t + s_i + c_{i0} < b_0 \wedge (i,0) \in \mathcal{E} \wedge U \neq \emptyset \wedge v < m - 1 \wedge R = \emptyset$$
$$\text{finished}(U,i,v,t) = t + s_i + c_{i0} < b_0 \wedge (i,0) \in \mathcal{E} \wedge U = \emptyset$$

Comparison. This model differs from the GRID generated version (presented in Section 4.3) in its management of pickup and delivery visits and the updates to the state variable R. By leveraging domain-specific knowledge of paired nodes, it eliminates the need for the pre_j and suc_j sets.

The first two lines of Equation (5b) describe these core transitions:

- **Pickup:** When visiting $j \in \mathcal{I}^+$, the specific delivery node d_j is directly added to R, rather than fetching a set of successors from a general data structure.
- **Delivery:** A node $j \in \mathcal{I}^-$ is reachable only if $j \in R$. This simplifies the precedence check: instead of verifying that all predecessors in pre_j have been removed from U, it simply ensures the corresponding pickup has already occurred for the current vehicle.

Except for these differences in handling the pickup-delivery logic, the two models are otherwise identical.

A.2 CVRP

Problem Definition. The CVRP is defined as a simplification of the ECVRP. A complete directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is given where $\mathcal{N} = \{0\} \cup \mathcal{I}$ is the set of nodes and $\mathcal{E} = \{(i,j) | i,j \in \mathcal{N}, i \neq j\}$ is the set of edges between the nodes. Node 0 is the depot housing a fleet of m vehicles with individual capacity q, and $\mathcal{I} \subset \mathcal{N}$ represents the set of customers. Each edge has an associated non-negative value d_{ij} representing the distance between nodes i and j. Each customer $i \in \mathcal{I}$ has a positive delivery demand μ_i. The objective is to find a set of routes for the vehicles that visit all the customers without exceeding the vehicle capacity, while minimizing the total travel cost.

GRID-generated Model. Let U be the set of unvisited customers, i the current location, and v the index of the current vehicle. These are the core state variables of the model. Let l be the current load. This is the additional variable defined by the translation of native features. Let $V(U,i,v,l)$ be the minimum cost to visit all customers in U from i with load l using $m - v$ vehicles. Since no preprocessing has been applied and the graph remains fully connected, the condition $(i,j) \in \mathcal{E}$ is implicitly satisfied for each $i,j \in \mathcal{N}, i \neq j$, and thus does not require explicit verification.

$$\text{compute } V(N \setminus \{0\}, 0, 0, 0) \tag{6a}$$

$$V(U,i,v,l) = \min \begin{cases} \min\limits_{j \in U : l + \mu_j \leq q} d_{ij} + V(U \setminus \{j\}, j, v, l + \mu_j) & \\ d_{i0} + V(U, 0, v+1, 0) & \text{if } U \neq \emptyset \wedge i \neq 0 \wedge v < m-1 \\ d_{i0} + V(\emptyset, 0, v, l) & \text{if } U = \emptyset \wedge i \neq 0 \\ 0 & \text{if } U = \emptyset \wedge i = 0 \end{cases} \tag{6b}$$

$$V(U,i,v,l) \leq V(U,i,v',l') \quad \text{if } v \leq v' \wedge l \leq l' \tag{6c}$$

$$V(U,i,v,l) = \infty \quad \text{if } (m-v) \times q - l < \sum_{j \in U} \mu_j \tag{6d}$$

$$V(U,i,v,l) \geq \max \left\{ \sum_{j \in (U \cup \{0\}) \setminus \{i\}} \min_{k \in \mathcal{N} \setminus \{j\}} c_{kj}, \sum_{j \in (U \cup \{i\}) \setminus \{0\}} \min_{k \in \mathcal{N} \setminus \{j\}} c_{jk} \right\} \tag{6e}$$

The first line in Equation (6b) defines the *Visit j* transition, where a customer is reachable by the current vehicle only if the resulting cumulative load does not exceed the maximum capacity q. Upon execution, the state variables representing unvisited customers U and the current location i are updated, while the current load l is incremented by μ_j. The subsequent cases delineate the *Change Vehicle* and *Finish* transitions, alongside the base case.

Inequality (6c) specifies the resource variables. Equation (6d) establishes a state constraint based on capacity feasibility: at any given state, the total residual capacity of both the current and remaining vehicles must be sufficient to satisfy the aggregate demand of all unvisited customers, introduced by the native feature to define the demand. Finally, Inequality (6e) introduces the usual dual bound over the total distance travelled.

