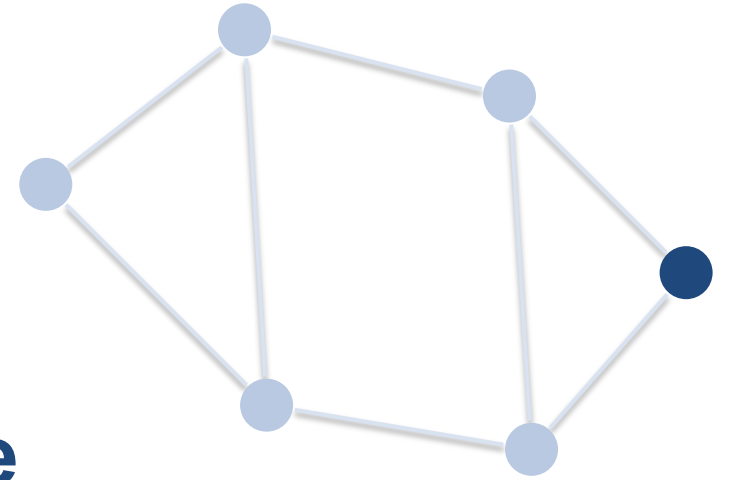




GRID: Graph-Based Modelling Interface for Domain-Independent Dynamic Programming

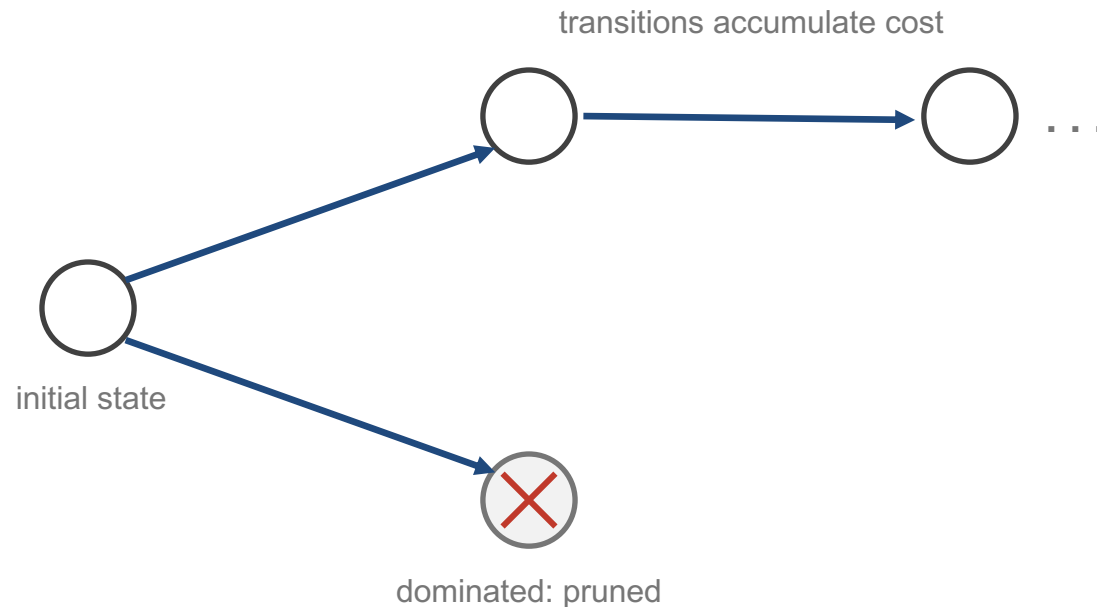
Fabio Giordana - Zeynep Kiziltan - Ryo Kuroiwa

University of Bologna - National Institute of Informatics / SOKENDAI

CP 2026 - Lisbon, 20-23 July 2026

Dynamic Programming for Combinatorial Optimization

Solutions built step by step, as a sequential decision process.



States & transitions

states summarize partial decisions -
transitions extend them, accumulating cost

Dominance relations

dominated states are pruned: often highly
effective exact algorithms

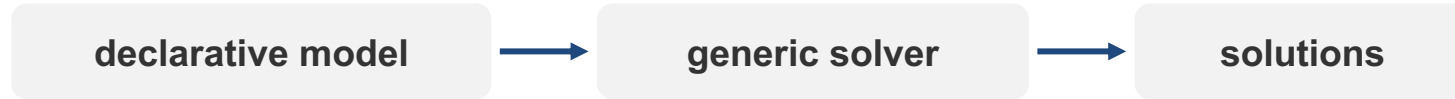
One algorithm per problem

classically, DP means hand-coding a
dedicated algorithm each time

Can the problem specification be separated from the search?

