Column Generation with Domain-Independent Dynamic Programming

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

Edward Lam
Department of Data Science and Artificial Intelligence, Monash University, Melbourne, Australia

Abstract

Column generation and branch-and-price (B&P) are leading mathematical optimization methods for large-scale exact optimization, iterating between solving a master problem and a pricing problem. Due to the difficulty of discrete optimization, high-performance column generation often relies on a custom pricing algorithm built specifically to exploit the problem's structure. This bespoke nature of the pricing solver makes column generation a problem-specific method and hinders the use of generic implementations across a wide range of problems. We show that domain-independent dynamic programming (DIDP), a model-based paradigm for dynamic programming, can be used as a generic pricing solver. We develop new modeling features and a solving algorithm for DIDP to achieve better performance in typical pricing problems. We demonstrate that in four problem classes, our implementations of B&P, with pricing by DIDP, empirically outperform an existing automated B&P solver and B&P with pricing by mixed-integer programming or constraint programming.

2012 ACM Subject Classification Computing methodologies → Modeling methodologies; Theory of computation → Dynamic programming; Theory of computation → Mathematical optimization

Keywords and phrases Modelling & Modelling Languages,Dynamic Programming,Operations Research & Mathematical Optimisation

Digital Object Identifier 10.4230/LIPIcs.CP.2026.30

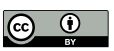
Supplementary Material *Software*: `Code used for experiments`

Funding *Ryo Kuroiwa*: JSPS KAKENHI grant number JP25K24378
Edward Lam: Australian Research Council, grant DE240100042

1 Introduction

Column generation is a method for solving a linear program by looping between solving a restricted master problem with a subset of decision variables and a pricing problem to generate variables that potentially lead to a better solution, rather than enumerating all variables in advance. Branch-and-price (B&P), the combination of branch-and-bound and column generation to solve mixed-integer programs (MIPs), has achieved state-of-the-art performance in solving large-scale combinatorial optimization problems, such as routing and scheduling [21]. Highly efficient problem-specific algorithms to solve the pricing problems have been developed, typically using dynamic programming (DP) [24, 20, 27]. However, generalizing such algorithms to new problems is not necessarily easy due to their problem-specific nature and requires high implementation effort. While generic column generation approaches have been investigated [8, 11], they solve pricing problems using constraint-based approaches such as MIP and constraint programming (CP), rather than DP.

In this paper, we investigate the use of domain-independent dynamic programming (DIDP), a recently proposed model-based paradigm based on DP [17], to solve the pricing problems in column generation. By using DIDP, instead of designing and implementing



problem-specific pricing algorithms, we formulate a pricing problem as a declarative DP model and apply general-purpose solvers. We aim to move column generation closer towards a declarative model-based solving paradigm while enjoying the strengths of DP.

To achieve better performance in typical pricing problems, we develop three new modeling features in a software framework for DIDP: higher-order expressions such as filter and reduce operations, set resource variables, and the fractional knapsack expression. We also develop a new solving algorithm for DIDP inspired by successful DP pricing algorithms. As case studies, we formulate declarative column generation models and implement B&P built on them in four routing and scheduling problems, using the new modeling features. The experimental results show that the new features and algorithm contribute to better performance in multiple problem classes. Our approach empirically outperforms generic B&P baselines: an existing automated B&P solver and B&P approaches using MIP or CP for pricing.

2 Related Work

High-performance column generation often depends on specialized pricing solvers that can exploit specific substructures, limiting truly generic column generation solvers. We briefly review the main automated approaches. GCG is an open-source MIP solver that reformulates a given MIP model so that column generation can be applied [8]. Unless a problem-specific pricing algorithm is implemented by a user, GCG solves both the master and pricing problems using SCIP [1], an open-source MIP solver. Similarly, Coluna,[1] a decomposition framework for mathematical optimization in Julia, also provides automatic column generation and uses a MIP solver for pricing by default. VRPSolver is a non-commercial but proprietary B&P code for solving vehicle routing problems [23], with which users can model certain problem variants within the library's limitations. In the CP community, previous work investigated using CP to solve pricing problems, achieving success in problems such as staff scheduling, packing, and sports scheduling [11].

Many pricing problems can be reduced to a shortest path problem with resource constraints (SPPRC) [14]. Salani, Basso, and Giuffrida [28] proposed PathWyse, a solver library for SPPRC, but its modeling interface is designed to handle only typical SPPRC constraints, such as capacities and time windows, and C++ customization is required for more complex variants. The Boost Graph Library,[2] a C++ library for graph algorithms and data structures, also implements a function to solve SPPRC. However, it is less declarative than model-based paradigms such as MIP, CP, and DIDP because a problem is defined by implementing a set of functions in C++.

3 Background

We formally introduce column generation and domain-independent dynamic programming.

3.1 Column Generation

Many combinatorial optimization problems can be posed as a set partitioning or set covering formulation that selects a set of combinatorial objects (e.g., a path, schedule, cutting pattern) from a large universe. MIP formulations of these problems often associate each object with a

[1] `https://github.com/atoptima/Coluna.jl`
[2] `https://www.boost.org/doc/libs/1_91_0/libs/graph/doc/index.html`

variable/column. The linear relaxation contains an exponential number of columns, so it cannot be constructed explicitly. Instead, column generation solves a sequence of smaller linear relaxations over a subset of columns, and dynamically generates new columns [21].

Let X be the set of all columns, with $n = |X|$. Define the *master problem (MP)* as $\min\{c^\top \lambda : A\lambda \geq b, \lambda \geq 0\}$, where $\lambda = (\lambda_1, \ldots, \lambda_n)$, $c \in \mathbb{Q}^n$, $A \in \mathbb{Q}^{m \times n}$, and $b \in \mathbb{Q}^m$. Column generation maintains a subset $X' \subseteq X$ with $|X'| = n' \leq n$ and solves the corresponding *restricted master problem (RMP)* $\min\{c'^\top \lambda' : A'\lambda' \geq b, \lambda' \geq 0\}$, where $c' \in \mathbb{Q}^{n'}$ and $A' \in \mathbb{Q}^{m \times n'}$ are the corresponding submatrices. Let $\hat{\pi} \in \mathbb{R}^m_+$ be an optimal dual solution to the RMP. For any column $j \in X$, define its reduced cost as $\bar{c}_j = c_j - A_{\cdot,j}^\top \hat{\pi}$. Any column $j \in X \setminus X'$ with $\bar{c}_j < 0$ may be needed in a lower-cost solution to the RMP, and therefore should be added to X'.

Since the full set of columns is too large to enumerate, columns with negative reduced cost are found by solving a *pricing problem*. Assume that each column is represented by a vector $x \in \mathbb{Z}^k$ satisfying internal constraints $Dx \geq e$, with cost $c(x)$ and column coefficients $a(x)$. Then, the pricing problem is $\min\{c(x) - a(x)^\top \hat{\pi} : Dx \geq e, x \in \mathbb{Z}^k\}$. If the minimum is negative, the corresponding column is added to X', and the process repeats. Otherwise, the current optimal RMP solution is also optimal for the MP. Branch-and-price (B&P) embeds column generation inside branch-and-bound to enforce integrality constraints on MP.

3.2 Dynamic Programming

Combinatorial problems can be defined in DP by states and transitions, with costs or profits on transitions. As a running example, consider SPPRC instantiated as a pricing problem for the vehicle routing problem with time windows (VRPTW) [31] with capacity and time-window resources and an elementary (no-revisit) constraint. Let $(\mathcal{N}, \mathcal{A})$ be a directed graph with nodes $\mathcal{N} = \{0, \ldots, n+1\}$ (source 0, sink $n+1$) and arcs $\mathcal{A} \subseteq \{\mathcal{N} \times \mathcal{N} : i \neq j, i < n+1, j > 0\}$. Each customer i has load $l_i > 0$, release time $a_i \geq 0$, deadline $b_i \geq 0$, and service duration $s_i \geq 0$. Each arc (i, j) has distance $d_{i,j} \geq 0$ and travel cost $c_{i,j}$. The goal is an elementary path from 0 to $n+1$ of minimum total travel cost such that cumulative load never exceeds capacity Q and the visit to each node i occurs within $[a_i, b_i]$.

Let $V(\mathcal{R}, i, q, t)$ be the minimum cost from node i to $n+1$ when the unvisited set is $\mathcal{R} \subseteq \{1, \ldots, n\}$, current load is q, and time is t. We may visit node j only if $j \in \mathcal{R} \cup \{n+1\}$, $(i, j) \in \mathcal{A}$, $q + l_j \leq Q$, and $t + s_i + d_{i,j} \leq b_j$. Once visiting j, we consider a subproblem of computing the minimum cost from node j to $n+1$, where j is removed from the unvisited set, the load is increased by l_j, and the time is updated to the later of the arrival time $t + s_i + d_{i,j}$ and the beginning of the time window a_j at j. As a result, we compute $V(\mathcal{R} \setminus \{j\}, j, q + l_j, \max\{t + s_i + d_{i,j}, a_j\})$. Using the set of nodes that can be visited next, $\mathrm{Next}(\mathcal{R}, i, q, t) = \{j \in \mathcal{R} \cup \{n+1\} : (i, j) \in \mathcal{A} \wedge q + l_j \leq Q \wedge t + s_i + d_{i,j} \leq b_j\}$, V is defined by the following Bellman equation:

$$V(\mathcal{R}, i, q, t) = \begin{cases} \min\limits_{j \in \mathrm{Next}(\mathcal{R}, i, q, t)} c_{i,j} + V\left(\mathcal{R} \setminus \{j\}, j, q + l_j, \max\left\{t + s_i + d_{i,j}, a_j\right\}\right) & \text{if } i \leq n \\ 0 & \text{else.} \end{cases} \tag{1}$$

The optimal objective value is $V(\{1, \cdots, n\}, 0, 0, 0)$. In column generation, Bellman equations are typically solved using problem-specific algorithms [24, 20, 27].

3.3 Domain-Independent Dynamic Programming

Domain-independent dynamic programming (DIDP) is a model-based DP paradigm for combinatorial optimization. Previous work developed Dynamic Programming Description Language (DyPDL) [17], a declarative modeling formalism for DIDP. Currently, DyPDL is limited to DP formulations where a solution can be represented as a sequence of decisions, and a DP model is defined by *state variables*, *transitions*, *base cases*, and *state constraints*.

A state variable has a type, either numeric, element, or set. A *numeric variable* takes a value in $\mathbb{Q}$, an *element variable* in $\mathbb{Z}_0^+$, and a *set variable* in $2^{\mathbb{Z}_0^+}$. A state is represented by full value assignments $(d_1, \dots, d_n)$ to the state variables $(x_1, \dots, x_n)$ such that d_i is in the domain of x_i, and we denote the value of a state variable x_i in state S by $S[x_i]$. For our example in Equation (1), $\mathcal{R}$, i, q, and t are modeled as state variables, where $\mathcal{R}$ is a set variable, i is an element variable, and q and t are numeric variables.

Expressions are functions of a state, where *numeric expressions* return a numeric value, *element expressions* return a nonnegative integer, *set expressions* return a set of nonnegative integers, and *conditions* return a Boolean value ($\bot$ or $\top$). Given a condition c, we denote $S \models c$ if $c(S) = \top$, i.e., S satisfies c, and $S \not\models c$ if $c(S) = \bot$. In practice, expressions are built from a predefined set of operations such as arithmetic and set-theoretic operations.