Comparison. This GRID-generated model differs from the DIDP model from previous work [19] in its management of vehicle changes and the subsequent transitions structure:

- **GRID model:** The *Change Vehicle* transition returns to the depot as an independent step, after which a separate *Visit j* transition is applied. This decomposition keeps each transition minimal.

- **DIDP model:** A single transition simultaneously returns to the depot and visit the next customer j, directly encoding the assumption that a new vehicle departs from the depot to serve j.

The finer-grained structure in GRID offers greater modelling flexibility. For instance, it naturally accommodates multi-depot formulations, where the depot to which a vehicle returns and the one from which the new vehicle departs do not need to coincide. We conjecture that this finer granularity also contributes to the slight performance improvement observed in Section 5.2: decoupling the return to the depot from the next customer visit enables a more accurate assessment of state quality through resource variables and dual bounds, since the weight of a vehicle change is no longer entangled with the cost of a subsequent trip.

B ECVRP Preprocessing

In this section, we detail the preprocessing procedures applied to the instances of the ECVRP before solving. The aim is to prune unfeasible edges based on the load and the battery constraints.

Deleting edges by capacity

A couple of customers cannot be visited by the same vehicle if their combined demand is higher than the vehicle's capacity q.

$$\forall i, j \in \mathcal{I}, i \neq j : \mu_i + \mu_j > q \implies (i, j) \notin \mathcal{E}.$$

Deleting edges by battery

Considering a specific edge (i, j), it can be pruned if the battery becomes negative along the edge or if the remaining battery at j is insufficient to continue, even in the best possible scenario. Since the battery discharge along an edge increases with higher loads, we can define the minimum discharge by considering the minimum loads at each step.

For $i \neq 0$, the minimum loads before and after reaching the node are:

$$\mathsf{min\_load\_before_i} = \mu_i + \mu_j, \quad \mathsf{min\_load\_after_i} = \mu_j \quad \mathsf{min\_load\_after_j} = 0$$

For $i = 0$, the vehicle departs with full load: $\mathsf{min\_load\_after_i} = q$ and $\mathsf{min\_load\_after_j} = q - \mu_j$.

We define $\mathsf{min\_to}_z = \min_{k \in \mathcal{N}:\,(k,z) \in \mathcal{E}} d_{kz}$ and $\mathsf{min\_from}_z = \min_{k \in \mathcal{N}:\,(z,k) \in \mathcal{E}} d_{zk}$. The edge (i, j) is removed if any of the following conditions holds:

- **Battery exhausted before reaching i (only for $i \neq 0$):**

$$\mathsf{max\_battery\_after_i} = b - \mathsf{min\_to}_i \times \left(r + \frac{\mathsf{min\_load\_before_i}}{q} \right) < 0$$

If $\mathsf{max\_battery\_after_i} \geq 0$ and $i \in \mathcal{F}$, the battery resets: $\mathsf{max\_battery\_after_i} = b$. For $i = 0$, we set directly $\mathsf{max\_battery\_after_i} = b$.

- **Battery exhausted along edge (i, j):**

$$\mathsf{max\_battery\_after_j} = \mathsf{max\_battery\_after_i} - d_{ij} \times \left(r + \frac{\mathsf{min\_load\_after_i}}{q} \right) < 0$$

If $\mathsf{max\_battery\_after_j} \geq 0$ and $j \in \mathcal{F}$, the battery resets: $\mathsf{max\_battery\_after_j} = b$.

Table 2 PDPTW model building times (s).

Size	DIDP	GRID	MIP	CP-SAT	SeqVar	VRSolver
$[1, 200]$	0.01	0.04	0.47	0.19	0.90	0.02
$[201, 400]$	0.03	0.15	1.61	0.76	6.21	0.05
$[401, 600]$	0.12	0.80	6.28	3.30	98.08	0.22
$[601, 800]$	0.28	2.11	13.40	7.84	493.01	0.51
$[801, 1000]$	0.54	3.67	22.54	14.57	1579.80	0.96
$[1001, \infty)$	0.82	5.62	32.87	23.50	–	1.53

- **Insufficient battery to leave j (only for $j \neq 0$):**

$$\mathsf{battery\_lb_j} = \mathsf{min\_from}_j \times \left(r + \frac{\mathsf{min\_load\_after_j}}{q} \right)$$

$$\mathsf{max\_battery\_after_j} < \mathsf{battery\_lb_j} \implies (i, j) \notin \mathcal{E}$$