Model-Based Paradigms: MIP and CP

One recipe: a declarative model + a generic solver.



MIP

linear constraints and objective
generic branch-and-cut solvers
Gurobi, CPLEX

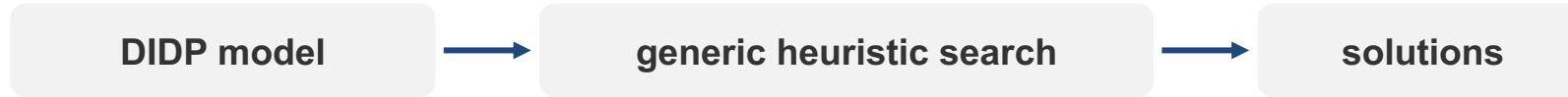
CP

global constraints, structured variables
high-level languages and libraries
MiniZinc, CPMPy

The same recipe for dynamic programming: DIDP.

DIDP: Domain-Independent Dynamic Programming

The model/solver separation, brought to dynamic programming.



A model, not an algorithm

state variables, transitions and dominance relations

Generic solvers

heuristic search over the state space - e.g. LNBS

Reusable across domains

the same solving technology for many problems

Modelling directly in DIDP is still low-level and error-prone.

State variables, transitions and dominance relations are written by hand - no high-level modelling layer, unlike CP and MIP.

Next: what a hand-written DIDPPy model looks like.

Modelling in DIDPPy: CVRP example

State (U, i, l, k): unvisited customers, current location, current load, vehicles used.

compute $V(N \setminus \{0\}, 0, 0, 1)$

$$V(U, i, l, k) = \begin{cases} \min \left\{ \min_{j \in U: l + d_i \leq q} c_{ij} + V(U \setminus \{j\}, j, l + d_i, k), \min_{j \in U} c_{i0} + c_{0j} + V(U \setminus \{j\}, j, d_i, k + 1) \right\} & \text{if } k < m \\ \min_{j \in U: l + d_i \leq q} c_{ij} + V(U \setminus \{j\}, j, l + d_i, k) & \text{if } k = m \\ c_{i0} + V(U, 0, k, l) & \text{if } U = \emptyset \wedge i \neq 0 \\ 0 & \text{if } U = \emptyset \wedge i = 0. \end{cases}$$

$$V(U, i, l, k) \leq V(U, i, l', k') \text{ if } l \leq l' \wedge k \leq k'.$$

resource variables

$$V(U, i, l, k) = \infty \text{ if } (m - k + 1)q - l < \sum_{j \in U} d_j.$$

state constraint

$$V(U, i, k, l) \geq \sum_{j \in (U \cup \{0\}) \setminus \{i\}} \min_{k \in N \setminus \{j\}} c_{kj}.$$

dual bounds

$$V(U, i, k, l) \geq \sum_{j \in (U \cup \{i\}) \setminus \{0\}} \min_{k \in N \setminus \{j\}} c_{jk}.$$