A transition defines *effects* on the state variables and *preconditions* to apply, using expressions. In our example, we define the first line of Equation (1) by using a transition corresponding to visiting each customer j, whose effect on the set variable $\mathcal{R}$ is $\mathcal{R} \setminus \{j\}$ (a set expression), effect on the element variable i is j (an element expression), effect on the numeric variable q is $q + l_j$ (a numeric expression), and effect on the numeric variable t is $\max\{t + s_i + d_{i,j}, a_j\}$ (a numeric expression). By $S[\![\tau]\!]$, we denote the *successor state* of state S, where the values of state variables are updated by the effects of transition τ. To visit customer j, it must satisfy $j \in \mathrm{Next}(\mathcal{R}, i, q, t)$, i.e., conditions $j \in \mathcal{R} \cup \{n+1\}$, $(i, j) \in \mathcal{A}$, $q + l_j \leq Q$, and $t + s_i + d_{i,j} \leq b_j$ are preconditions. By $\mathcal{T}(S)$, we denote the set of *applicable* transitions whose preconditions are satisfied by S.

A base case consists of a set of conditions when no further transition is applicable. A state satisfying a base case is called a *base state*. In our example, we define the second line of Equation (1) using a base case, where the set of conditions is $\{i = n+1\}$. An *S-solution* is a sequence of transitions $\sigma = \langle \sigma_1, ..., \sigma_m \rangle$ inducing a sequence of states $\langle S^0, ..., S^m \rangle$ such that $S^0 = S$, S^m is a base state, and $S^i = S^{i-1}[\![\sigma_i]\!]$ with $\sigma_i \in \mathcal{T}(S^{i-1})$ for $i = 1, \dots, m$. We assume the additive cost structure: the cost of the S-solution σ is $\sum_{i=1}^{m} w_{\sigma_i}(S^{i-1}) + w_{\mathrm{base}}(S^m)$, where w_τ is a numeric expression associated with transition τ, and w_{base} is a numeric expression for the base cases. In our example, $w_\tau(\mathcal{R}, i, q, t) = c_{i,j}$ for a transition τ to visit location j, and $w_{\mathrm{base}}(\mathcal{R}, i, q, t) = 0$. Assuming minimization, by $V(S)$, we denote the minimum S-solution cost. A special state called the *target state* S^I is defined for a DyPDL model, and the optimal solution for a DyPDL model is the minimum cost S^I-solution. In our example, $S^I = (\mathcal{R} = \mathcal{N}, i = 0, q = 0, t = 0)$. In addition to the above components, *state constraints* are conditions that must be satisfied by all states, but they do not appear in our example.

3.3.1 Redundant Information

In DyPDL, redundant information implied by other parts of the model can be explicitly defined, which can potentially be useful for a solver. DyPDL provides two specific features solely for redundant information: *resource variables* and *dual bound functions*.

In problem-specific DP algorithms, state dominance is sometimes exploited, where one state is known to be superior to another. For our example in Equation (1), a state $(\mathcal{R}, i, q_1, t_1)$

leads to a better or equal solution than a state $(\mathcal{R}, i, q_2, t_2)$ with the same $\mathcal{R}$ and i if $q_1 \leq q_2$ and $t_1 \leq t_2$. This state dominance can be written as the following inequality:

$$V(\mathcal{R}, i, q_1, t_1) \leq V(\mathcal{R}, i, q_2, t_2) \text{ if } q_1 \leq q_2 \wedge t_1 \leq t_2. \tag{2}$$

In DyPDL, to represent state dominance, a numeric or element variable can be declared as a resource variable with a preference for less or greater. A state S is preferred over another state S' if $S[r] \leq S'[r]$ for each resource variable r that prefers less, $S[r] \geq S'[r]$ for each resource variable r that prefers greater, and $S[x] = S'[x]$ for each non-resource variable x. In our example, q and t are resource variables where less is preferred.

A dual bound function η returns a lower bound $\eta(S)$ on the minimum S-solution cost $V(S)$ and an upper bound for maximization, similar to a dual bound in mathematical optimization. In DyPDL, a dual bound function is described by an expression, similar to other components. For our example in Equation (1), since the travel cost of an arc can be negative, we can use a dual bound function that only considers the most negative incoming arc for each node $j = 1, \dots, n$. We can underestimate the incoming arc cost by $c_j^{\text{in}} = \min_{(k,j) \in \mathcal{A}} c_{k,j}$ for node j. We have the following dual bound function:

$$V(\mathcal{R}, i, q, t) \geq \begin{cases} 0 & \text{if } i = n+1 \\ \sum_{j \in \mathcal{R}} \min\left\{c_j^{\text{in}}, 0\right\} + c_{n+1}^{\text{in}} & \text{else.} \end{cases} \tag{3}$$

Similarly, we can use another dual bound function using $c_j^{\text{out}} = \min_{(j,k) \in \mathcal{A}} c_{j,k}$ instead of c_j^{in} for $j \in \mathcal{R}$ and c_i^{out} instead of c_{n+1}^{in}.

3.3.2 Solvers

Once a DP model is formulated, it is solved by general-purpose solvers. Previous work [17] developed solvers based on heuristic state-space search algorithms [26], such as CAASDy, which uses A* [13], and complete anytime beam search (CABS) [32]. In these solvers, a DP model is solved by finding a shortest path from the target state to a base state in a state space graph, where the nodes are states, and an edge $(S, S[\![\tau]\!])$ exists if $\tau \in \mathcal{T}(S)$. Each edge $(S, S[\![\tau]\!])$ is labeled with transition τ and having the weight $w_\tau(S)$.

4 New Modeling Features in DIDP

We extend didp-rs,[3] a software implementation of DyPDL, to efficiently model and solve problems commonly seen in pricing. We add three new features: higher-order expressions, set resource variables, and fractional knapsack expressions.

4.1 Higher-Order Expressions

For the SPPRC example in Equation (1), a state is represented by a set of unvisited nodes $\mathcal{R}$, the current node i, the current load q, and the current time t. While we update $\mathcal{R}$ to $\mathcal{R} \setminus \{j\}$ when j is visited, we can also remove a node $k \in \mathcal{R}$ that can no longer be visited by its deadline b_k. Let $d_{j,k}^*$ be the shortest travel time from node j to node k, which can be precomputed. If $\max\{t + s_i + d_{i,j}, a_j\} + s_j + d_{j,k}^* > b_k$, then node k cannot be visited after visiting j from the current state. In addition, k cannot be visited after j if it results in

[3] https://github.com/domain-independent-dp/didp-rs

overload, i.e., $q + l_j + l_k > Q$. Thus, the update on $\mathcal{R}$ can be represented by an expression $\mathcal{R}'(j) = \left\{k \in \mathcal{R} \setminus \{j\} : \max\{t + s_i + d_{i,j}, a_j\} + s_j + d^*_{j,k} \leq b_k \wedge q + l_j + l_k \leq Q\right\}$.

In didp-rs, expressions are built from a library of operations on state variables. The solvers maintain expression tree data structures and evaluate them during solving. For set expressions, set operations such as union, intersection, and difference are implemented, but $\mathcal{R}'(j)$ cannot be directly represented with them.

We introduce higher-order expressions, which apply an expression to each element of a given set. In particular, a *filter operation* is a set expression that returns a subset of a given set whose elements satisfy a given condition. With our interface, a user specifies a filter operation by two components: a set expression $\mathcal{X}$ and a parameterized condition c, which is a function that returns a condition $c(x)$ given a parameter x. The parameter x is a placeholder and is replaced with each element of a set $\mathcal{X}(S)$ when evaluated, given a state S, and an element $i \in \mathcal{X}(S)$ is removed if $S \not\models c(i)$. In other words, the filter operation represents an expression that returns $\{x \in \mathcal{X}(S) : S \models c(x)\}$ given a state S. For our example, $\mathcal{R}'(j)$ can be represented by a filter operation defined by a set expression $\mathcal{R} \setminus \{j\}$ and a parameterized condition $\max\{t + s_i + d_{i,j}, a_j\} + s_j + d^*_{j,k} \leq b_k \wedge q + l_j + l_k \leq Q$, where k is the parameter. We present example code using the filter operation in DIDPPy, the Python interface of didp-rs, in Listing 1. Here, we define the placeholder using `model.add_local_var` and give it to a filter expression (`.filter`) as the first argument, where `r` is a set variable.

Code Listing 1 DIDPPy example of the filter operation.

```
q_next = q + l[j]
t_next = dp.max(t + s[i] + d[i,j], a[j])
k = model.add_local_var()
r_next = r.remove(j).filter(
    k, (t_next + s[j] + d_star[j,k] <= b[k]) & (q_next + l[k] <= Q)
)
```

We also introduce the *reduce sum operation* returning $\sum_{x \in \mathcal{X}(S)} e(x)(S)$ given a state S, where $\mathcal{X}$ is a set expression, x is a placeholder, and e is a parameterized numeric expression. We show its use case later in Section 6.2.

4.2 Set Resource Variables

In the current DyPDL, only numeric and element variables can be resource variables to define state dominance. However, in pricing problems, state dominance is sometimes defined by a set variable. For the SPPRC example in Equation (1), we can define state dominance where state $(\mathcal{R}_1, i, q_1, t_1)$ is better than or as good as $(\mathcal{R}_2, i, q_2, t_2)$ with the same i if $\mathcal{R}_2 \subseteq \mathcal{R}_1$, $q_1 \leq q_2$, and $t_1 \leq t_2$ since having more candidates to visit potentially leads to a shorter path.

We introduce *set resource variables*: a state S is preferred to another state S' only if the value of a set resource variable in S is a subset or superset of that in S'. Similar to numeric and element resource variables, the preference (less or greater) specifies whether a subset or superset is better. When less/greater is specified for a set resource variable $\mathcal{X}$, S is preferred to S' only if $S[\mathcal{X}] \subseteq S'[\mathcal{X}]/S'[\mathcal{X}] \subseteq S[\mathcal{X}]$. We present example code in Listing 2, where `node` is an object type to define the maximum cardinality of the set variable, `target` specifies the value in the target state, and `less_is_better=False` specifies that greater is preferred.

Code Listing 2 DIDPPy example of a set resource variable

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

4.3 Fractional Knapsack Expression

For our example in Equation (1), we presented in Example 3 a dual bound function based on the minimum incoming arc cost $c_j^{\text{in}} = \min_{(k,j)\in\mathcal{A}} c_{k,j}$ for each node j. We can also define a dual bound based on the current load q and the capacity Q. By visiting node j, we increase the load by l_j and the cost by at least c_j^{in}. Given a state $(\mathcal{R}, i, q, t)$,

$$V(\mathcal{R}, i, q, t) \geq \min_{\mathcal{J}\subseteq\mathcal{R}: q+\sum_{j\in\mathcal{J}} l_j \leq Q} \sum_{j\in\mathcal{J}} \min\left\{c_j^{\text{in}}, 0\right\} + c_{n+1}^{\text{in}} \text{ if } i \neq n+1. \tag{4}$$

The first term can be viewed as the negation of the optimal cost of the 0-1 knapsack problem, which is to maximize the total profit of items packed into a knapsack with a fixed capacity. In particular, the knapsack has capacity $Q-q$, and each node $j \in \mathcal{R}$ with $c_j^{\text{in}} < 0$ corresponds to an item with the profit $-c_j^{\text{in}}$ and weight l_j. We argue that a similar substructure is common in pricing problems when a subset of elements with the negative reduced costs needs to be selected under a resource constraint.

