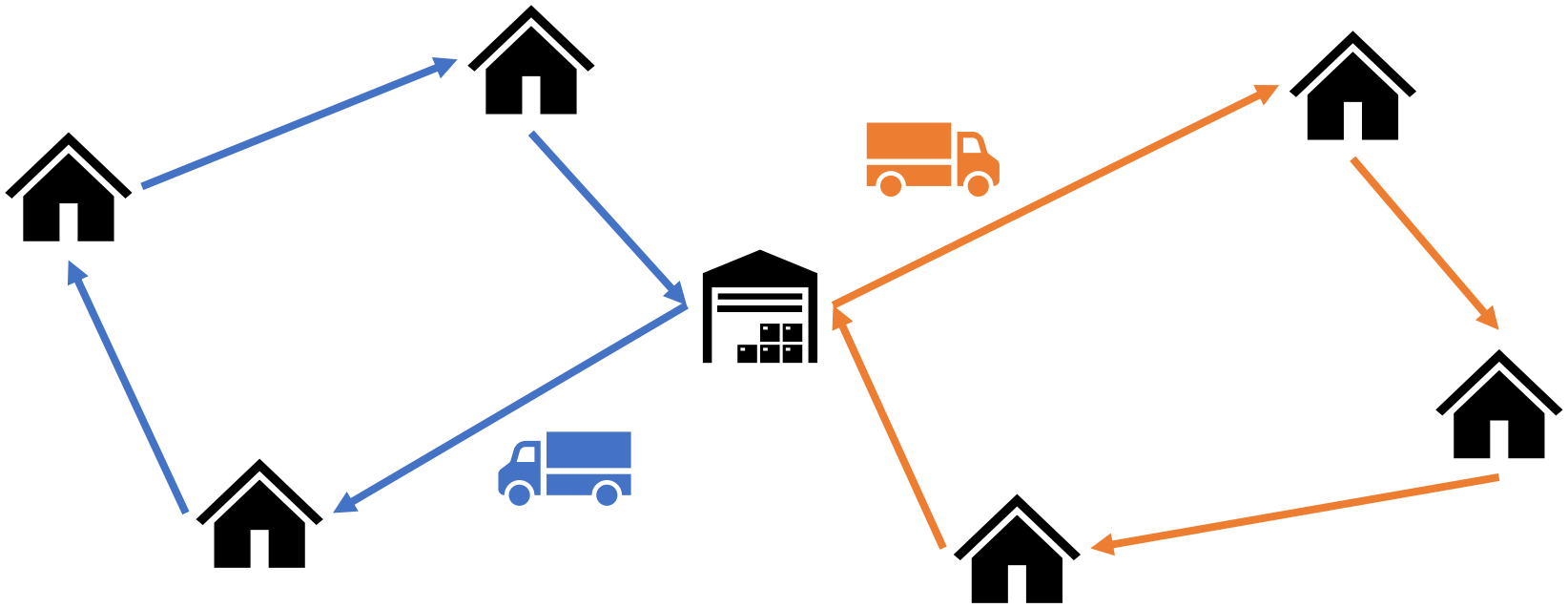
Column Generation with Domain-Independent Dynamic Programming

Ryo Kuroiwa, National Institute of Informatics / SOKENDAI, Japan
Edward Lam, Monash University, Australia

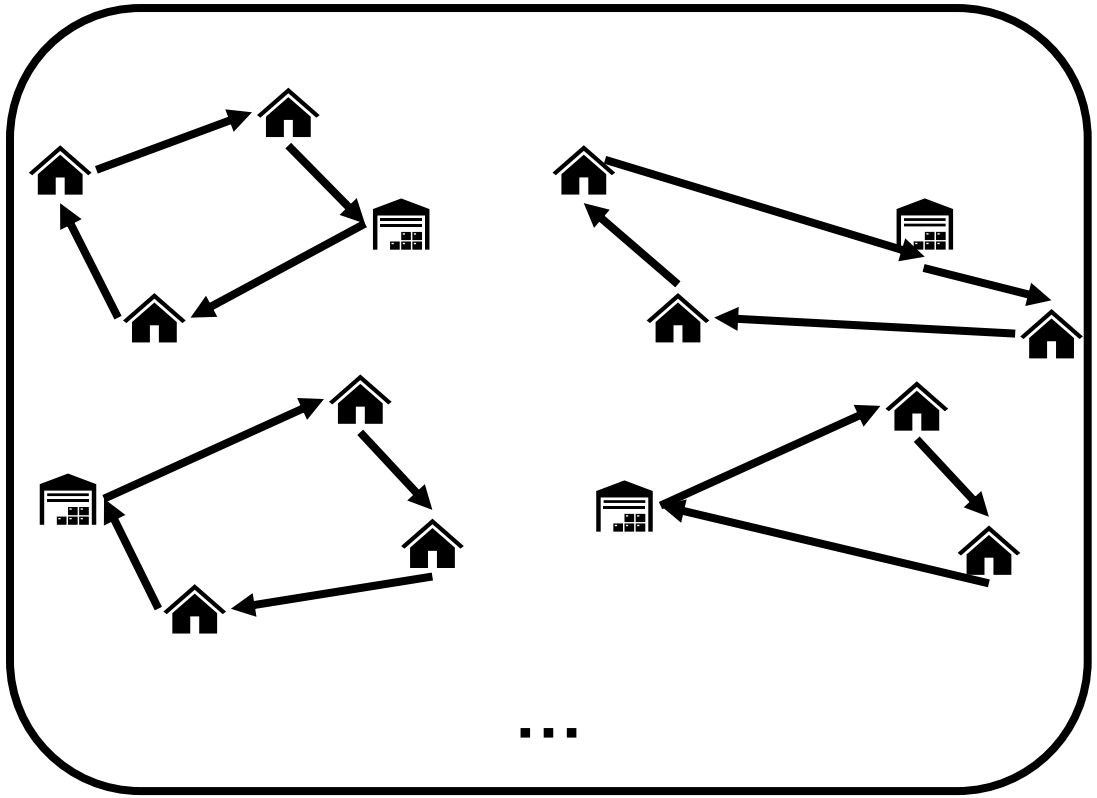
Example: Vehicle Routing Problems (VRPs)