Modelling in DIDPPy: CVRP example

■ State variables

■ Transitions

■ Base case

■ State constraint & dual bounds

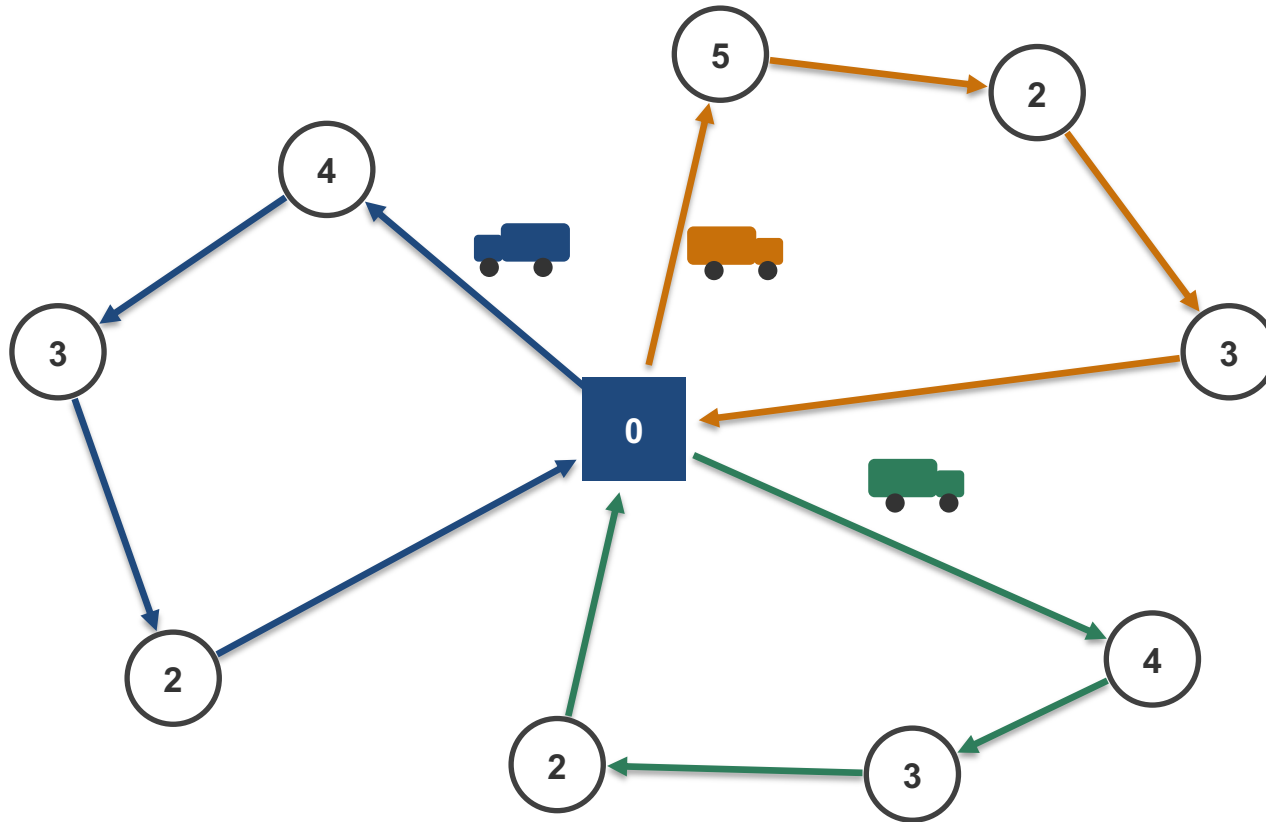
≈ 60 lines

```
1 # n: number of nodes (0: depot)
2 # m: number of vehicles
3 # q: vehicle capacity
4 # d[j]: demand of customer j
5 # c[i][j]: travel distance from node i to node j
6 import didppy as dp
7
8 model = dp.Model()
9 customer = model.add_object_type(number=n)
10 unvisited = model.add_set_var(object_type=customer, target=list(range(1, n)))
11 location = model.add_element_var(object_type=customer, target=0)
12 load = model.add_int_resource_var(target=0, less_is_better=True)
13 vehicles = model.add_int_resource_var(target=1, less_is_better=True)
14
15 weight = model.add_int_table(d)
16 distance = model.add_int_table(c)
17
18 model.add_base_case([unvisited.is_empty(), location == 0])
19
20 for j in range(1, n):
21     visit = dp.Transition(
22         name="visit {}".format(j),
23         cost=distance[location, j] + dp.IntExpr.state_cost(),
24         effects=[(unvisited, unvisited.remove(j)),
25                 (location, j),
26                 (load, load + weight[j])],
27         preconditions=[unvisited.contains(j), load + weight[j] <= q],
28     )
29     model.add_transition(visit)
```

```
30 for j in range(1, n):
31     visit_via_depot = dp.Transition(
32         name="visit {} with a new vehicle".format(j),
33         cost=distance[location, 0]+distance[0, j]+dp.IntExpr.state_cost(),
34         effects=[(unvisited, unvisited.remove(j)),
35                 (location, j),
36                 (load, weight[j]),
37                 (vehicles, vehicles + 1)],
38         preconditions=[unvisited.contains(j), vehicles < m],
39     )
40     model.add_transition(visit_via_depot)
41     return_to_depot = dp.Transition(
42         name="return",
43         cost=distance[location, 0] + dp.IntExpr.state_cost(),
44         effects=[(location, 0)],
45         preconditions=[unvisited.is_empty(), location != 0],
46     )
47     model.add_transition(return_to_depot)
48 model.add_state_constr((m - vehicles + 1) * q - load >= weight[unvisited])
49 min_distance_to = model.add_int_table(
50     [min(c[k][j] for k in range(n) if j != k) for j in range(n)])
51 model.add_dual_bound(
52     min_distance_to[unvisited]
53     + (location != 0).if_then_else(min_distance_to[0], 0))
54 min_distance_from = model.add_int_table(
55     [min(c[j][k] for k in range(n) if j != k) for j in range(n)])
56 model.add_dual_bound(
57     min_distance_from[unvisited]
58     + (location != 0).if_then_else(min_distance_from[location], 0))
```

GRID: Model the VRP on its Graph

Nodes, edges and vehicle types, instead of state variables and transitions.



customers with demands - three capacity-feasible routes

The routing graph $G = (N, E)$

Locations $\rightarrow$ nodes

Vehicle trips $\rightarrow$ edges

Fleet $\rightarrow$ vehicle types

Describe the graph and its attributes: the DP model is derived automatically.

