RPID: Rust Programmable Interface for Domain-Independent Dynamic Programming

Ryo Kuroiwa
National Institute of Informatics, Tokyo, Japan
The Graduate University of Advanced Studies, SOKENDAI, Tokyo, Japan

J. Christopher Beck
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Abstract

In domain-independent dynamic programming (DIDP), a problem is formulated as a dynamic programming (DP) model and then solved by a general-purpose solver. In the existing software for DIDP, a model is defined using expressions composed of a predefined set of operations. In this paper, we propose the Rust Programmable Interface for DIDP (RPID), new software for DIDP, where a model is defined by Rust functions. We discuss the design of RPID and compare it with existing DP-based frameworks, including decision diagram-based (DD-based) solvers. In our experiments, RPID is up to hundreds of times faster than the existing DIDP implementation with the same models. In addition, new DIDP models, enabled by the flexibility of RPID, outperform existing models in multiple problem classes. We also show that the relative performance of RPID and existing DD-based solvers depends on problem class with, so far, no clear dominant solver technology.

2012 ACM Subject Classification Theory of computation → Dynamic programming

Keywords and phrases Decision Diagrams & Dynamic Programming, Modelling & Modelling Languages

Digital Object Identifier 10.4230/LIPIcs.CP.2025.23

Supplementary Material *Software*: `https://github.com/domain-independent-dp/rpid` [13]
archived at `swh:1:dir:9f071b7f5f3daf914c19752ae76692b3a8d1b108`
Software: `https://github.com/Kurororoto/didp-rust-models` [12]
archived at `swh:1:dir:810758e7707cef89647835b7e209d2c94e7bdfd2`

Funding SciNet is funded by ISED Canada; the Digital Research Alliance of Canada; Ontario Research Fund:RE; and the University of Toronto.
J. Christopher Beck: This work is supported by the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements Computations were performed on the Niagara supercomputer at the SciNet HPC Consortium. SciNet is funded by ISED Canada; the Digital Research Alliance of Canada; Ontario Research Fund:RE; and the University of Toronto.

1 Introduction

Dynamic programming (DP) [1] is a problem-solving methodology based on a state-based representation. Recent work has developed model-based paradigms for combinatorial optimization based on DP, where a problem is formulated as a declarative DP model and then solved by a general-purpose solver, similarly to constraint programming (CP). For such paradigms, there are currently two primary directions with different solving approaches: domain-independent dynamic programming (DIDP) [14, 17], using state space search algorithms for solvers, and decision diagram-based (DD-based) solvers [2, 8, 18], using branch-and-bound algorithms with graph data structures called decision diagrams (DDs).

The software implementation of DIDP, didp-rs [14, 17], provides three interfaces: a Rust library, a Python library, and a command-line interface that takes files written in a modeling language as input. In all interfaces, the DP model is described by expressions composed of a predefined set of operations. Internally, the expressions are converted to the same expression tree data structure in Rust and then evaluated by an interpreter. In contrast, the existing two DD-based solvers, ddo [8] and CODD [18], are libraries in which the model is defined by structs and functions in the corresponding programming language.

Ideally, a problem solving system exhibits strong computational performance while allowing modelers to quickly develop and experiment with problem formulations. Research progress for DP-based approaches has been slowed because the competing approaches differ in both the solver technology and the modeling interface. For example, it is unclear whether observed performance differences among solvers are due to underlying advantages of the solver technology or to the requirement for run-time interpretation of expressions. Similarly, it is difficult to determine whether improvements in modeling productivity are worthwhile if there is a performance impact. Indeed, this tension revives earlier debates in the CP community about "packages vs. languages" [20].

We propose the Rust Programmable Interface for DIDP (RPID), a new interface for DIDP, where a DP model is defined using Rust code, similar to ddo and CODD. If writing a model in Rust is acceptable, RPID is a faster and more flexible option for using DIDP than didp-rs. In addition, with RPID, we can compare DIDP and the DD-based solvers, excluding the fundamental differences in interface design. Our contributions are as follows:

- We introduce RPID, novel DIDP software, that is faster and more flexible.
- We discuss design choices in RPID, comparing it to existing software.
- We empirically compare the performance of RPID and existing DIDP and DD-based solver software. We show that RPID is up to hundreds of times faster than didp-rs, and each of RPID, ddo, and CODD outperforms the other two in different problem classes.
- We show that new DIDP models, facilitated by the flexibility of RPID, outperform existing DIDP models in four of five problem classes tested.