- Visit all customers using multiple vehicles
- Objective based on tours, e.g., minimizing the total travel distance
- Side constraints, e.g., vehicle capacity and time windows



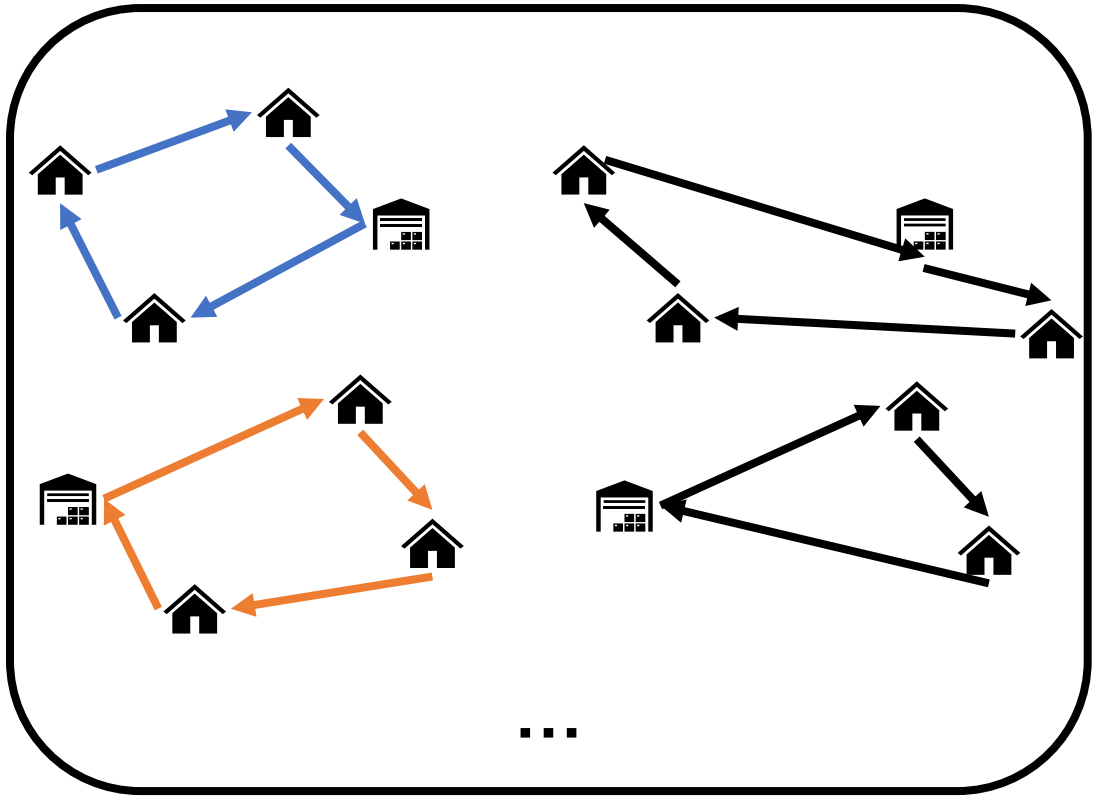
Set Partitioning Formulation for a VRP

Set of paths satisfying side-constraints
for a single vehicle

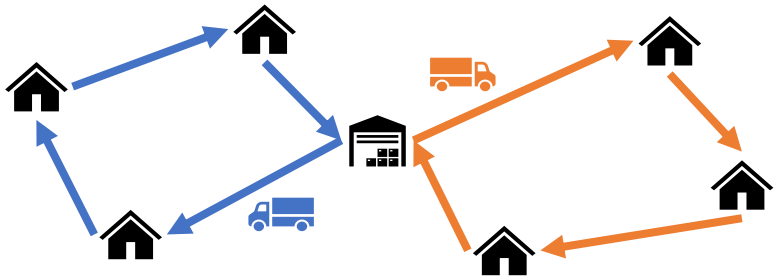


Set Partitioning Formulation for a VRP

Set of paths satisfying side-constraints for a single vehicle



$|K|$ paths visits each customer exactly once