GRID model (Python)

Compiler

DIDPPy model

DIDP solver

Modelling Elements

One container, three entity classes. Everything else is attributes.

RoutingModel



Node

demand
time window, service time
optional, max_visits
depot flag



Edge

distance
travel time



VehicleType

count
capacity
user-defined variables

add_node()

add_edge()

add_vehicle_type()

set_objective()

solve()

Entities are created once and stored. Attributes carry the semantics of the variant.

Native Features: CVRP in GRID

```
1  # m: number of vehicles, q: vehicle capacity
2  # customers: set of customers i
3  # mu[i]: demand of customer i
4  # edges: set of edges (i,j) with distance c[i][j]
5
6  model = RoutingModel()
7  model.add_vehicle_type(id=0, count=m, capacity=q)
8  model.add_node(id=0, depot=True)
9
10 for i in customers:
11     model.add_node(
12         id=i,
13         demand=mu[i],
14     )
15
16 for (i, j) in edges:
17     model.add_edge(
18         node_from=model.get_node(i), node_to=model.get_node(j),
19         distance=c[i][j],
20     )
21
22 model.set_objective(metric="distance")
```

capacity= → **attribute on VehicleType**
declared once, for the whole fleet

`model.add_vehicle_type(id=0, count=m, capacity=q)`

demand= → **attribute on Node**
one node per customer

`model.add_node(
 id=i,
 demand=mu[i],
)`

metric="distance" → **built-in objective**
total distance, minimized over all routes

`model.set_objective(metric="distance")`

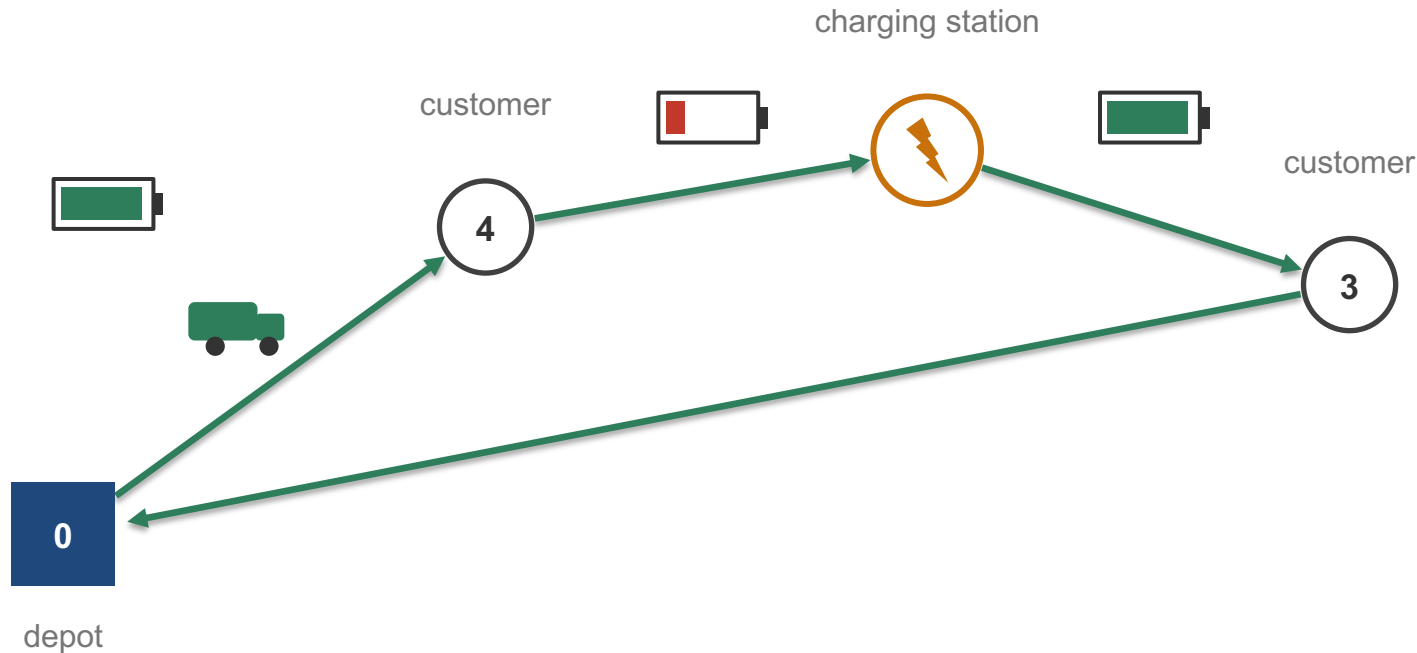
Capacity handling, route structure and dual bounds are generated automatically.