Since the 0-1 knapsack problem is NP-hard [15], computing the right-hand side of Inequality (4) is also NP-hard. We use the Dantzig bound [2], a polynomial-time upper bound on the optimal objective value for the 0-1 knapsack problem. Given the capacity C and a set of items $\mathcal{N}$ with weight $w_j > 0$ and the profit $p_j > 0$ for each $j \in \mathcal{N}$, the Dantzig bound can be computed as follows. First, the items are sorted in a descending order of $\frac{p_j}{w_j}$. Second, the items are included in the knapsack in sorted order as long as the total weight does not exceed the capacity C, and let $\mathcal{I}$ be the set of such items. When the current item j has the weight w_j larger than $C - \sum_{i\in\mathcal{I}} w_i$, it is fractionally included with the profit $\frac{p_j}{w_j}\left(C - \sum_{i\in\mathcal{I}} w_i\right)$, i.e., the objective value is upper bounded by $\frac{p_j}{w_j}\left(C - \sum_{i\in\mathcal{I}} w_i\right) + \sum_{i\in\mathcal{I}} p_i$.

The Dantzig bound was used with RPID, another DIDP software where a model is defined by Rust functions [16]. However, with expressions in didp-rs, efficiently modeling the Dantzig bound is difficult due to its algorithmic nature. We introduce a new expression, called the fractional knapsack expression, denoted by $\mathsf{fractional\_knapsack}\left(\mathcal{X}, C, (p_j)_{j=1,\dots,n}, (w_j)_{j=1,\dots,n}\right)$, where $\mathcal{X}$ is a set expression, C is a numeric expression, and $(p_j)_{j=1,\dots,n}$ and $(w_j)_{j=1,\dots,n}$ are lists of n numeric expressions. Then, the expression represents the Dantzig bound for the 0-1 knapsack problem, where given a state S, the set of items is $\mathcal{X}(S)$, the capacity of the knapsack is $C(S)$, and each item $x \in \mathcal{X}(S)$ has the profit p_x and the weight w_x. For our example, when $i \neq n+1$, we represent the dual bound function as follows:

$$V(\mathcal{R}, i, q, t) \geq -\mathsf{fractional\_knapsack}\left(\mathcal{R}, Q-q, \left(\max\left\{-c_j^{\text{in}}, 0\right\}\right)_{j=1,\dots,n}, \left(l_j\right)_{j=1,\dots,n}\right) + c_{n+1}^{\text{in}} \tag{5}$$

where $c_j^{\text{in}} = \min_{(k,j)\in\mathcal{A}} c_{k,j}$ is the minimum incoming travel cost to node j. Zero-weight items are ignored by the expression. In our example code (Listing 3), (i == n+1).if_then_else(..., ...) is an expression that evaluates to the first argument (0) if the current location i equals n + 1 (the sink node) and to the second argument (the fractional knapsack bound) if not.

We can use three additional dual bound functions similar to Inequality (5): the first one uses the capacity $b_{n+1} - t - s_i$ and the weight $\min_{(k,j)\in\mathcal{A}} d_{k,j} + s_j$; the second one uses the

Code Listing 3 DIDPPy example of a fractional knapsack expression

```
model.add_dual_bound(
    (i == n + 1).if_then_else(
        0,
        -dp.fractional_knapsack(
            r, Q - q, [max(-cin[j], 0) for j in range(1, n + 1)], l
        )
    )
)
```

profit $\max\left\{-c_j^{\text{out}}, 0\right\}$ with $c_j^{\text{out}} = \min_{(j,k)\in\mathcal{A}} c_{j,k}$ instead of c_j^{in}; the third one uses the profit $\max\left\{-c_j^{\text{out}}, 0\right\}$, the capacity $b_{n+1} - t$, and the weight $\min_{(j,k)\in\mathcal{A}} s_j + d_{j,k}$.

5 Generic Labeling Solver for DIDP

Our solver is built on top of the anytime heuristic search framework of DIDP proposed in previous work [17], which solves a DP model by finding a shortest path from the target state to a base state in a state space graph. In that framework, states to be searched are maintained in a priority queue called an *open list*. In each iteration, one state is selected and removed from the open list. Then, it is *expanded*, i.e., its successor states are generated by applying transitions and then inserted into the open list if they are not dominated by existing states. Each concrete algorithm differs in how it selects states to expand from the open list. In existing DIDP solvers, including CAASDy and CABS, the state is selected based on its f-value, which is the sum of its g-value, the path cost to reach the state from the target state, and its h-value, the estimated path cost from the state to a base state by a heuristic function h. The existing solvers use the dual bound function η defined in the model as the heuristic function. When multiple dual bound functions are defined, the maximum value is used for each state.

Unlike the existing solvers in DIDP, our new solver is inspired by problem-specific labeling algorithms for solving the SPPRC [14, 24] and selects states based on resource variables. We explain the motivation of our labeling solver using the example SPPRC for VRPTW. Suppose two states $S_1 = (\mathcal{R}_1, i, q_1, t_1)$ with g-value $g(S_1)$ and $S_2 = (\mathcal{R}_2, i, q_2, t_2)$ with g-value $g(S_2)$ such that $\mathcal{R}_2 \subseteq \mathcal{R}_1$, $q_2 \geq q_1$, $t_2 \geq t_1$, and $g(S_2) \geq g(S_1)$, i.e., S_1 dominates S_2 with a better or equal g-value. Since extending the path from the target state to S_1 results in a better or equal solution than that of S_2, we do not need to search S_2 once expanding S_1. In other words, expanding S_2 before S_1 should be avoided, and expanding the state with more preferred resource variable values potentially reduces search effort. With this motivation, our algorithm expands a state based on a lexicographic order of resource variables.

We present pseudocode in Algorithm 1. Except for line 4, the algorithm and implementation details mirror the existing solvers in DIDP. When the target state violates state constraints, we immediately return an empty set and terminate (line 1). We initialize the set of solutions found Σ with an empty set (line 2). We also maintain the current best solution cost $\overline{\gamma}$, initialized with ∞ (line 2). For each state S, we record the best sequence of transitions to reach it, $\sigma(S)$, and the g-value $g(S)$, corresponding to the accumulated path cost. Given $\sigma(S) = \langle\sigma_1, \ldots, \sigma_m\rangle$, we have $g(S) = \sum_{i=1}^{m} w_{\sigma_i}(S^{i-1})$ where $S^i = S^{i-1}[\![\sigma_i]\!]$ for $i = 1, \ldots, m$ and $S^0 = S^I$. For the target state S^I, the path is empty with $g(S^I) = 0$. The set G stores all generated states to check state dominance with a newly generated state, and

Algorithm 1 Generic labeling solver for a DyPDL model. The target state is denoted by S^I and the dual bound function by η.

1: **if** S^I does not satisfy the state constraints **then return** $\emptyset$
2: $\Sigma \leftarrow \emptyset$, $\overline{\gamma} \leftarrow \infty$, $\sigma(S^I) \leftarrow \langle\rangle$, $g(S^I) \leftarrow 0$, $O \leftarrow \{S^I\}, G \leftarrow \{S^I\}$ ▷ Initialization.
3: **while** $O \neq \emptyset$ **do**
4: Let $S \in O$ be the lexicographically minimum state
5: $O \leftarrow O \setminus \{S\}$ ▷ Pop the chosen state.
6: **if** S is a base state and $g(S) + w_{\text{base}}(S) < \overline{\gamma}$ **then**
7: $\Sigma \leftarrow \Sigma \cup \{\sigma(S)\}$, $\overline{\gamma} \leftarrow g(S) + w_{\text{base}}(S)$ ▷ Update the solutions.
8: $O \leftarrow \{S' \in O : g(S') + \eta(S') < \overline{\gamma}\}$ ▷ Prune states in the open list.
9: **else**
10: **for all** $\tau \in \mathcal{T}(S) : S[\![\tau]\!]$ satisfies all state constraints **do**
11: $g_{\text{current}} \leftarrow g(S) + w_\tau(S)$ ▷ Compute the g-value.
12: **if** no state $S' \in G$ is preferred to $S[\![\tau]\!]$ with $g(S') \leq g_{\text{current}}$ **then**
13: $G \leftarrow \{S' \in G : S[\![\tau]\!]$ is not preferred to $S' \vee g(S') < g_{\text{current}}\}$
14: $O \leftarrow \{S' \in O : S[\![\tau]\!]$ is not preferred to $S' \vee g(S') < g_{\text{current}}\}$
15: **if** $g_{\text{current}} + \eta(S[\![\tau]\!]) < \overline{\gamma}$ **then**
16: $\sigma(S[\![\tau]\!]) \leftarrow \langle \sigma(S); \tau \rangle$, $g(S[\![\tau]\!]) \leftarrow g_{\text{current}}$
17: $G \leftarrow G \cup \{S[\![\tau]\!]\}$, $O \leftarrow O \cup \{S[\![\tau]\!]\}$ ▷ Insert the successor state.
18: **return** Σ ▷ Return solutions.

the open list O stores states to be searched, both of which initially contain only the target state. The algorithm proves optimality (or infeasibility) when the open list becomes empty (line 3) and returns the set of solutions found.

In each step, the lexicographical-minimum state S is removed from the open list (lines 4 and 5). States are lexicographically ordered based on the values of resource variables. Given resource variables $r_1, \ldots, r_{n'}$, a state S is lexicographically smaller than S' if there exists $1 \leq i \leq n'$ such that $S[r_j] = S'[r_j]$ for $1 \leq j < i$, $S[r_i] \neq S'[r_i]$, and $S[r_i]$ is preferred to $S'[r_i]$. In our implementation, we compare element resource variables, numeric resource variables, and set resource variables in order. Resource variables of the same type are compared in order of definition. When all resource variables have the same values, we break ties by the f-value ($f(S) = g(S) + \eta(S)$), and then the η-value, where smaller is preferred. When no resource variables are used, our algorithm is the same as CAASDy, which uses A*.

If S is a base state, then $\sigma(S)$ is a solution, and the best solution cost $\overline{\gamma}$ is updated if $\sigma(S)$ is better (lines 6–7). In addition, all states $S' \in O$ with $g(S') + \eta(S') \geq \overline{\gamma}$ are removed from the open list since they cannot lead to a better solution (line 8).

If S is not a base state, its successor state $S[\![\tau]\!]$ is generated for every applicable transition in $\tau \in \mathcal{T}(S)$ if it satisfies the state constraints (line 10). If $S[\![\tau]\!]$ is dominated by another state S' in G with a better or equal g-value, it cannot lead to a solution better than S', so $S[\![\tau]\!]$ is ignored (line 12). Otherwise, states dominated by $S[\![\tau]\!]$ with a better or equal g-value are removed from G (line 13). For this procedure, G is implemented as a hash table, where keys are the values of the non-resource variables, and entries are arrays of pointers to states. When a successor state is generated, an array of states with the same non-resource variable values is retrieved from the hash table. The successor state is compared against each state in the array to detect dominance and appended to the array if not dominated.