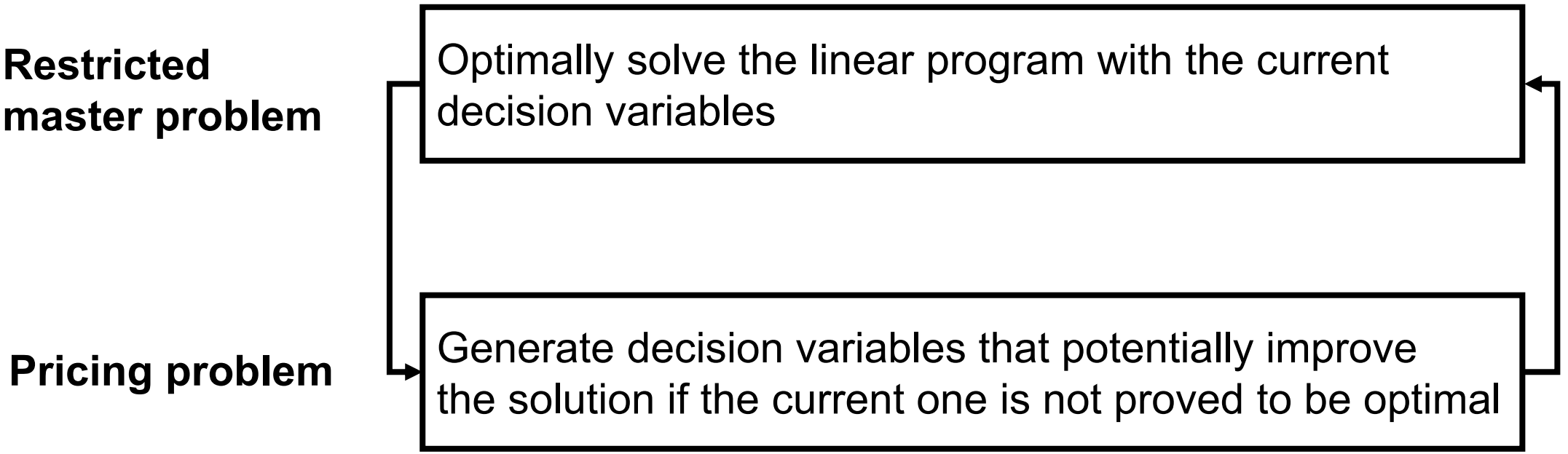
$$\min \sum_{p \in \mathcal{P}} c_p \lambda_p \qquad (9a)$$

$$\sum_{p \in \mathcal{P}} a_{p,j} \lambda_p = 1 \quad \forall j = 1, \ldots, n \qquad (9b)$$

$$\lambda_p \in \mathbb{Z}_+ \qquad \forall p \in \mathcal{P}. \qquad (9c)$$

Column Generation

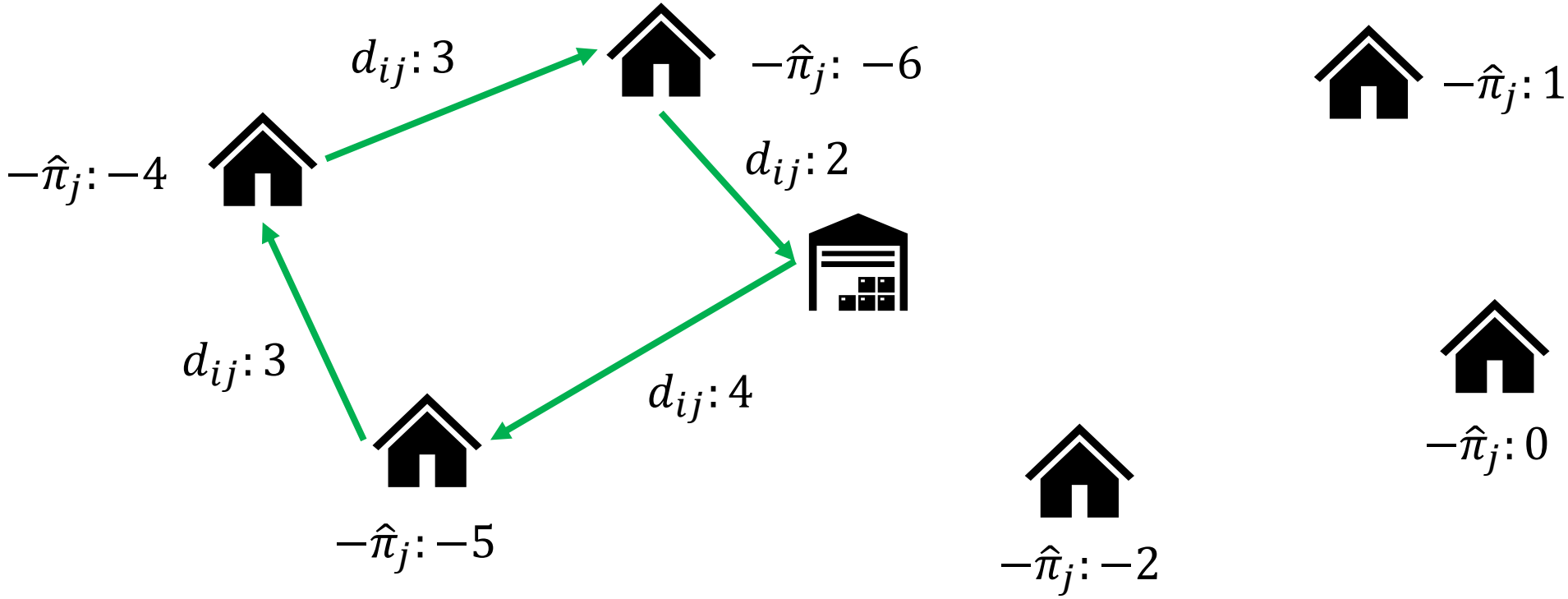
State-of-the art exact approach for multiple problems including VRPs



Combination with branch-and-bound (**branch-and-price**) is necessary to find an optimal integer solution