The whole model, no transitions written.

PDPTW is modelled the same way (Listing 1 in the paper).

Custom Features: ECVRP Battery Constraint

Electric vehicles: limited battery, recharged at charging stations.



Discharge depends on the load

the heavier the vehicle, the faster the battery drains

Recharge at stations

optional nodes - multiple visits allowed

Beyond fixed “dimensions”

difficult to express

battery decreases by $\text{distance} \cdot (r + \text{load} / \text{capacity})$ on every edge

A coupling between two evolving quantities: exactly what custom features are for.

Custom Features: ECVRP Battery Constraint

```
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)
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")
```

Route state lives on the VehicleType
load (integer) - battery (float, prefer high)

load = v.add_integer_var(initial_value=q)
battery = v.add_float_var(initial_value=b, preference="high")

Updates and checks on nodes and edges
add_transition - add_constraint

customer.add_constraint(load - mu[i] >= 0)
customer.add_transition(var=load, expression=load - mu[i])

Expressions between variables
the battery update depends on the current load

next_battery = battery - d[i][j] * (r + load / q)
edge.add_transition(var=battery, expression=next_battery)
edge.add_constraint(next_battery >= 0)

Omitted here: the station-cycle pruning with a set variable. Listings 2–3 in the paper.

From GRID to DIDPPy

One generic compilation scheme: routes are built sequentially, one vehicle at a time.

GRID model and semantics

Mandatory nodes (customers)

Current position

Fleet of vehicles

User-defined variables: load, battery, ...

Objective metric

DIDPPy model

set variable U : unvisited nodes

element variable i : current node

element variable v : current vehicle (resource, lower better)

numeric / element / set state variables, with preferences

transition weights + dual bound function

Transitions

Visit j

Change vehicle

Finish

Base case: $U = \emptyset$, vehicle back at its depot

Native features add resource variables, state constraints and dual bounds automatically.

Experimental Validation

Three goals: overhead, model quality, and solving performance.

1

Compilation overhead: how long does GRID take to build models?

2

Model quality: GRID-compiled vs hand-crafted DIDPPy models

3

Solving performance vs CP, MIP and VRP-specific solvers

Variants & benchmarks

CVRP

CVRPLIB (A, B, E, F, M, P, X, DIMACS)

PDPTW

Li & Lim instances, 100 to 1000+ nodes

ECVRP

Mavrovouniotis et al. (CEC 2020)

Model-based — accept custom constraints and objectives

DP

DIDP (hand-crafted)

MIP

Gurobi

CP

OR-Tools CP-SAT

CP + sequence vars

MaxiCP

VRP-specific

VRP-specific

PyVRP

OR-Tools VRSolver

Per run: single thread - 5-minute time limit - 8 GB memory.