After dominance detection, the dual bound value $\eta(S[\![\tau]\!])$ is computed. If $g(S) + w_\tau(S) + \eta(S[\![\tau]\!])$ is worse than the best solution cost, the successor state $S[\![\tau]\!]$ is ignored (line 15).

Otherwise, $\sigma(S[\![\tau]\!])$ and $g(S[\![\tau]\!])$ are initialized or updated, and $S[\![\tau]\!]$ is inserted into the open list and G (line 17). Here, $\langle \sigma(S); \tau \rangle$ is a sequence of transitions extending $\sigma(S)$ with τ.

6 Case Studies

We present DIDP models for pricing across two scheduling problems.

6.1 Parallel Machine Scheduling

We consider the parallel machine scheduling problem commonly denoted as $P||\sum w_j C_j$ [4]. In this problem, a set of n non-preemptive jobs $\mathcal{J}$ are scheduled on m identical machines, where each job $j \in \mathcal{J}$ has processing time p_j and weight w_j. Given a schedule, let C_j be the completion time of job j. The objective is to minimize the total weighted completion time $\sum_{j \in \mathcal{J}} w_j C_j$. Previous work studied B&P for this problem [30], which used DP to solve the pricing problem. However, unlike the original DP algorithm that does not use the dual bound function, our DP model uses a fractional knapsack expression as the dual bound function.

Problem (6) shows the MP. Let $\mathcal{S}$ be the set of schedules for a single machine, c_s be the cost of the schedule, $a_{s,j}$ equal to 1 if schedule s contains job j and 0 otherwise, and λ_s be a binary variable representing whether schedule $s \in \mathcal{S}$ is used. Objective (6a) minimizes the total cost, Constraint (6b) ensures that at most m schedules are used, and Constraint (6c) ensures that each job $j \in \mathcal{J}$ is scheduled exactly once.

$$\min \sum_{s \in \mathcal{S}} c_s \lambda_s \tag{6a}$$

$$\sum_{s \in \mathcal{S}} \lambda_s \leq m \tag{6b}$$

$$\sum_{s \in \mathcal{S}} a_{s,j} \lambda_s = 1 \qquad \forall j \in \mathcal{J} \tag{6c}$$

$$\lambda_s \in \mathbb{Z}_+ \qquad \forall s \in \mathcal{S}. \tag{6d}$$

The pricing problem finds a schedule that minimizes the reduced cost, computed from the dual value π_j for each job j defined by Constraint (6c) and the dual value defined by Constraint (6b). Without loss of generality, we assume that jobs are ordered so that $w_j/p_j \geq w_{j+1}/p_{j+1}$ for $j = 1, ..., n-1$, and job j is scheduled earlier than job $k > j$ on the same machine [5]. In addition, we can derive the release date (the minimum start time) a_j and the deadline (the maximum completion time) b_j of job j based on a theoretical analysis [30]. Therefore, we decide whether to include job j in the schedule, starting from $j = 0$ and ensuring that the starting time is in time window $[a_j, b_j]$.

In our DP formulation shown in Problem (7), an element variable j represents the job currently considered, and a numeric variable t represents the current time. The first line of Equation (7a) is a base case, where all jobs have been considered. The second line considers a situation where j can be scheduled satisfying the time window constraint. It takes the minimum of the two cases: one where j is scheduled and the other where j is not scheduled. While j is updated to $j+1$ in any case, the cost is increased by $-\pi_j + w_j(t + p_j)$ and t is increased by p_j if j is scheduled. The third line ignores job j as it cannot be scheduled within the time window. The reduced cost is computed by subtracting the dual value for Constraint (6b) from $V(j = 1, t = 0)$.

An upper bound on the completion time of a schedule is given by $H = \sum_{k \in \mathcal{J}} p_k/m + (m-1) \max_{k \in \mathcal{J}} p_k/m$. Therefore, the sum of processing time scheduled from state (j, t)

must be less than or equal to $H - t$. The completion time of each job is at least $a_k + p_k$. Given these values, in Inequality (7b), we consider a dual bound function based on the 0-1 knapsack problem with the set of items $\{j, ..., n\}$ and the capacity $H - t$, where each job k has the profit $v_k = \max\{\pi_k - w_k(a_k + p_k), 0\}$ and the weight p_k.

$$V(j,t) = \begin{cases} 0 & \text{if } j = n+1 \\ \min\{-\pi_j + w_j(t+p_j) + V(j+1, t+p_j), V(j+1,t)\} & \text{if } a_j \le t \le b_j - p_j \\ V(j+1,t) & \text{else} \end{cases} \tag{7a}$$

$$V(j,t) \ge -\mathsf{fractional\_knapsack}\left(\{k \in \mathcal{J} : k \ge j\}, H - t, (v_k)_{k \ge j}, (p_k)_{k \ge j}\right). \tag{7b}$$

Following the previous work [30], we change the time window of one job at each branching.

6.2 Multi-runway Aircraft Scheduling

The multi-runway aircraft scheduling problem (MRASP), described by [10], schedules a set of heterogeneous aircraft on a set of identical runways while respecting minimum separation times between aircraft and minimizing a weighted sum of scheduled times. Let $\mathcal{M} = \{1, \dots, m\}$ be the set of m identical runways and let $\mathcal{O} = \{\text{takeoff}, \text{landing}\}$ be the set of operations. Let $\mathcal{G}$ be the set of aircraft classes and $\mathcal{N} = \{1, \dots, n\}$ be the set of aircraft. Every aircraft $i \in \mathcal{N}$ is associated with a class $g_i \in \mathcal{G}$, an operation $o_i \in \mathcal{O}$, a release time $a_i \ge 0$, a deadline $b_i \ge a_i$ and a cost weight $w_i > 0$. Every tuple $(g_i, o_i, g_j, o_j) \in \mathcal{G} \times \mathcal{O} \times \mathcal{G} \times \mathcal{O}$ is associated with a minimum separation time $d_{g_i,o_i,g_j,o_j} \ge 0$. An aircraft $j \in \mathcal{N}$ scheduled sometime after $i \in \mathcal{N}$, $i \ne j$, on the same runway must occur at least d_{g_i,o_i,g_j,o_j} later. In general, the triangle inequality does not hold in d. Therefore, it is insufficient to check the minimum separation time of every aircraft and its immediate successor. Instead, the minimum separation time of every later aircraft must be checked for every aircraft.

Previous work proposed B&P for MRASP [10]. As MRASP can be viewed as a variant of parallel machine scheduling, where runways are machines and aircraft operations are jobs, the MP is the same as Problem (6). The pricing problem is to find a schedule for a single runway satisfying the separation time constraints, for which the previous work used DP as a specialized algorithm but did not provide a declarative DP model.

In our DP model in Problem (8), we use a set variable $\mathcal{R}$ representing the set of aircraft whose operations can be conducted, an element variable i representing the last aircraft landed, and a numeric variable t representing the time to start the operation for i. Each transition schedules an operation for aircraft $j \in \mathcal{R}$, removing j from $\mathcal{R}$ and updating i to j. If the triangle inequality holds for the separation time, we can start the operation for j at time $\max\{t + d_{g_i,o_i,g_j,o_j}, a_j\}$, considering only the separation time between i and j. However, since the triangle inequality does not hold, depending on the aircraft operations scheduled before i, we may need to start the operation for j later. We use additional numeric variables $(e_{g,o})_{(g,o)\in\mathcal{Q}}$ representing the earliest time that class g can perform operation o, where

$$\mathcal{Q} = \{(g,o) \in \mathcal{G} \times \mathcal{O} : \exists g_1 \in \mathcal{G}, o_1 \in \mathcal{O}, g_2 \in \mathcal{G}, o_2 \in \mathcal{O}, d_{g_1,o_1,g_2,o_2} + d_{g_2,o_2,g,o} < d_{g_1,o_1,g,o}\}.$$

In other words, we maintain $e_{g,o}$ for $(g,o) \in \mathcal{Q}$ such that there exist pairs of aircraft classes and operations violating the triangle inequality. Given these variables, t is updated to $t'(j) = \max\{t + d_{g_i,o_i,g_j,o_j}, a_j, e_{g_j,o_j}\}$ if $(g_j, o_j) \in \mathcal{Q}$ and $t'(j) = \max\{t + d_{g_i,o_i,g_j,o_j}, a_j\}$ otherwise. Each $e_{g,o}$ is updated to $e'_{g,o}(j) = \max\{e_{g,o}, t'(j) + d_{g_j,o_j,g,o}\}$. For $\mathcal{R}$, we exclude aircraft k whose operation cannot be executed by the deadline, i.e., $t'(j,k) > b_k$, where $t'(j,k) =$

$\max\{t'(j)+d_{g_j,o_j,g_k,o_k}, e'_{g_k,o_k}(j)\}$ if $(g_k, o_k) \in \mathcal{Q}$ and $t'(j,k) = t'(j)+d_{g_j,o_j,g_k,o_k}$ otherwise. In addition, if the cost of scheduling aircraft k cannot be negative, i.e., $-\pi_k+w_k t'(j,k) > 0$, there is no benefit to consider k. Therefore, $\mathcal{R}$ is updated to $\mathcal{R}'(j) = \{k \in \mathcal{R} \setminus \{j\} : t'(j,k) \leq b'_k\}$, where $b'_k = \min\{b_k, -\pi_k/w_k\}$, implemented by the filter operation.

$$V\left(\mathcal{R}, i, t, (e_{g,o})_{(g,o)\in\mathcal{Q}}\right) = \begin{cases} 0 & \text{if } i = n+1 \\ \min\limits_{j\in\mathcal{R}\cup\{n+1\}:t'(j)\leq b'_j} -\pi_j + w_j t'(j) + V\left(\mathcal{R}'(j), j, t'(j), \left(e'_{g,o}(j)\right)_{(g,o)\in\mathcal{Q}}\right) & \text{if } i \leq n \end{cases} \tag{8a}$$

$$V\left(\mathcal{R}_1, i, t_1, (e^1_{g,o})_{(g,o)\in\mathcal{Q}}\right) \leq V\left(\mathcal{R}_2, i, t_2, (e^2_{g,o})_{(g,o)\in\mathcal{Q}}\right) \quad \text{if } \mathcal{R}_2 \subseteq \mathcal{R}_1 \wedge t_1 \leq t_2 \wedge e^1_{g,o} \leq e^2_{g,o}, \forall (g,o) \in \mathcal{Q} \tag{8b}$$

$$V\left(\mathcal{R}, i, t, (e_{g,o})_{(g,o)\in\mathcal{Q}}\right) \geq \begin{cases} 0 & \text{if } i = n+1 \\ \sum_{j\in\mathcal{R}} \min\{0, -\pi_j + w_j t'(j)\} & \text{else.} \end{cases} \tag{8c}$$

We represent a transition to finish scheduling by introducing a dummy aircraft $n+1$, with $a_j = w_j = \pi_j = 0$, $b_j = \infty$, and $d_{g_i,o_i,g_{n+1},o_{n+1}} = 0$ for any $i \in \mathcal{N}$ and define a base case in the first line of Equation (8a). The minimum reduced cost is computed from $V(\mathcal{R} = \{j \in \mathcal{N} : -\pi_j + w_j a_j < 0\}, i = 0, t = 0, (e_{g,o} = 0)_{(g,o)\in\mathcal{Q}})$, where $i = 0$ is a dummy aircraft with $d_{g_0,o_0,g_j,o_j} = 0$ for any $j \in \mathcal{N}$. We define $\mathcal{R}$ as a set resource variable where greater is preferred (Inequality (8b)). All numeric variables are also resource variables, where less is preferred. The dual bound function in Inequality (8c) is based on the fact that the schedule time of aircraft operation j is at earliest $t'(j)$. The dual bound function takes the sum of a parameterized expression $\min\{0, -\pi_j + w_j t'(j)\}$ with a set expression $\mathcal{R}$.