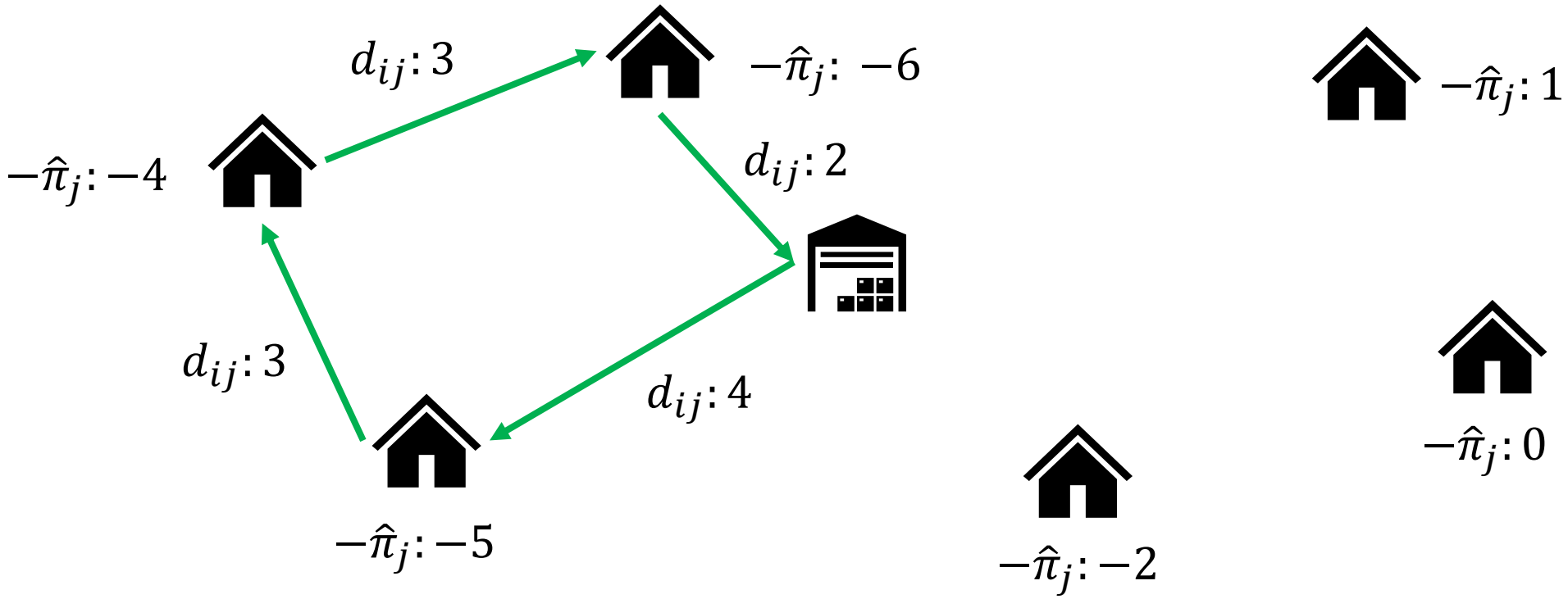
Pricing Problem: a VRP as an Example

- Find a path starting and ending at the depot, visiting each customer at most once, and satisfying side-constraints
- Minimize the total arc cost defined by $-\hat{\pi}_j + d_{ij}$ (can be negative), where $\hat{\pi}_j$ is a dual value given by the restricted master problem



Solving Pricing Problem

- Problem-specific dynamic programming algorithms are very efficient but require high implementation effort
- Declarative solving paradigms such as MIP and CP?



Domain-Independent Dynamic Programming (DIDP)

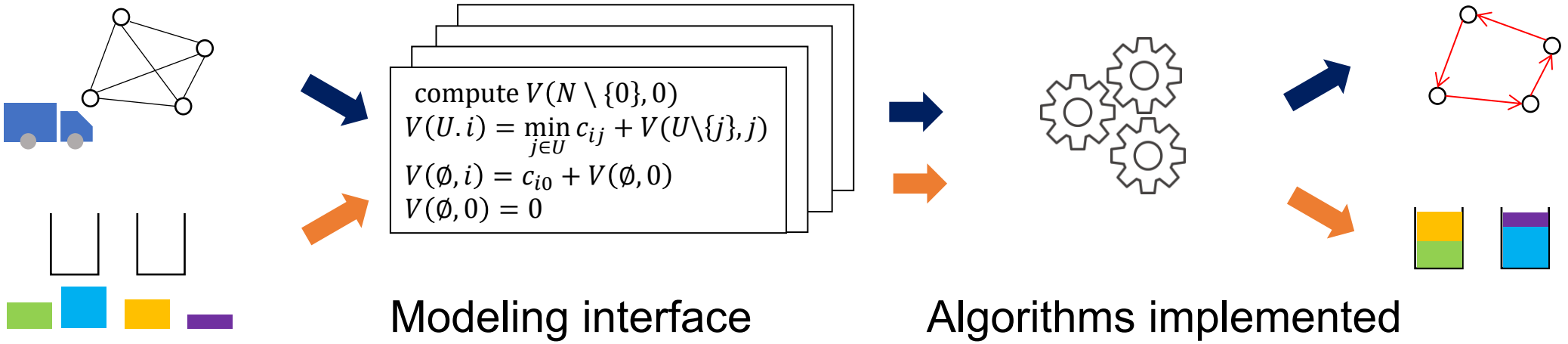
General-purpose solver framework for dynamic programming (DP), similarly to a general-purpose CP solver

Combinatorial Optimization Problem

DP Model

General-Purpose DP Solver

Solution



Modeling interface such as Python library

Algorithms implemented in Rust

[K and Beck AIJ 2026]

DP for the Constrained Shortest Path Problem

$V(i, R, l)$: the shortest path cost from current location i to the depot

- R: the set of reachable locations
- l: the cumulative load, increased by w_j when location j is visited, with the capacity constraint $l \leq q$