2 Dynamic Programming for Combinatorial Optimization

In dynamic programming (DP) [1], a problem is represented by a *state*, and the *value* of the state corresponds to the optimal objective value of the problem. In this paper, we assume that a state S is transformed into another state $S[\![\tau]\!]$, called *a successor state*, by applying a *transition* τ, and the value of a state $V(S)$ is recursively defined by the values of its successor states. To prevent infinite recursion, when a state S satisfies particular conditions, $V(S)$ is non-recursively defined by a function v: $V(S) = v(S)$. In such a case, we call S a *base state*. For computing $V(S)$, we introduce the following assumption: a weight function w_τ is associated with each transition, which returns the *transition weight* $w_\tau(S) \in \mathbb{Q}$ given a state S, and $V(S)$ is computed by applying a binary operator $\circ$, such as $+$, to $w_\tau(S)$ and $V(S[\![\tau]\!])$. Let $\mathcal{T}(S)$ be the set of applicable transitions in a state S. In a minimization problem, $V(S)$ is represented by the following recursive equation, called a Bellman equation:

$$V(S) = \begin{cases} v(S) & \text{if } S \text{ is a base state} \\ \min\limits_{\tau \in \mathcal{T}(S)} w_\tau(S) \circ V(S[\![\tau]\!]) & \text{otherwise.} \end{cases} \quad (1)$$

The optimal objective value of the problem can be computed by solving the above equation. For maximization, min is replaced with max. We assume that $V(S) = \infty$ if $\mathcal{T}(S) = \emptyset$ in the second line ($V(S) = -\infty$ for maximization).

As an example, in the traveling salesperson problem with time windows (TSPTW) [5] we are given a set of n customers $N = \{0, ..., n-1\}$, where 0 is the depot, and the travel time c_{ij} from customer i to j. A solution is a tour starting from the depot at time $t = 0$, visiting each customer i within its time window $[a_i, b_i]$, and returning to the depot. The objective function is to minimize the total travel time $\sum_{i=0}^{n-1} c_{x_i, x_{i+1}}$ where x_i is the i-th customer in the tour with $x_0 = x_n = 0$. In the DP formulation, a state is represented by a set of unvisited customers $U \subseteq N \setminus \{0\}$, the current location $i \in N$, and the current time $t \geq 0$. Each transition visits one of the unvisited customers j that can be reached by the deadline b_j. The Bellman equation is defined as

$$V(U, i, t) = \begin{cases} c_{i0} & \text{if } U = \emptyset \\ \min\limits_{j \in U : t + c_{ij} \leq b_j} c_{ij} + V(U \setminus \{j\}, j, \max\{t + c_{ij}, a_j\}) & \text{if } U \neq \emptyset. \end{cases} \tag{2}$$

The original problem corresponds to state $(N \setminus \{0\}, 0, 0)$.

2.1 Domain-Independent Dynamic Programming (DIDP)

Domain-independent dynamic programming (DIDP) is a model-based paradigm where a combinatorial optimization problem is formulated as a declarative DP model and is solved by a general-purpose solver [14, 17]. The model is defined in Dynamic Programming Description Language (DyPDL), which is explicitly designed for combinatorial optimization by allowing a user to incorporate redundant information via *state constraints*, *state dominance*, and a *dual bound function*.

A state constraint is a condition that must be satisfied by all states. In TSPTW, since we need to visit all customers, a state does not lead to a solution if one of the customers cannot be reached by its deadline. Let c^*_{ij} be the shortest travel time from customer i to j, which can be precomputed from the travel costs. A state (U, i, t) must satisfy $t + c^*_{ij} \leq b_j$ for each customer $j \in U$. Using the value function V,

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

This constraint is redundant, i.e., implied by Equation (2), but can be helpful for a solver.[1]

A state S dominates another state S' if the value of S is equal to or better than that of S', i.e., $V(S) \leq V(S')$ in minimization. In DyPDL, a user can explicitly define a sufficient condition for state dominance. In the TSPTW example, (U, i, t) is at least as good as (U, i, t') if $t \leq t'$, i.e.,

$$V(U, i, t) \leq V(U, i, t') \quad \text{if } t \leq t'. \tag{4}$$

A dual bound function η defines a lower/upper bound on $V(S)$ in minimization/maximization, i.e., $V(S) \geq \eta(S)$ for minimization. In TSPTW, the travel cost to visit customer j can be underestimated by $c^{\text{to}}_j = \min_{k \in N \setminus \{j\}} c_{kj}$. Given a state (U, i, t), the sum of c^{to}_j over $j \in U \cup \{0\}$ is a dual bound function. Similarly, the travel cost from j to a customer is underestimated by $c^{\text{from}}_j = \min_{k \in N \setminus \{j\}} c_{jk}$. In other words,

$$V(U, i, t) \geq \max\left\{ \sum_{j \in U \cup \{0\}} c^{\text{to}}_j, \sum_{j \in U \cup \{i\}} c^{\text{from}}_j \right\}. \tag{5}$$

[1] In general, a state constraint is not necessarily redundant.

2.1.1 State Space Search Solvers for DIDP

Kuroiwa and Beck [14, 16, 17] developed DIDP solvers using state space search algorithms. In particular, they used heuristic search algorithms such as A* [9] and beam search [21]. At each step, a state space search algorithm selects one state S from a set of candidates. Then, S is *expanded*, i.e., S is removed from the candidates and its successor states are added. In the beginning, the state corresponding to the original problem is the only candidate and, subsequently, the expansion of a state is repeated until termination criteria are met. The solvers use the dual bound function to select a state to expand and prune states that do not lead to a better solution than the current incumbent. The solvers also exploit state dominance to prune states that are known not to be better than another candidate or an already expanded state.