Following the previous work [10], branching introduces constraints specifying aircraft k must or must not be scheduled immediately after j. With this constraint, scheduling aircraft with a positive cost can be beneficial; if aircraft j with a large negative dual value cannot be scheduled after the current aircraft i but can be after another aircraft k, we may want to schedule k even if its cost is positive. Thus, except for the root node of B&P, we use $b'_k = b_k$ in $\mathcal{R}'(j)$ and $(\mathcal{R} = \mathcal{N}, i = 0, t = 0, (e_{g,o} = 0)_{(g,o)\in\mathcal{Q}})$ as the target state.

7 Empirical Evaluation

To evaluate the overall performance of our framework as an exact method, we apply DIDP as a pricing solver in B&P. We compare it with generic B&P baselines and conduct an ablation study to evaluate the importance of the new DIDP features. We use the following four problem classes:

- **VRPTW:** The pricing problem is the SPPRC used in our running example. We present the MP in Appendix A. We use the Solomon [29] instances with 50 and 100 customers. There are 112 instances in total.
- **Pickup and delivery problem with time windows (PDPTW) [3]:** The pricing problem is similar to that of VRPTW. The DIDP model uses the filter operation, a set resource variable, and the fractional knapsack expression as the dual bound function. We show the MP and the DIDP model for pricing based on previous work [25] in Appendix A. We use all 56 100-customer instances from Li and Lim [19].
- $P||\sum w_j C_j$**:** We generate instances following previous work [30]. In particular, we use $n = 20, 30, 40, 50$ and $m = 3, 4, 5$ with three different configurations for p_j and w_j: p_j

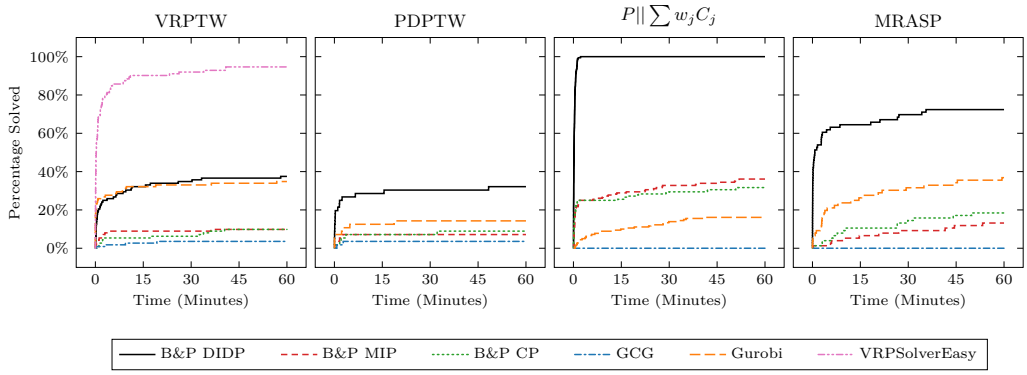


Figure 1 Percentage of instances solved over time by different approaches.

uniformly sampled from $[1, 10]$ and w_j uniformly sampled from $[10, 100]$, p_j and w_j uniformly sampled from $[1, 100]$, and p_j and w_j uniformly sampled from $[10, 20]$. For each of the 36 configurations, we generate five instances, resulting in 180 instances in total.

- **MRASP:** We use the 76 instances published in [10].

We add the new features for DIDP to didp-rs v0.9.0, implemented in Rust. In our B&P implementation (B&P DIDP), we use SCIP 9.2.1 with its Python interface PySCIPOpt and DIDPPy, the Python interface for didp-rs. For each problem class, we formulate the MIP model for the RMP using PySCIPOpt and implement a pricer by formulating the DIDP model using DIDPPy. Using the pricer, SCIP generates variables after solving each linear relaxation in branch-and-bound. We also implement problem-specific branching rules using PySCIPOpt (see Section 6 for $P||\sum w_jC_j$ and MRASP and Appendix A for VRPTW and PDPTW). We use Farkas pricing [8] to generate columns for infeasible linear programs, adjusting the objective value and the dual bound function in the pricing problems. Our code is provided as supplementary material. Each instance is solved using a single thread on an Intel Xeon Gold 6338 CPU with a time-out of 1 hour.

7.1 Comparison with Generic Branch-and-Price Baselines

We implement MIP pricing (B&P MIP), where the pricing problems are formulated as MIP models and solved by Gurobi 13.0.2 [12] via its Python interface gurobipy. We also implement CP pricing (B&P CP), where pricing problems are formulated as CP models and solved with Google OR-Tools CP-SAT [22] via its Python interface. We use an existing MIP model of pricing problems for VRPTW [14] and MRASP [9]. We implement MIP models for the other two problem classes and CP models for all problem classes (presented in Appendix A). We run the pricer until it finds an optimal solution or reaches the 1-hour time limit. The DIDP, MIP, and CP solvers possibly find suboptimal solutions before the optimal one. From these solutions, we add all variables with negative reduced cost, so different pricing solvers may generate different sets of variables and require different numbers of iterations to converge.

As an automated B&P baseline, we use GCG 3.5.5 with its Python interface PyGCGOpt. GCG takes a compact MIP formulation as input. We use existing MIP models for VRPTW [31], PDPTW [7], and MRASP [10]. For VRPTW and PDPTW, we evaluate two variations of existing MIP models [31, 7], the two-index and three-index formulations, and use the better one: three-index for GCG in VRPTW and two-index for the rest. For $P||\sum w_jC_j$, we create a compact MIP formulation, presented in Appendix A. We also solve the compact

Table 1 Comparison of B&P DIDP against B&P MIP and B&P CP in the number of pricing iterations (#iterations), the average number of columns generated for a pricing problem (Avg. #columns), and the average solving time for a pricing problem (Avg. time). We divide the competitor's value by B&P DIDP's, averaged over the co-solved instances, with the standard deviation shown.

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

formulations using Gurobi with gurobipy.

Figure 1 shows the percentage of optimally solved instances over time. B&P DIDP is stronger than the generic B&P baselines and Gurobi on all four problems. For VRPTW, we also evaluate VRPSolver via its open-source Python interface, VRPSolverEasy [6], which clearly shows how far behind all generic approaches are. To our knowledge, VRPSolverEasy is not compatible with other problem classes.

Table 1 compares B&P DIDP with B&P MIP and B&P CP by showing the average relative ratio of the number of pricing iterations, the average number of generated columns for each pricing problem, and the average time to solve each pricing problem. In VRPTW, PDPTW, and MRASP, MIP and CP require more pricing iterations than DIDP on average, possibly because fewer columns are generated per pricing iteration. In $P||\sum w_j C_j$, DIDP always generates one column, which is an optimal solution, for each pricing problem and requires more pricing iterations than MIP and CP. In terms of solving time, DIDP is faster than MIP and CP in all problem classes on average, up to hundreds of times, while the standard deviation is large due to the small number of co-solved instances. Except for $P||\sum w_j C_j$, both faster solving time per iteration and the smaller number of total iterations due to more columns per iteration seem to contribute to the overall superiority of DIDP.

7.2 Ablation Study of the New DIDP Features

Figure 2 shows the performance of B&P DIDP with different configurations, and Table 2 compares pricing statistics of different configurations, including the number of state expansions (the number of times line 10 of Algorithm 1 is reached). Our base configuration, 'Full', uses the labeling solver with all the modeling features. Each new feature contributes to performance improvements across multiple problem classes. Full is best in VRPTW, but other configurations are better in other problem classes, though it does not lose much.

7.2.1 Comparison of Different DIDP Solvers

We evaluate configurations using CAASDy or CABS as a DIDP solver, instead of labeling. These algorithms expand states minimizing f-values. While CAASDy is based on A*, it is

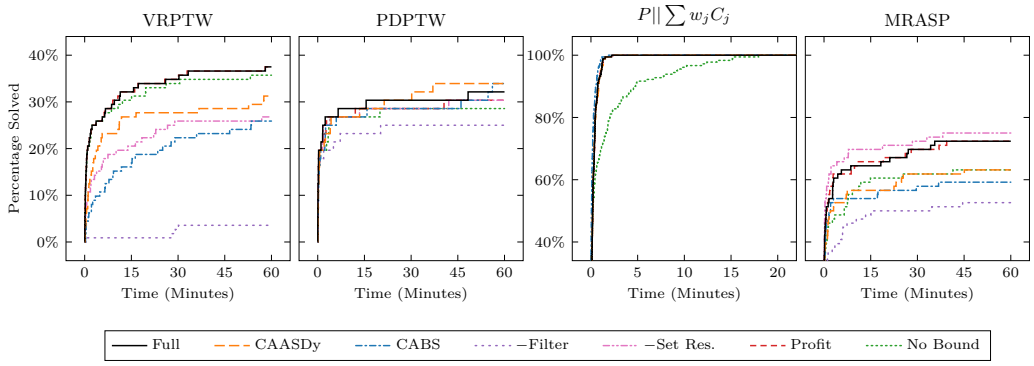


Figure 2 Percentage of instances solved over time by different configurations of DIDP pricers.

Table 2 Comparison of labeling against CAASDy and CABS in the number of pricing iterations (#iterations), the average number of columns generated for a pricing problem (Avg. #columns), the average solving time for a pricing problem (Avg. time), and the average number of expansions for a pricing problem (Avg. #expansions). We show the competitor's value divided by Full's, averaged over the co-solved instances, with the standard deviation.

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
$P\|\|\sum w_jC_j$	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

generalized to support negative costs [17]. In VRPTW and PDPTW, as shown in Table 2, CAASDy reduces the number of expansions and solving time on average. However, CAASDy also reduces the number of columns generated per iteration and results in more pricing iterations. As a result, in Figure 2, labeling is better in VRPTW, while CAASDy solves a few more instances in PDPTW. In MRASP, CAASDy increases the number of expansions and solving time on average, and is outperformed by labeling.

In VRPTW, PDPTW, and MRASP, CABS increases the number of expansions and solving time compared with labeling on average. In multiple instances of PDPTW, CABS reduces the number of expansions and solving time, resulting in a few more solved instances than labeling. In $P||\sum w_jC_j$, the DIDP model does not have resource variables, so Full uses CAASDy, which is equivalent to labeling. In this problem, CABS increases the number of expansions but reduces the solving time on average, which can be attributed to the low-level implementation differences between CAASDy and CABS.

In summary, expanding the state with the minimum f-value can solve a pricing problem faster, but overall performance may degrade due to fewer columns generated. This result confirms the importance of heuristic pricing, which generates promising columns before exactly solving a pricing problem, as used in the literature (e.g., [25, 10]). Designing a DIDP solver combining heuristic pricing with exact solving is future work.