Results: model building time

< 2 min

GRID model building in ECVRP, 1000+ nodes

≈ 6 s

GRID model building in CVRP and PDPTW, 1000+ nodes

Expression aggregation

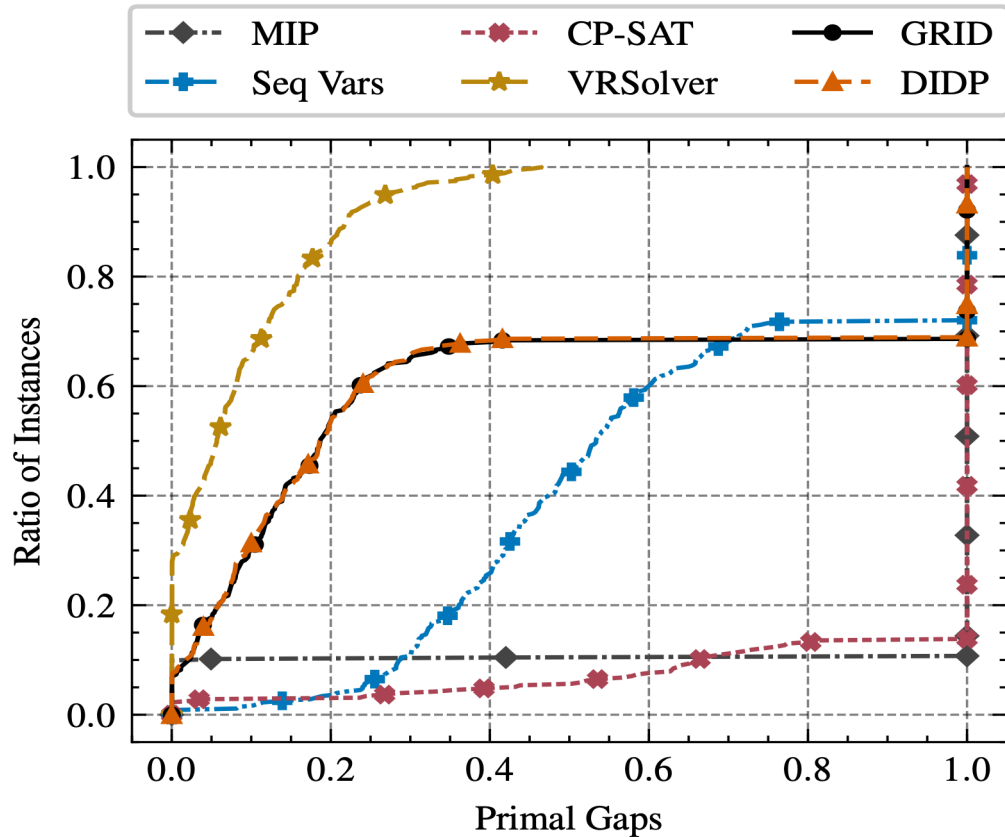
parsing and aggregating the edge-update expressions
dominates the ECVRP overhead

MIP & CP-SAT

run out of memory on large ECVRP instances: duplicated
charging stations inflate the model

Full building times: Table 1 in the paper.

PDPTW: primal gap distribution



VRSolver clearly leads

guided local search pays off on this classical variant

GRID $\approx$ hand-crafted DIDP

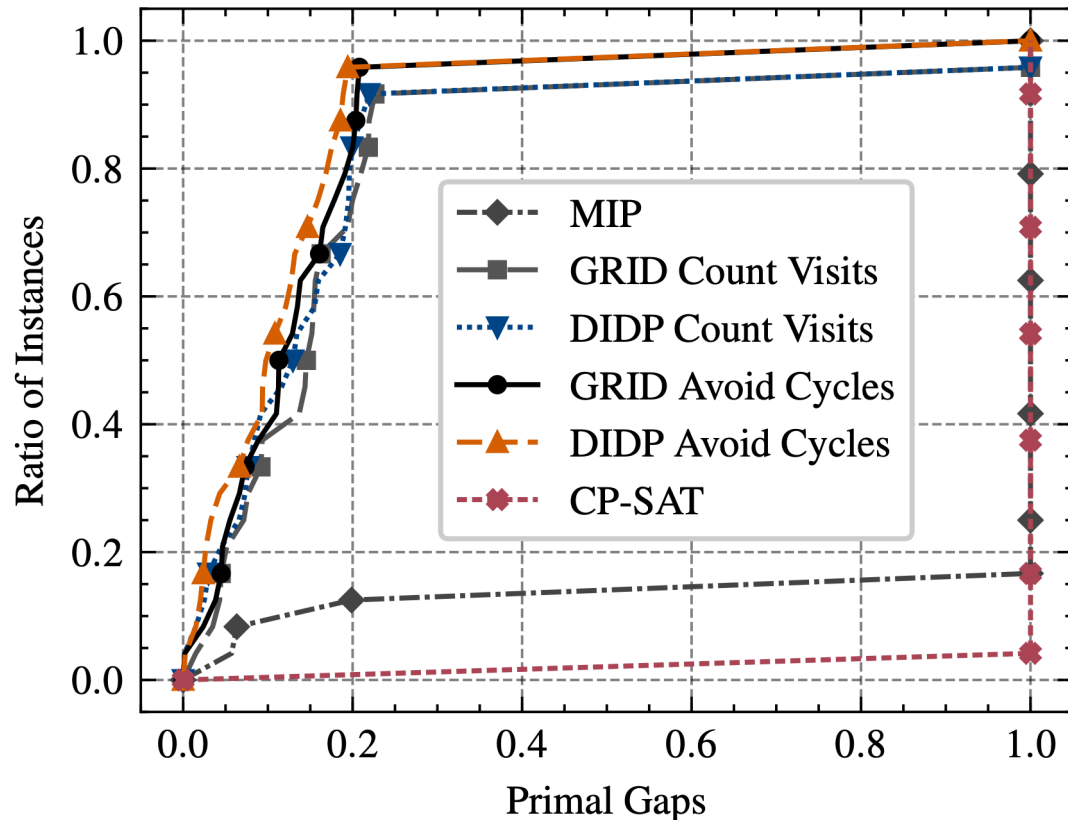
the two curves stay close over the whole instance set