DP for the Constrained Shortest Path Problem

$V(i, R, l)$: the shortest path cost from current location i to the depot

- R: the set of reachable locations
- l: the cumulative load, increased by w_j when location j is visited, with the capacity constraint $l \leq q$

$$V(i, R, l) = \begin{cases} 0 & \text{if } i = n + 1 \\ \min\limits_{j \in R: l + w_j \leq q} -\hat{\pi}_j + d_{ij} + V(j, R \setminus \{j\}, l + w_j) & \text{otherwise} \end{cases}$$

DP for the Constrained Shortest Path Problem

$V(i, R, l)$: the shortest path cost from current location i to the depot

- R: the set of reachable locations
- l: the cumulative load, increased by w_j when location j is visited, with the capacity constraint $l \leq q$

Reached the end depot

$$V(i, R, l) = \begin{cases} 0 & \text{if } i = n + 1 \\ \min\limits_{j \in R: l + w_j \leq q} -\hat{\pi}_j + d_{ij} + V(j, R \setminus \{j\}, l + w_j) & \text{otherwise} \end{cases}$$

DP for the Constrained Shortest Path Problem

$V(i, R, l)$: the shortest path cost from current location i to the depot

- R: the set of reachable locations
- l: the cumulative load, increased by w_j when location j is visited, with the capacity constraint $l \leq q$

$$V(i, R, l) = \begin{cases} 0 & \text{if } i = n + 1 \\ \min\limits_{j \in R: l + w_j \leq q} -\hat{\pi}_j + d_{ij} + V(j, R \setminus \{j\}, l + w_j) & \text{otherwise} \end{cases}$$

Visiting location j

DP for the Constrained Shortest Path Problem

$V(i, R, l)$: the shortest path cost from current location i to the depot

- R: the set of reachable locations
- l: the cumulative load, increased by w_j when location j is visited, with the capacity constraint $l \leq q$

$$V(i, R, l) = \begin{cases} 0 & \text{if } i = n + 1 \\ \min\limits_{j \in R: l + w_j \leq q} -\hat{\pi}_j + d_{ij} + V(j, R \setminus \{j\}, l + w_j) & \text{otherwise} \end{cases}$$

Select the best next location j

DP for the Constrained Shortest Path Problem

$V(i, R, l)$: the shortest path cost from current location i to the depot

- R: the set of reachable locations
- l: the cumulative load, increased by w_j when location j is visited, with the capacity constraint $l \leq q$

$$V(i, R, l) = \begin{cases} 0 & \text{if } i = n + 1 \\ \min\limits_{j \in R: l + w_j \leq q} -\hat{\pi}_j + d_{ij} + V(j, R \setminus \{j\}, l + w_j) & \text{otherwise} \end{cases}$$

By solving the equation and computing $V(0, N \cup \{n + 1\}, 0)$, an optimal solution for the pricing problem is obtained

Modeling Example in DIDPPy

https://didppy.rtfd.io

```
model = dp.Model(maximize=False, float_cost=True)
node = model.add_object_type(number=n)
d = model.add_int_table(travel_time)

i = model.add_element_var(object_type=node, target=0)
r = model.add_set_var(object_type=node, target=list(range(1, n + 1)))
l = model.add_int_var(target=0)  # noqa: E741

for j in range(1, n + 1):
    visit = dp.Transition(
        name=f"visit {j}",
        cost=-pi[j] + d[i, j] + dp.FloatExpr.state_cost(),
        effects=[(i, j), (r, r.remove(j)), (l, l + w[j])],
        preconditions=[r.contains(j), l + w[j] <= q],
    )
    model.add_transition(visit)

model.add_base_case([i == n + 1], cost=0)
```

Modeling Example in DIDPPy

https://didppy.rtfd.io

```
model = dp.Model(maximize=False, float_cost=True)
node = model.add_object_type(number=n)
d = model.add_int_table(travel_time)
```

State variables (i, R, l)

```
i = model.add_element_var(object_type=node, target=0)
r = model.add_set_var(object_type=node, target=list(range(1, n + 1)))
l = model.add_int_var(target=0)  # noqa: E741

for j in range(1, n + 1):
    visit = dp.Transition(
        name=f"visit {j}",
        cost=-pi[j] + d[i, j] + dp.FloatExpr.state_cost(),
        effects=[(i, j), (r, r.remove(j)), (l, l + w[j])],
        preconditions=[r.contains(j), l + w[j] <= q],
    )
    model.add_transition(visit)

model.add_base_case([i == n + 1], cost=0)
```

Modeling Example in DIDPPy

https://didppy.rtfd.io

```
model = dp.Model(maximize=False, float_cost=True)
node = model.add_object_type(number=n)
d = model.add_int_table(travel_time)

i = model.add_element_var(object_type=node, target=0)
r = model.add_set_var(object_type=node, target=list(range(1, n + 1)))
l = model.add_int_var(target=0)  # noqa: E741

for j in range(1, n + 1):
    visit = dp.Transition(
        name=f"visit {j}",
        cost=-pi[j] + d[i, j] + dp.FloatExpr.state_cost(),
        effects=[(i, j), (r, r.remove(j)), (l, l + w[j])],
        preconditions=[r.contains(j), l + w[j] <= q],
    )
    model.add_transition(visit)

model.add_base_case([i == n + 1], cost=0)
```

$-\hat{\pi}_j + d_{ij} + V(j, R \setminus \{j\}, l + w_j)$
for $j \in R$ and $l + w_j \leq q$