Table 3 Comparison of Full against −Filter and −Set Res. in the number of pricing iterations (#iterations), the average number of columns generated for a pricing problem (Avg. #columns), the average solving time for a pricing problem (Avg. time), and the average number of expansions for a pricing problem (Avg. #expansions). We show the competitor's value divided by Full's, averaged over the co-solved instances, with the standard deviation.

−Filter vs. Full	#iterations	Avg. #columns	Avg. time (s)	Avg. #expansions
VRPTW	0.94 ± 0.18	1.07 ± 0.14	82.51 ± 91.16	27.11 ± 18.23
PDPTW	0.94 ± 0.14	1.13 ± 0.16	4.58 ± 6.53	5.09 ± 2.66
MRASP	0.93 ± 0.16	1.11 ± 0.19	29.26 ± 87.70	3.72 ± 4.05
−Set Res. vs. Full	**#iterations**	**Avg. #columns**	**Avg. time (s)**	**Avg. #expansions**
VRPTW	1.01 ± 0.10	0.99 ± 0.05	11.11 ± 16.36	19.99 ± 37.65
PDPTW	1.05 ± 0.17	0.99 ± 0.07	1.30 ± 0.28	1.42 ± 0.34
MRASP	1.13 ± 0.30	0.82 ± 0.10	3.21 ± 10.51	7.29 ± 21.57

7.2.2 Comparison of Different DIDP Models

Disabling the filter operation (−Filter) consistently increases the number of state expansions and the solving time, resulting in worse performance across all problem classes. Disabling the set resource variables (−Set Res.) increases the number of expansions, but its effect on overall performance depends on the problem class. In VRPTW, it significantly increases the average solving time and results in worse overall performance. In PDPTW, the effect is moderate, but Full solves two more instances than −Set Res. In MRASP, while Full is faster on average, −Set Res. is faster in multiple instances, resulting in a few more instances solved. When a set resource variable is used, we may detect more dominated states, but it increases the cost of each dominance check; we must compare each state against more states sharing the same non-resource variable values, and this cost does not necessarily pay off in MRASP.

We also evaluate Profit, a configuration with a dual bound function that takes the sum over all negative reduced cost items: the fractional knapsack bound with an infinite capacity for VRPTW, PDPTW, and $P||\sum w_j C_j$ (Equation (3) for VRPTW), and $\sum_{j \in \mathcal{R}} \min\{0, -\pi_j + w_j a_j\}$ for MRASP, ignoring the current time t. The configuration No Bound does not use a dual bound function. We show the pricing statistics in Table 4 in Appendix B. In all problem classes, No Bound is the worst, increasing the number of expansions and solving time. Profit also increases the number of expansions in PDPTW, $P||\sum w_j C_j$, and MRASP. In VRPTW and $P||\sum w_j C_j$, Profit has almost the same solving time as Full on average. In PDPTW, Full solves one more instance than Profit. In MRASP, Profit increases the solving time on average, but solves the same number of instances as Full.

8 Conclusion

We investigate DIDP as a generic pricing solver for column generation, introducing a generic labeling algorithm and three modeling features. Using these additions, we model the pricing problems declaratively and implement branch-and-price for four routing and scheduling problems. The experiments show that DIDP pricing is substantially faster than generic pricing using MIP and CP, and highlight that the new features are beneficial in multiple problem classes. To fill the gap with problem-specific branch-and-price, developing DIDP solvers for heuristic pricing and incorporating problem-specific cuts are promising directions.

References

1 Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. Constraint integer programming: A new approach to integrate CP and MIP. In *CPAIOR*, pages 6–20, 2008. doi:10.1007/978-3-540-68155-7_4.

2 George B. Dantzig. Discrete-variable extremum problems. *Oper. Res.*, 5(2):266–277, 1957. doi:10.1287/opre.5.2.266.

3 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.

4 W. L. Eastman, S. Even, and I. M. Isaacs. Bounds for the optimal scheduling of n jobs on m processors. *Manag. Sci.*, 11(2):268–279, 1964. doi:10.1287/mnsc.11.2.268.

5 Salah E. Elmaghraby and Sung H. Park. Scheduling jobs on a number of identical machines. *AIIE Trans.*, 6(1):1–13, 1974. doi:10.1080/05695557408974926.

6 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.

7 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.

8 Gerald Gamrath and Marco E. Lübbecke. Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In *SEA*, pages 239–252, 2010. doi:10.1007/978-3-642-13193-6_21.

9 Ahmed Ghoniem and Farbod Farhadi. A column generation approach for aircraft sequencing problems: a computational study. *J. Ope. Res. Soc.*, 66(10):1717–1729, 2015. doi:10.1057/jors.2014.131.

10 Ahmed Ghoniem, Farbod Farhadi, and Mohammad Reihaneh. An accelerated branch-and-price algorithm for multiple-runway aircraft sequencing problems. *Eur. J. Oper. Res.*, 246(1):34–43, 2015. doi:10.1016/j.ejor.2015.04.019.

11 Stefano Gualandi and Federico Malucelli. Constraint programming-based column generation. *Ann. Oper. Res.*, 204(1):11–32, 2013. doi:10.1007/s10479-012-1299-7.

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

13 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968. doi:10.1109/TSSC.1968.300136.

14 Stefan Irnich and Guy Desaulniers. Shortest path problems with resource constraints. In Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors, *Column Generation*, pages 33–65. Springer US, Boston, MA, 2005. doi:10.1007/0-387-25486-2_2.

15 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, Boston, MA, 1972. doi:10.1007/978-1-4684-2001-2_9.

16 Ryo Kuroiwa and J. Christopher Beck. RPID: Rust Programmable Interface for Domain-Independent Dynamic Programming. In *CP*, volume 340, pages 23:1–23:21, 2025. doi:10.4230/LIPIcs.CP.2025.23.

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

18 Jean-Louis Lauriere. A language and a program for stating and solving combinatorial problems. *Artif. Intell.*, 10(1):29–127, 1978. doi:10.1016/0004-3702(78)90029-2.

19 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.

20 Leonardo Lozano, Daniel Duque, and Andrés L. Medaglia. An exact algorithm for the elementary shortest path problem with resource constraints. *Transp. Sci.*, 50(1):348–357, 2016. doi:10.1287/trsc.2014.0582.

21 Marco E Lübbecke and Jacques Desrosiers. Selected topics in column generation. *Oper. Res.*, 53(6):1007–1023, 2005. doi:10.1287/opre.1050.0234.

22 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.

23 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.

24 Luigi Di Puglia Pugliese and Francesca Guerriero. A survey of resource constrained shortest path problems: Exact solution approaches. *Networks*, 62(3):183–200, 2013. doi:10.1002/net.21511.

25 Stefan Ropke and Jean-François Cordeau. Branch and cut and price for the pickup and delivery problem with time windows. *Transp. Sci.*, 43(3):267–286, 2009. doi:10.1287/trsc.1090.0272.

26 Stuart Russell and Peter Norvig. Solving problems by searching. In *Artificial Intelligence: A Modern Approach*, chapter 3, pages 63–109. Pearson, fourth edition, 2020.

27 Ruslan Sadykov, Eduardo Uchoa, and Artur Pessoa. A bucket graph-based labeling algorithm with application to vehicle routing. *Transp. Sci.*, 55(1):4–28, 2021. doi:10.1287/trsc.2020.0985.

28 Matteo Salani, Saverio Basso, and Vincenzo Giuffrida. PathWyse: a flexible, open-source library for the resource constrained shortest path problem. *Optim. Method. Softw.*, 39(2):298–320, 2024. doi:10.1080/10556788.2023.2296978.

29 Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Oper. Res.*, 35(2):254–265, 1987. doi:10.1287/opre.35.2.254.

30 J. M. van den Akker, J. A. Hoogeveen, and S. L. van de Velde. Parallel machine scheduling by column generation. *Oper. Res.*, 47(6):862–872, 1999. doi:10.1287/opre.47.6.862.

31 Daniele Vigo and Paolo Toth, editors. *Vehicle Routing: Problems, Methods, and Applications*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, second edition, 2014.

32 Weixiong Zhang. Complete anytime beam search. In *AAAI*, pages 425–430, 1998.

A Additional Models

We present column generation models that are not covered in the main text. In addition, we show a compact MIP formulation for $P||\sum w_j C_j$.

A.1 VRPTW

In VRPTW, a directed graph $(\mathcal{N}, \mathcal{A})$ is given, where $\mathcal{N} = \{0, \dots, n+1\}$ is the set of nodes, and $\mathcal{A} \subseteq \{\mathcal{N} \times \mathcal{N} : i \neq j, i < n+1, j > 0\}$. Each customer i has load $l_i > 0$, release time $a_i \geq 0$, deadline $b_i \geq 0$, and service duration $s_i \geq 0$. Each arc (i, j) has distance $d_{i,j} \geq 0$ and travel cost $c_{i,j}$. A solution is to minimize the total travel cost of a set of paths visiting all customers $\{1, \dots, n\}$ exactly once. Each path satisfies the following conditions: it starts from node 0 and ends at node $n+1$; the cumulative load never exceeds capacity Q; and the visit to each node i occurs within $[a_i, b_i]$.

Let $\mathcal{P}$ be a set of paths, c_p be the total travel cost of path $p \in \mathcal{P}$, and $a_{p,i}$ be a constant such that $a_{p,i} = 1$ if path p visits node $i \in \mathcal{N}$ and $a_{p,i} = 0$ otherwise. Problem (9) shows the

MP, using a binary variable λ_p representing the selection of path p.

$$\min \sum_{p \in \mathcal{P}} c_p \lambda_p \tag{9a}$$

$$\sum_{p \in \mathcal{P}} a_{p,j} \lambda_p = 1 \qquad \forall j = 1, \dots, n \tag{9b}$$

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

For pricing, we have shown the DIDP model as a running example, using the travel cost $c_{i,j} = -\pi_j + d_{i,j}$, where π_j is the dual value for Constraint (9b). The MIP model is taken from previous work [14]. Branching introduces constraints specifying that edge (j, k) must be or must not be used, where (j, k) is the most fractional edge.

A.1.1 CP Model for Pricing

We use a Boolean variable $x_{i,j} \in \{\bot, \top\}$ indicating that edge (i, j) is used, a Boolean variable y_i indicating that a node i is visited, and an integer variable t_i representing the time when node i is visited.

$$\min \sum_{i=1,\dots,n} -\pi_i \mathbb{1}(y_i) + \sum_{(i,j) \in \mathcal{A}} d_{i,j} \mathbb{1}(x_{i,j}) \tag{10a}$$

$$\mathsf{Circuit}\left(\{(i, j, x_{i,j}) \mid (i, j) \in \mathcal{A}\} \cup \{(i, i, \neg y_i \mid i = 1, \dots, n)\} \cup \{(n+1, 0, \top)\}\right) \tag{10b}$$

$$x_{i,j} \implies t_j \geq t_i + s_i + d_{i,j} \qquad \forall (i, j) \in \mathcal{A} \tag{10c}$$

$$\sum_{i=1,\dots,n} l_i \mathbb{1}(y_i) \leq Q \tag{10d}$$

$$t_i \in [a_i, b_i] \qquad \forall i \in \mathcal{N} \tag{10e}$$

$$x_{i,j} \in \{\bot, \top\} \qquad \forall (i, j) \in \mathcal{A} \tag{10f}$$

$$y_i \in \{\bot, \top\} \qquad \forall i = 1, \dots, n. \tag{10g}$$

In the objective function, we use the function $\mathbb{1} : \{\bot, \top\} \to \{0, 1\}$ such that $\mathbb{1}(\top) = 1$ and $\mathbb{1}(\bot) = 0$. We use Circuit, the circuit global constraint [18] implemented in Google OR-Tools CP-SAT, which takes a set of triples $(i, j, x_{i,j})$ as input, ensuring that edge (i, j) is used in a Hamiltonian circuit if and only if the Boolean variable $x_{i,j} = \top$. We also introduce a triple $(i, i, \neg y_i)$, ensuring that node i is not in the circuit if and only if $y_i = \bot$, and $(n+1, 0, \top)$ to make a path from 0 to $n+1$ a circuit.