Other approaches remain behind

sequence variables run out of memory on larger instances

Figure 1a in the paper. Ratio of instances (y) reaching a primal gap of at most x .

ECVRP: primal gap distribution



Only model-based approaches apply

PyVRP, VRSolver and sequence variables cannot express the battery

DIDP models clearly ahead

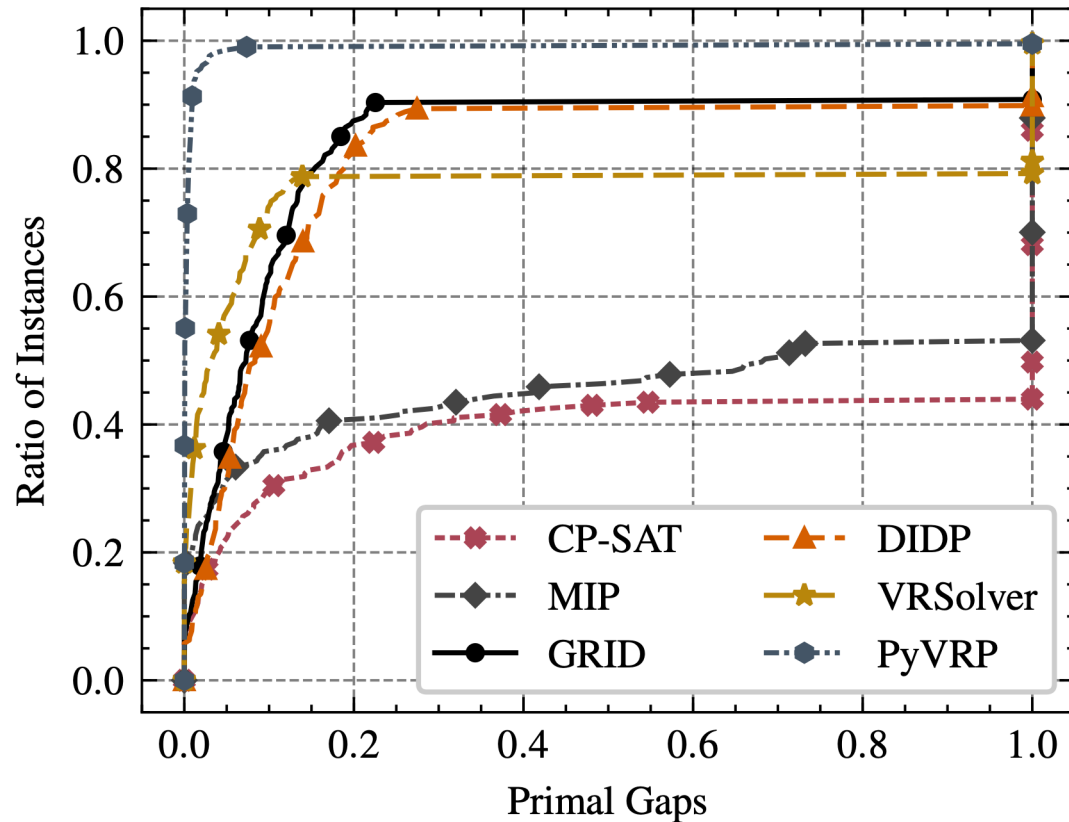
GRID and hand-crafted DIDP outperform MIP and CP-SAT

Avoid Cycles beats Count Visits

user-defined expressions inject domain knowledge - and pay off

Figure 1b in the paper. Ratio of instances (y) reaching a primal gap of at most x.

CVRP: primal gap distribution



PyVRP clearly best

state-of-the-art heuristics on the classical variant

VRSolver strong early, then fades

ends slightly below GRID and DIDP over all instances

GRID slightly ahead of prior DIDP

the formulation differences have little impact overall

Figure 1c in the paper. Ratio of instances (y) reaching a primal gap of at most x .