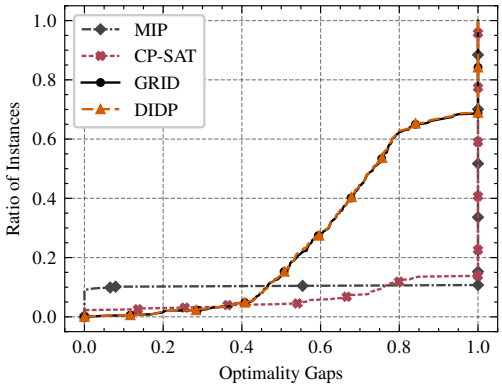
C Additional Experimental Results

In this section, we provide supplementary results that further support the performance analysis discussed in the main text. Specifically, Table 2 and Table 3 show the model-building times for both PDPTW and CVRP, comparing the computational efficiency of the different optimization paradigms across various instance sizes. In both problems, DIDP and GRID are among the fastest approaches, with MIP and CP-SAT showing significantly higher build times as instance size grows. The longer build times for the sequence variables model on PDPTW are likely due to the instantiation of global constraints prior to search. The VRP-specific solvers (VRSolver, PyVRP) are the fastest overall, with near-zero build times.

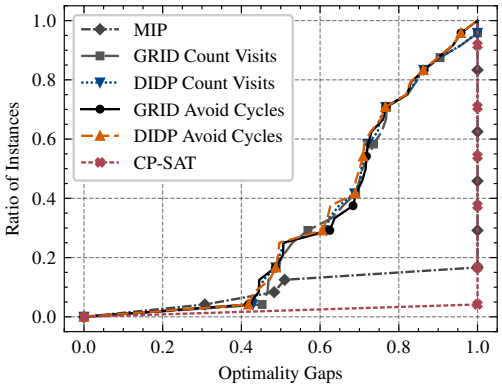
In addition to the solution quality analysis reported in the main text, we analyse the strength of the bounds proven by the solvers through the optimality gap, defined as the relative gap to the best objective value γ and the best lower bound $\underline{\gamma}$ achieved by a solver, $|\gamma - \underline{\gamma}|/\gamma$. Note that the optimality gap is only reported for solvers that expose a lower bound at the end of the solving process. Meta-heuristic solvers such as VRSolver and PyVRP do not provide this by nature, while for MaxiCP we were unable to retrieve this information from the available API. For PDPTW, as shown in Figure 2a, we observe that the MIP model solved with Gurobi achieves lower gaps on the instances where feasible solutions are found. However, it fails to find feasible solutions in harder instances, leading to DIDP and GRID having better optimality gaps for a higher ratio of instances. The CP model solved by CP-SAT achieves optimality gaps close to those of MIP, remaining well below DIDP and GRID across the majority of instances. For ECVRP (Figure 2b) DIDP and GRID outperform the MIP and CP-SAT model, most likely due to the increased size of the latter caused by the duplication of the stations. Finally, Figure 2c shows the optimality gaps for CVRP, where, as for PDPTW, the MIP model seems to be penalized by the lower number of solved instances.

Table 3 CVRP model building times (s).

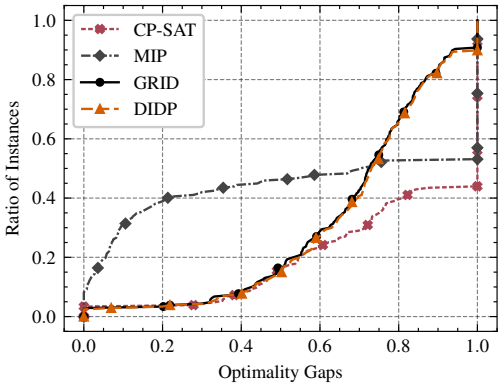
Size	DIDP	GRID	MIP	CP-SAT	VRSolver	PyVRP
$[1, 200]$	0.01	0.04	0.38	0.13	< 0.00	0.01
$[201, 400]$	0.05	0.44	4.08	1.64	< 0.00	0.17
$[401, 600]$	0.13	1.49	12.15	5.50	< 0.00	0.63
$[601, 800]$	0.23	2.74	23.43	10.50	< 0.00	1.57
$[801, 1000]$	0.36	4.62	42.41	19.33	< 0.00	3.20
$[1001, \infty)$	0.45	5.83	55.66	20.88	< 0.00	4.50



(a) PDPTW.



(b) ECVRP.



(c) CVRP.

Figure 2 Distribution of instances over the optimality gap at the time limit. A point (x, y) indicates that in $100 \times y$ percent of instances the optimality gap is at most x.