A.2 PDPTW

In PDPTW, a vehicle picks up an item at one customer and delivers it to another customer, while respecting time windows [3]. Let m be the number of tasks, $\mathcal{N} = \{1, \dots, n\}$ be the set of tasks, and $\mathcal{L} = \{0, \dots, 2n+1\}$ be the set of locations. Each task $i \in \mathcal{N}$ is associated with a pickup location $i \in \mathcal{L}$ and a delivery location $n + i \in \mathcal{L}$. Nodes 0 and $2n+1$ represent the start and end depot locations, respectively.

Every task $i \in N$ has a load $l_i \geq 0$, and every location $i \in \mathcal{L}$ has release time $a_i \geq 0$, deadline $b_i \geq 0$ and service duration $s_i \geq 0$. Each vehicle has the maximum capacity of Q. Let $\mathcal{A} = \{(i, j) \in \mathcal{L} \times \mathcal{L} : i \neq j, i < 2n+1, j > 0\}$ be the set of arcs. Every arc $(i, j) \in A$ has a travel distance $d_{i,j} \geq 0$. The objective is to minimize the number of vehicles used and the total travel distance. Following the previous work [7], our objective function is the sum of the number of used vehicles multiplied by a constant penalty u and the total travel distance,

where $u = 10000$. For all evaluated methods, we tighten time windows and reduce arcs by preprocessing, following [3].

The MP is exactly the same as Problem (9). Branching introduces constraints specifying that edge (j, k) must be or must not be used.

A.2.1 DIDP Model for Pricing

In the pricing DIDP model, we use five state variables: $\mathcal{R}$ is the set of tasks whose pickup locations are reachable; $\mathcal{O}$ is the set of tasks whose pickup locations are visited; i is the current location; q is the current load; and t is the current time. The original problem corresponds to the target state $(\mathcal{R} = \mathcal{N}, \mathcal{O} = \emptyset, i = 0, q = 0, t = 0)$.

$$V(\mathcal{R}, \mathcal{O}, i, q, t) = \min_{j \in \text{Next}(\mathcal{R}, \mathcal{O}, i, q, t)} \begin{cases} 0 & \text{if } i = 2n+1 \\ d_{i,j} - \pi_j + V(\mathcal{R}'(j), \mathcal{O} \cup \{j\}, j, q + l_j, t'(j)) & \text{if } j \in \mathcal{N} \\ d_{i,j} + V(\mathcal{R}'(j), \mathcal{O} \setminus \{j-n\}, j, q - l_{j-n}, t'(j)) & \text{if } j - n \in \mathcal{N} \end{cases} \tag{11a}$$

$$V(\mathcal{R}, \mathcal{O}, i, q, t) = \infty \qquad \text{if } \exists j \in \mathcal{O}, t + s_i + d^*_{i,n+j} > b_{n+j} \tag{11b}$$

$$V(\mathcal{R}_1, \mathcal{O}, i, q_1, t_1) \leq V(\mathcal{R}_2, \mathcal{O}, i, q_2, t_2) \qquad \text{if } \mathcal{R}_2 \subseteq \mathcal{R}_1 \wedge q_1 \leq q_2 \wedge t_1 \leq t_2 \tag{11c}$$

$$V(\mathcal{R}, \mathcal{O}, i, q, t) \geq \max \begin{cases} 0 \text{ if } i = 2n+1 \\ -\mathsf{fractional\_knapsack}\left(\mathcal{R}, b_{2n+1} - t - d^{\text{in}}(\mathcal{O}), \left(v^{\text{in}}_j\right)_{j=1,\dots,n}, \left(w^{\text{in}}_j\right)_{j=1,\dots,n}\right) \\ -\mathsf{fractional\_knapsack}\left(\mathcal{R}, b_{2n+1} - t - d^{\text{out}}(\mathcal{O}), \left(v^{\text{out}}_j\right)_{j=1,\dots,n}, \left(w^{\text{out}}_j\right)_{j=1,\dots,n}\right) \end{cases} \tag{11d}$$

Equation (11a) is the Bellman equation. The base case is when $i = 2n + 1$. Each transition visits a location $j \in \text{Next}(\mathcal{R}, \mathcal{O}, i, q, t)$, where $\text{Next}(\mathcal{R}, \mathcal{O}, i, q, t) = \{j \in \mathcal{R} : (i, j) \in \mathcal{A}, t + s_i + d_{i,j} \leq b_j, q + l_j \leq Q\} \cup \{j + n : j \in \mathcal{O}, (i, j + n) \in \mathcal{A}, t + s_i + d_{i,j+n} \leq b_{j+n}\} \cup \{2n + 1 : q = 0\}$. When a location j is visited, i is updated to j, and t is updated to $t'(j) = \max\{t + s_i + d_{i,j}, a_j\}$. Using the filter operation, $\mathcal{R}$ is updated to $\mathcal{R}'(j) = \{k \in \mathcal{R} \setminus \{j\} \mid t'(j) + s_j + d^*_{j,k} \leq b_k\}$, where $d^*_{j,k}$ is the precomputed shortest possible time to reach location k from j. If j is a pickup location, $\mathcal{O}$ is updated to $\mathcal{O} \cup \{j\}$, and q is updated to $q + l_j$. If j is a delivery location, $\mathcal{O}$ is updated to $\mathcal{O} \setminus \{j\}$, and q is updated to $q - l_j$. Equation (11b) ensures that the delivery location of each task in $\mathcal{O}$ must be visited by the deadline. Inequality (11c) defines state dominance using $\mathcal{R}$, q, and t as resource variables. Inequality (11d) is the dual bound function. We use the fractional knapsack bound considering the deadline for the end depot. Using $d^{\text{in}}_j = \min_{k \in \mathcal{L}:(k,j) \in \mathcal{A}} d_{k,j}$, to complete the tasks $\mathcal{O}$ and return to the end depot, we need at least $d^{\text{in}}(\mathcal{O}) = \sum_{j \in \mathcal{O}} \left(d^{\text{in}}_{n+j} + s_{n+j}\right) + d^{\text{in}}_{2n+1}$. Completing task j increases the cost by at least $v^{\text{in}}_j = d^{\text{in}}_j - \pi_j + d^{\text{in}}_{n+j}$ and the time by at least $w^{\text{in}}_j = d^{\text{in}}_j + s_j + d^{\text{in}}_{n+j} + s_{n+j}$. We consider the 0-1 knapsack problem with the capacity $b_{2n1} - t - d^{\text{in}}(\mathcal{O})$, where each item $j \in \mathcal{R}$ has profit v^{in}_j and weight w^{in}_j. We also use a similar bound using $d^{\text{out}}_j = \min_{k \in \mathcal{L}:(j,k) \in \mathcal{A}} d_{j,k}$, $d^{\text{out}}(\mathcal{O}) = d^{\text{out}}_i + \sum_{j \in \mathcal{O}} \left(s_{n+j} + d^{\text{out}}_{n+j}\right)$, $v^{\text{out}}_j = d^{\text{out}}_j - \pi_j + d^{\text{out}}_{n+j}$, and $w^{\text{out}}_j = s_j + d^{\text{out}}_j + s_{n+j} + d^{\text{out}}_{n+j}$. The reduced cost is computed by adding the penalty u to the objective value.

A.2.2 MIP Model for Pricing

In the pricing MIP model, we use a binary variable $x_{i,j} \in \{0, 1\}$ indicating that edge (i, j) is traversed, continuous variables t_i and q_i to represent the time and load at location i.

$$\min \sum_{i \in \mathcal{N}} -\pi_i \sum_{(i,j) \in \mathcal{A}} x_{i,j} + \sum_{(i,j) \in \mathcal{A}} d_{i,j} x_{i,j} \tag{12a}$$

$$\sum_{(0,j)\in\mathcal{A}} x_{0,j} = \sum_{(i,2n+1)\in\mathcal{A}} x_{i,2n+1} = 1 \tag{12b}$$

$$\sum_{(i,j)\in\mathcal{A}} x_{i,j} = \sum_{(j,i)\in\mathcal{A}} x_{j,i} \qquad \forall i = 1,\dots,2n \tag{12c}$$

$$\sum_{(i,j)\in\mathcal{A}} x_{i,j} = \sum_{(n+i,j)\in\mathcal{A}} x_{n+i,j} \qquad \forall i \in \mathcal{N} \tag{12d}$$

$$t_{n+i} \geq t_i + s_i + d^*_{i,n+i} \qquad \forall i \in \mathcal{N} \tag{12e}$$

$$t_j \geq t_i + s_i + d_{i,j} - M(1 - x_{i,j}) \qquad \forall (i,j) \in \mathcal{A} \tag{12f}$$

$$q_j \geq q_i + l_j - M(1 - x_{i,j}) \qquad \forall (i,j) \in \mathcal{A}, j \in \mathcal{N} \tag{12g}$$

$$q_{n+j} \geq q_i - l_j - M(1 - x_{i,n+j}) \qquad \forall (i,n+j) \in \mathcal{A}, j \in \mathcal{N} \tag{12h}$$

$$q_0 = 0 \tag{12i}$$

$$q_i \in [l_i, Q], q_{n+i} \in [0, Q - l_i] \qquad \forall i \in \mathcal{N} \tag{12j}$$

$$t_i \in [a_i, b_i] \qquad \forall i \in \mathcal{L} \tag{12k}$$

$$x_{i,j} \in \{0,1\} \qquad \forall (i,j) \in \mathcal{A}. \tag{12l}$$

Constraint (12d) ensures that the pickup location is visited if and only if the corresponding delivery location is visited. Constraint (12e) ensures that the delivery location is visited after the corresponding pickup location. Constraint (12f) ensures that time window constraints, using $M = b_i + s_i + d_{i,j} - a_j$. Constraints (12g) and (12h) ensure the capacity constraints at pickup and delivery locations, using $M = \overline{q}_i$ and $M = \overline{q}_i - l_j$, respectively, where $\overline{q}_i$ is an upper bound on q_i defined in Constraint (12i) and (12j).