Results: primal gaps, side by side

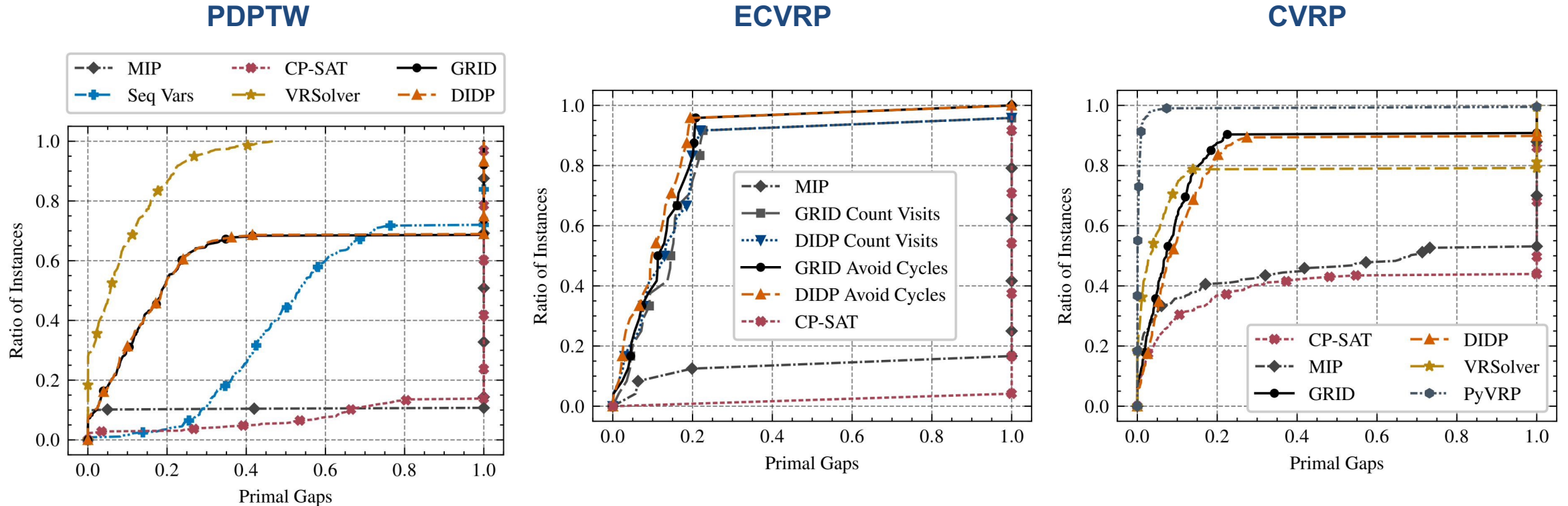


Figure 1 in the paper. Ratio of instances (y) reaching a primal gap of at most x.

GRID $\approx$ hand-crafted DIDP. Both outperform MIP and CP-SAT.

VRP-specific solvers lead on classical variants, but cannot express the battery constraint

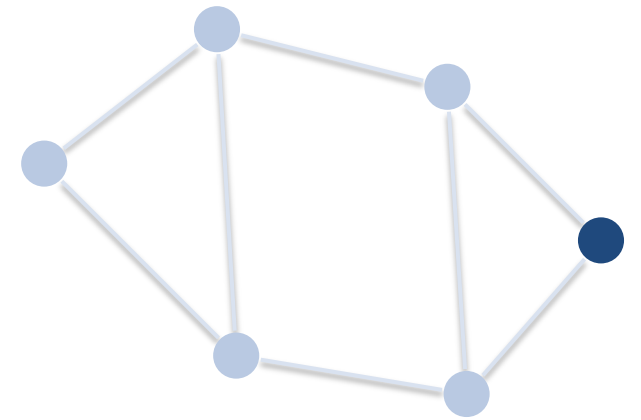
Conclusions & Future Work

Conclusions

- High-level, declarative, DP-oriented modelling on a routing graph
- Automatic compilation to DIDPPy with performance preserved vs native DIDP models
- No solver dominates across the three variants: GRID is competitive on all of them

Future work

- More solver back-ends: other DP solvers, then CP, MIP, VRP solvers
- Beyond VRPs: related sequencing problems - e.g. job scheduling
- An improved compilation layer



Thank you!

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

Fabio Giordana · Zeynep Kiziltan · Ryo Kuroiwa

Paper

[doi.org/10.4230/
LIPIcs.CP.2026.26](https://doi.org/10.4230/LIPIcs.CP.2026.26)



GRID

[pygridopt.
readthedocs.io](https://pygridopt.readthedocs.io)