Modeling Example in DIDPPy

https://didppy.rtfd.io

```
model = dp.Model(maximize=False, float_cost=True)
node = model.add_object_type(number=n)
d = model.add_int_table(travel_time)

i = model.add_element_var(object_type=node, target=0)
r = model.add_set_var(object_type=node, target=list(range(1, n + 1)))
l = model.add_int_var(target=0)  # noqa: E741

for j in range(1, n + 1):
    visit = dp.Transition(
        name=f"visit {j}",
        cost=-pi[j] + d[i, j] + dp.FloatExpr.state_cost(),
        effects=[(i, j), (r, r.remove(j)), (l, l + w[j])],
        preconditions=[r.contains(j), l + w[j] <= q],
    )
    model.add_transition(visit)

model.add_base_case([i == n + 1], cost=0)
```

$V(i, R, t) = 0$ if $i = n + 1$

Dominance in DIDP

- Having less load is better
  $V(i, R, l) \leq V(i, R, l')$ if $l \leq l'$
  Modeled by an integer resource state variable in the current DIDP

```
l = model.add_int_resource_var(target=0, less_is_better=True)
```

Dominance in DIDP

- Having less load is better
  $V(i, R, l) \leq V(i, R, l')$ if $l \leq l'$
  Modeled by an integer resource state variable in the current DIDP

```
l = model.add_int_resource_var(target=0, less_is_better=True)
```

- Having more reachable locations is better in addition
  $V(i, R, l) \leq V(i, R', l')$ if $R \supseteq R' \wedge l \leq l'$
  Modeled by a set resource state variable (a **new feature**)

```
r = model.add_set_resource_var(
    object_type=node, target=list(range(1, n + 1)), less_is_better=False
)
```

New Feature: Filtering Unreachabe Locations

A location cannot be reached considering the capacity after visiting j
$R \leftarrow \{k \in R \setminus \{j\} \mid l + w_j + w_k \leq q\}$

Implemented by a filtering operation (a **new feature**)

```
k = model.add_local_var()

for j in range(1, n + 1):
    visit = dp.Transition(
        name=f"visit {j}",
        cost=-pi[j] + d[i, j] + dp.FloatExpr.state_cost(),
        effects=[
            (i, j),
            (r, r.remove(j).filter(k, l + w[j] + w[k] <= q)),
            (l, l + w[j]),
        ],
        preconditions=[r.contains(j), l + w[j] <= q],
    )
    model.add_transition(visit)
```

New Feature: Filtering Unreachabe Locations

A location cannot be reached considering the capacity after visiting j
$R \leftarrow \{k \in R \setminus \{j\} \mid l + w_j + w_k \leq q\}$

Implemented by a filtering operation (a **new feature**)

```
k = model.add_local_var()

for j in range(1, n + 1):
    visit = dp.Transition(
        name=f"visit {j}",
        cost=-pi[j] + d[i, j] + dp.FloatExpr.state_cost(),
        effects=[
            (i, j),
            (r, r.remove(j).filter(k, l + w[j] + w[k] <= q)),
            (l, l + w[j]),
        ],
        preconditions=[r.contains(j), l + w[j] <= q],
    )
    model.add_transition(visit)
```

New Feature: Fractional Knapsack Bound

Given a state (i, R, l), a lower bound on the shortest path cost can be obtained by solving the following knapsack problem:

$$\min \sum_{j \in R} -\hat{\pi}_j x_j$$

$$\text{s.t.} \sum_{j \in R} w_j \, x_j \leq q - l$$

$$x_j \in \{0,1\}, \forall j \in R$$

New Feature: Fractional Knapsack Bound

Given a state (i, R, l), a lower bound on the shortest path cost can be obtained by solving the following knapsack problem:

$$\min \sum_{j \in R} -\hat{\pi}_j x_j$$

$$\text{s.t.} \sum_{j \in R} w_j x_j \leq q - l$$

$$x_j \in \{0,1\}, \forall j \in R$$

The linear relaxation can be solved in $O(n)$ for each state (with sorting in preprocessing)

```
model.add_dual_bound(-dp.fractional_knapsack(r, q - l, pi, w))
```

Experimental Evaluation

Our branch-and-price implementations (fully in Python):

- **B&P DIDP**: SCIP 9.2.1 for master and DIDP for pricing, using a new solver, labeling, developed in this work
- **B&P MIP**: SCIP 9.2.1 for master and Gurobi 13.0.2 for pricing
- **B&P CP**: SCIP 9.2.1 for master and OR-Tools CP-SAT for pricing

Baselines:

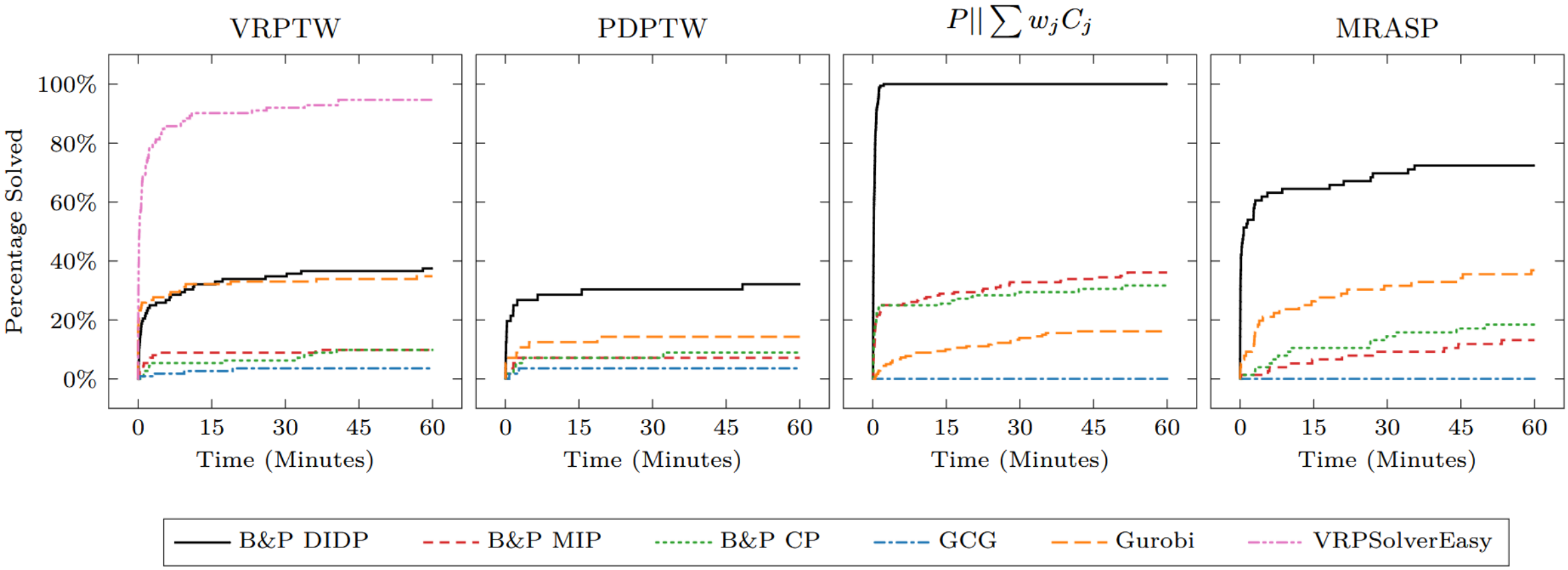
- **GCG** 3.5.5: column generation solver with automatic reformulation
- **VRPSolverEasy**: column generation solver specific to VRPs
- **Gurobi** 13.0.2: directly solve a compact MIP formulation

Benchmark Problems

- VRP with time windows (VRPTW)
- Pickup-and-delivery problem with time windows (PDPTW)
- Parallel machine scheduling to minimize the total weighted completion time ($P||\sum w_j C_j$)
  Pricing problem: knapsack with time window constraints and time-dependent profit
- Multi-runway aircraft scheduling problem (MRASP)
  Pricing problem: similar to single machine scheduling to minimize the total weighted completion time with sequence dependent setup time

VRPSolverEasy is applicable to only VRPTW (to our knowledge)

Comparison with Baselines



Comparing Different Pricing Solvers

B&P MIP vs. B&P DIDP	#iterations	Avg. #columns	Avg. time (s)
VRPTW	2.71 ± 1.13	0.26 ± 0.28	3.70 ± 2.74
PDPTW	1.83 ± 0.68	0.56 ± 0.16	36.90 ± 35.81
$P\|\|\sum w_j C_j$	0.56 ± 0.08	4.61 ± 0.51	199.71 ± 359.81
MRASP	1.11 ± 0.19	0.66 ± 0.12	895.38 ± 816.00

B&P CP vs. B&P DIDP	#iterations	Avg. #columns	Avg. time (s)
VRPTW	2.05 ± 0.61	0.25 ± 0.21	27.35 ± 29.75
PDPTW	1.41 ± 0.42	0.56 ± 0.16	72.22 ± 52.96
$P\|\|\sum w_j C_j$	0.50 ± 0.06	5.66 ± 1.20	174.55 ± 351.63
MRASP	1.06 ± 0.16	0.89 ± 0.16	545.00 ± 500.12

Relative values: the values by B&P MIP/CP divided by the values by B&P DIDP

Comparing Different Pricing Solvers

B&P MIP vs. B&P DIDP	#iterations	Avg. #columns	Avg. time (s)
VRPTW	2.71 ± 1.13	0.26 ± 0.28	3.70 ± 2.74
PDPTW	1.83 ± 0.68	0.56 ± 0.16	36.90 ± 35.81
$P\|\|\sum w_j C_j$	0.56 ± 0.08	4.61 ± 0.51	199.71 ± 359.81
MRASP	1.11 ± 0.19	0.66 ± 0.12	895.38 ± 816.00
B&P CP vs. B&P DIDP	**#iterations**	**Avg. #columns**	**Avg. time (s)**
VRPTW	2.05 ± 0.61	0.25 ± 0.21	27.35 ± 29.75
PDPTW	1.41 ± 0.42	0.56 ± 0.16	72.22 ± 52.96
$P\|\|\sum w_j C_j$	0.50 ± 0.06	5.66 ± 1.20	174.55 ± 351.63
MRASP	1.06 ± 0.16	0.89 ± 0.16	545.00 ± 500.12

MIP/CP requires more pricing iterations generating fewer columns in each except for $P||\sum w_j\, C_j$

Comparing Different Pricing Solvers

B&P MIP vs. B&P DIDP	#iterations	Avg. #columns	Avg. time (s)
VRPTW	2.71 ± 1.13	0.26 ± 0.28	3.70 ± 2.74
PDPTW	1.83 ± 0.68	0.56 ± 0.16	36.90 ± 35.81
$P\|\|\sum w_j C_j$	0.56 ± 0.08	4.61 ± 0.51	199.71 ± 359.81
MRASP	1.11 ± 0.19	0.66 ± 0.12	895.38 ± 816.00
B&P CP vs. B&P DIDP	**#iterations**	**Avg. #columns**	**Avg. time (s)**
VRPTW	2.05 ± 0.61	0.25 ± 0.21	27.35 ± 29.75
PDPTW	1.41 ± 0.42	0.56 ± 0.16	72.22 ± 52.96
$P\|\|\sum w_j C_j$	0.50 ± 0.06	5.66 ± 1.20	174.55 ± 351.63
MRASP	1.06 ± 0.16	0.89 ± 0.16	545.00 ± 500.12