A.2.3 CP Model

The pricing CP model is based on the same idea as the MIP model and similar to that of VRPTW. We use Boolean variable $x_{i,j} \in \{\bot, \top\}$ to indicate that edge (i,j) is traversed, $y_i \in \{\bot, \top\}$ to indicate that task i is completed, the circuit global constraint, and logical implications instead of big-M constraints. We input triples $(i, i, \neg y_i)$ and $(n+i, n+i, \neg y_i)$ to the circuit constraint to ensure that i and $n+i$ are visited if and only if $y_i = \top$.

$$\min \sum_{i\in\mathcal{N}} -\pi_i \mathbb{1}(y_i) + \sum_{(i,j)\in\mathcal{A}} d_{i,j}\mathbb{1}(x_{i,j}) \tag{13a}$$

$$\mathsf{Circuit}\left(\begin{array}{l}\{(i,j,x_{i,j}) \mid (i,j) \in \mathcal{A}\} \\ \cup\{(i,i,\neg y_i) \mid i \in \mathcal{N})\} \cup \{(n+i, n+i, \neg y_i) \mid i \in \mathcal{N}\} \cup \{(2n+1, 0, \top)\}\end{array}\right) \tag{13b}$$

$$x_{i,j} \implies t_j \geq t_i + s_i + d_{i,j} \qquad \forall (i,j) \in \mathcal{A} \tag{13c}$$

$$x_{i,j} \implies q_j = q_i + l_j \qquad \forall (i,j) \in \mathcal{A}, j \in \mathcal{N} \tag{13d}$$

$$x_{i,n+j} \implies q_{n+j} = q_i - l_j \qquad \forall (i,n+j) \in \mathcal{A}, j \in \mathcal{N} \tag{13e}$$

$$t_{n+i} \geq t_i + s_i + d^*_{i,n+i} \qquad \forall i \in \mathcal{N} \tag{13f}$$

$$q_0 = 0 \tag{13g}$$

$$q_i \in [l_i, Q], q_{n+i} \in [0, Q - l_i] \qquad \forall i \in \mathcal{N} \tag{13h}$$

$$t_i \in [a_i, b_i] \qquad \forall i \in \mathcal{L} \tag{13i}$$

$$x_{i,j} \in \{\bot, \top\} \qquad \forall (i,j) \in \mathcal{A} \tag{13j}$$

$$y_i \in \{\bot, \top\} \qquad \forall i \in \mathcal{N}. \tag{13k}$$

A.3 Parallel Machine Scheduling

We present the MIP and CP models for pricing and the compact MIP model given to GCG.

A.3.1 MIP Model for Pricing

In the pricing MIP model, we use a binary variable x_j indicating that job j is scheduled and a continuous variable C_j to represent the completion time of job j.

$$\min \sum_{j \in \mathcal{J}} -\pi_j x_j + w_j C_j \tag{14a}$$

$$C_j \geq (a_j + p_j) x_j \qquad \forall j \in \mathcal{J} \tag{14b}$$

$$C_j \geq \sum_{k=1}^{j} p_k x_k - M(1 - x_j) \qquad \forall j \in \mathcal{J} \tag{14c}$$

$$C_j \in [0, b_j] \qquad \forall j \in \mathcal{J} \tag{14d}$$

$$x_j \in \{0, 1\} \qquad \forall j \in \mathcal{J}. \tag{14e}$$

As mentioned in Section 6.1, if job j and job $k < j$ are scheduled, j is later than k. Thus, Constraint (14c) ensures that C_j becomes at least the sum of the processing times of scheduled jobs with $k \leq j$ if $x_j = 1$, using $M = \sum_{k=1}^{j} p_k$. Constraints (14b) and (14d) ensure the time windows.

A.3.2 CP Model for Pricing

The CP model is based on the same idea as the MIP model, while x_j is a Boolean variable.

$$\min \sum_{j \in \mathcal{J}} -\pi_j \mathbb{1}(x_j) + w_j C_j \tag{15a}$$

$$x_j \implies C_j \geq a_j + p_j \wedge C_j = p_j + \sum_{k=1}^{j-1} p_k x_k \qquad \forall j \in \mathcal{J} \tag{15b}$$

$$C_j \in [0, b_j] \qquad \forall j \in \mathcal{J} \tag{15c}$$

$$x_j \in \{\bot, \top\} \qquad \forall j \in \mathcal{J}. \tag{15d}$$

A.3.3 Compact MIP Model

In the compact MIP model, binary variable $x_{j,i}$ indicates that job j is assigned to machine i.

$$\min \sum_{j \in \mathcal{J}} w_j C_j \tag{16a}$$

$$\sum_{i=1}^{m} x_{j,i} = 1 \qquad \forall j \in \mathcal{J} \tag{16b}$$

$$C_j \geq \sum_{k=1}^{j} p_k x_{k,i} - M(1 - x_{j,i}) \qquad \forall j \in \mathcal{J}, \forall i = 1, \ldots, m \tag{16c}$$

$$C_j \in [a_j + p_j, b_j] \qquad \forall j \in \mathcal{J} \tag{16d}$$

$$x_{j,i} \in \{0, 1\} \qquad \forall j \in \mathcal{J}, \forall i = 1, \ldots, m. \tag{16e}$$

Constraint (16b) ensures a job is processed by exactly one machine.

A.4 CP Model for MRASP

The pricing CP model is similar to the existing pricing MIP model [10] but uses logical constraints. Boolean variable x_i indicates that aircraft i is scheduled, Boolean variable $y_{i,j}$ indicates that aircraft j is scheduled later than i, and an integer variable t_i represents the scheduled time for aircraft i.

$$\min \sum_{i \in \mathcal{N}} -\pi_i \mathbb{1}(x_i) + w_i t_i \tag{17a}$$

$$x_i \implies t_i \geq a_i \qquad \forall i \in \mathcal{N} \tag{17b}$$

$$y_{i,j} \implies x_i \wedge t_j \geq t_i + d_{g_i,o_i,g_j,o_j} \qquad \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \setminus \{i\} \tag{17c}$$

$$\mathsf{AtMostOne}(\{y_{i,j}, y_{j,i}\}) \qquad \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \setminus \{i\} \tag{17d}$$

$$x_i \wedge x_j \implies y_{i,j} \vee y_{j,i} \qquad \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \setminus \{i\} \tag{17e}$$

$$\neg y_{i,j} \qquad \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \setminus \{i\}, a_i + d_{g_i,o_i,g_j,o_j} > b_j \tag{17f}$$

$$\neg y_{j,i} \qquad \begin{array}{r} \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \setminus \{i\} \\ a_i < a_j, b_i \leq b_j, g_i = g_j, o_i = o_j \end{array} \tag{17g}$$

$$t_i \in [0, b_i] \qquad \forall i \in \mathcal{N} \tag{17h}$$

$$x_i \in \{\bot, \top\} \qquad \forall i \in \mathcal{N} \tag{17i}$$

$$y_{i,j} \in \{\bot, \top\} \qquad \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \setminus \{i\}. \tag{17j}$$

Constraints (17b) and (17h) ensure the time window constraints. Constraint (17c) ensures the separation time and link x_i and $y_{i,j}$ together with Constraint (17e) . Constraints (17f) and (17g) exclude impossible or suboptimal precedences and are adapted from the pricing MIP model used in the previous work.

In addition, in the root node of B&P, we introduce the following constraint, excluding aircraft i that cannot be scheduled with a negative reduced cost:

$$\neg x_i \qquad \forall i \in \mathcal{N}, -\pi_i + w_i a_i \geq 0. \tag{18a}$$

An equivalent constraint is also introduced in the pricing MIP model.

In branching, constraints to forbid or enforce that aircraft j is scheduled directly after i are introduced. Let $\mathcal{A}_{\text{forbidden}}$ and $\mathcal{A}_{\text{enforced}}$ be the set of pairs (i, j) such that scheduling j directly after i is forbidden or enforced. Then, we introduce the following constraints:

$$x_i \implies \bigvee_{j \in \mathcal{N} \setminus \{i\}} y_{j,i} \qquad \forall (0, i) \in \mathcal{A}_{\text{forbidden}} \tag{19a}$$

$$x_i \implies \bigvee_{j \in \mathcal{N} \setminus \{i\}} y_{i,j} \qquad \forall (i, n+1) \in \mathcal{A}_{\text{forbidden}} \tag{19b}$$

$$y_{i,j} \implies \sum_{k \in \mathcal{N} \setminus \{j\}} y_{k,j} - \sum_{k \in \mathcal{N} \setminus \{i\}} y_{k,i} \geq 2 \qquad \forall (i, j) \in \mathcal{A}_{\text{forbidden}} \tag{19c}$$

$$\neg y_{j,i} \qquad \forall (0, i) \in \mathcal{A}_{\text{enforced}}, \forall j \in \mathcal{N} \setminus \{i\} \tag{19d}$$

$$\neg y_{i,j} \qquad \forall (i, n+1) \in \mathcal{A}_{\text{enforced}}, \forall j \in \mathcal{N} \setminus \{i\} \tag{19e}$$

$$x_i \implies y_{i,j} \qquad \forall (i, j) \in \mathcal{A}_{\text{enforced}} \tag{19f}$$

$$x_j \implies y_{i,j} \qquad \forall (i, j) \in \mathcal{A}_{\text{enforced}} \tag{19g}$$

$$y_{i,j} \implies \sum_{k \in \mathcal{N} \setminus \{j\}} y_{k,j} - \sum_{k \in \mathcal{N} \setminus \{i\}} y_{k,i} = 1 \qquad \forall (i, j) \in \mathcal{A}_{\text{enforced}}. \tag{19h}$$

Constraint (19c) ensures that at least one aircraft is scheduled after i and before j. In a similar way, Constraint (19h) ensures that no aircraft is scheduled after i and before j. The linearized versions of the above constraints are used in the pricing MIP model.

B Additional Results

We present the pricing statistics for B&P DIDP configurations with different dual bound functions in Table 4.

Table 4 Comparison of Full against Profit and No Bound in the number of pricing iterations (#iterations), the average number of columns generated for a pricing problem (Avg. #columns), the average solving time for a pricing problem (Avg. time), and the average number of expansions for a pricing problem (Avg. #expansions). We show the competitor's value divided by Full's, averaged over the co-solved instances, with the standard deviation.

Profit vs. Full	#iterations	Avg. #columns	Avg. time (s)	Avg. #expansions
VRPTW	1.00 ± 0.00	1.00 ± 0.00	0.97 ± 0.04	1.00 ± 0.00
PDPTW	1.00 ± 0.00	1.00 ± 0.00	1.24 ± 0.45	1.35 ± 0.52
$P\|\|\sum w_j C_j$	1.01 ± 0.17	1.00 ± 0.01	1.00 ± 0.06	1.10 ± 0.06
MRASP	0.98 ± 0.18	1.01 ± 0.07	2.72 ± 6.10	1.68 ± 1.90
No Bound vs. Full	**#iterations**	**Avg. #columns**	**Avg. time (s)**	**Avg. #expansions**
VRPTW	1.00 ± 0.00	1.00 ± 0.00	1.12 ± 0.39	1.11 ± 0.11
PDPTW	1.00 ± 0.00	1.00 ± 0.00	1.60 ± 0.74	2.16 ± 1.04
$P\|\|\sum w_j C_j$	0.31 ± 0.10	21.14 ± 11.86	12.18 ± 16.57	94.71 ± 119.91
MRASP	1.08 ± 0.42	0.98 ± 0.08	11.19 ± 24.76	2.37 ± 2.71