MIP/CP takes much more time in each pricing iteration

Conclusion

- Declarative column generation with DIDP pricing, outperforming MIP and CP pricing
- DIDP augmented with new modeling features (will be merged soon), but still far from state-of-the-art

Tutorials and API References
https://didppy.rtfd.io

Project Page
https://didp.ai

GitHub Repo
https://github.com/domain-independent-dp/didp-rs

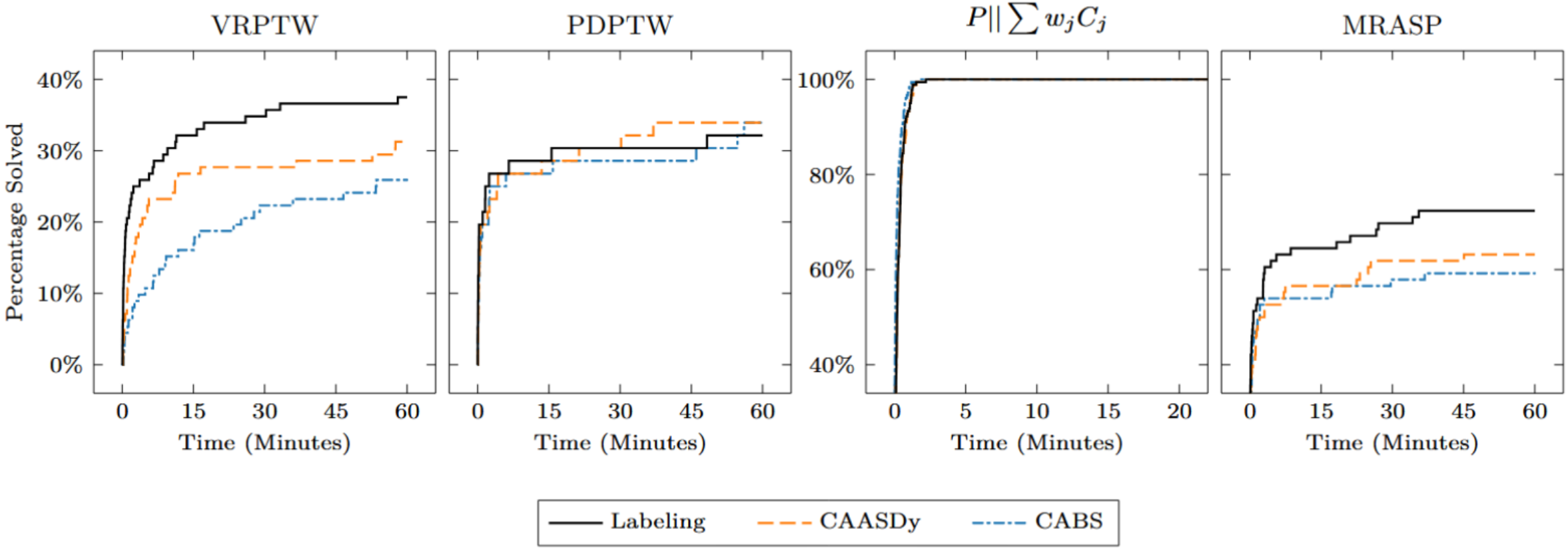
DIDP Solvers

Existing solvers from the AI community:

- A*
- Complete Anytime Beam Search (CABS)

Inspired by problem-specific pricing algorithms, we propose a **new solver, labeling**, which explores using the lexicographic order of resource variable values

Comparison of Different DIDP Solvers



Comparison of Different DIDP Solvers

CAASDy vs. labeling	#iterations	Avg. #columns	Avg. time (s)	Avg. #expansions
VRPTW	4.78 ± 1.73	0.10 ± 0.13	0.99 ± 0.18	0.87 ± 0.13
PDPTW	3.52 ± 0.79	0.12 ± 0.04	0.63 ± 0.15	0.42 ± 0.07
MRASP	2.39 ± 0.80	0.15 ± 0.05	3.92 ± 8.99	1.69 ± 2.25

CABS vs. labeling	#iterations	Avg. #columns	Avg. time (s)	Avg. #expansions
VRPTW	1.51 ± 0.50	0.40 ± 0.23	17.93 ± 21.78	19.39 ± 18.97
PDPTW	1.42 ± 0.29	0.44 ± 0.05	1.63 ± 0.46	1.44 ± 0.43
$P\|\|\sum w_j C_j$	0.76 ± 0.16	2.02 ± 0.34	0.53 ± 0.32	2.66 ± 0.37
MRASP	1.26 ± 0.28	0.48 ± 0.09	14.85 ± 47.59	7.89 ± 13.62

Column Generation Solvers

Approach	Characteristic
GCG	• Automatic decomposition of a compact MILP model • Solve pricing problems using MILP by default • Part of SCIP
Coluna	• Automatic decomposition of a compact MILP model • Solve pricing problems using MILP by default • Part of JuMP
VRPSolver	• Specific to VRP variants and related problems • VRP-oriented modeling • Solve pricing problems using customized dynamic programming algorithms
Our work (SCIP + DIDP)	• Explicitly formulate the master problem as MILP and the pricing problem as a dynamic programming model, both solved by general-purpose solvers