2.1.2 didp-rs: Software Implementation of DyPDL

Kuroiwa and Beck [14, 17] developed didp-rs, a software implementation of DyPDL and its solvers. There are four components: the modeling library dypdl, the solver library dypdl-heuristic-search, the command line interface didp-yaml, and the Python interface DIDPPy, all of which are implemented in Rust.

In didp-rs, states are defined by *state variables*, and each state variable has a type, either of *set*, *element*, *integer*, or *continuous*. In the TSPTW example, U is a set variable, i is an element variable, and t is an integer or continuous variable depending on a problem instance. The state dominance is specified by defining element, integer, and continuous variables as *resource variables* with a preference, either of less or greater. In our example, to represent Inequality (4), t is defined as a resource variable where less is preferred.

Transitions and base states are described by *expressions*, which are composed of predefined operations on state variables and evaluated given a state. In the TSPTW example, the condition to be a base state, $U = \emptyset$, and the condition to apply a transition, $t + c_{ij} \leq b_j$, are expressions returning a Boolean value. To update state variables, *set expression* $U \setminus \{j\}$, *element expression* j, and *integer or continuous expression* $\max\{t + c_{ij}, a_j\}$ are used. Expressions c_{i0} and c_{ij} are used to compute the value of a state. The state constraint (Equation (3)) and the dual bound function (Inequality (5)) are also defined by expressions.

With the dypdl library, a DP model is formulated in Rust by defining state variables and transitions. As of dypdl 0.8.0, an expression is represented as a tree data structure. The library overloads arithmetic operations in Rust and provides functions with which an expression tree can be constructed. The DP model can be solved by calling a solver implemented in dypdl-heuristic-search. During solving, to evaluate an expression tree given a state, a solver uses an interpreter implemented in the dypdl library. The two interfaces, didp-yaml and DIDPPy, convert a DP model written in a specific syntax into the data structure in dypdl and call solvers in dypdl-heuristic-search. In didp-yaml, YAML-DyPDL, a modeling language based on the YAML data format,[2] is implemented employing a LISP-like syntax for expressions. In DIDPPy, a user constructs expressions using Python syntax.

2.2 Decision Diagram-Based Solvers

Hooker [11] proposed that a DP formulation can be represented as a decision diagram (DD), a data structure based on a directed graph, and that a solution can be extracted from such a DD. Based on this observation, Bergman et al. [2] proposed a branch-and-bound algorithm

[2] `https://yaml.org`

to solve DPs based on DDs. Since constructing an exact DD, which fully represents the DP formulation, is computationally expensive, the algorithm repeatedly constructs restricted and relaxed DDs, which are computationally cheaper and provide bounds on the optimal objective value. To construct a relaxed DD, a function called a merge operator is required, which maps two states to a single state. DD-based branch-and-bound can also exploit state dominance and a dual bound function when they are defined [3, 7, 18].

2.2.1 Ddo

Ddo is a software library for DD-based solvers [8] in Rust and uses generics for modeling. In particular, ddo uses *traits*, which define an abstract interface. With traits, a function can be defined generically: one function definition is sufficient for different types of arguments as long as each type implements the required traits. In ddo 2.0.0, a DP model is formulated as a Rust program that defines a data type and implements a particular trait, Problem, for it. The data type for a state is defined as an *associated type*, State, and a concrete type is specified when implementing the trait. Problem requires seven methods:

1. initial_state returns the initial state, which corresponds to the original problem (for TSPTW, $(N \setminus \{0\}, 0, 0)$).
2. nb_variables defines the number of transitions in a solution, which is assumed to be fixed in all solutions. With this assumption, a base state is not defined.
3. for_each_domain defines transitions applied to a given state.
4. transition returns the successor state, given a state and a transition.
5. transition_cost returns the transition weight, given a state, a transition, and the resulting successor state.
6. initial_value returns a constant offset for the objective value. In ddo, the objective value of a solution is the sum of the transition weights and the offset.
7. next_variable controls the behavior of a solver and is not part of a DP model.

In addition, ddo requires the implementation of the Relaxation trait to define a merge operator and the StateRanking trait to define the order to select states to merge using the merge operator during solving. Note that the merge operator is an instruction to a solver rather than model information. A dual bound function is optionally defined in Relaxation, and state dominance is optionally defined by implementing the Dominance trait.

Ddo has a Python interface, Py-DDO, where a DP model is formulated as a Python class implementing a particular set of methods. Internally, Py-DDO defines a Rust struct that has a Python object as a member and implements Problem, Relaxation, and StateRanking for the struct by calling methods of the Python class.

2.2.2 CODD

CODD is a software library for DD-based solvers in C++ [18]. In CODD, a solver takes first-order functions defined as C++ lambda functions as input, representing a DP model and a merge operator. For a DP model, five functions are mandatory, which return the following:

1. Initial state.
2. Base state.
3. Set of applicable transitions, given a state.
4. Successor state, given a state and a transition.
5. Transition weight, given a state and a transition. The objective value of a solution is the sum of the weights.

In addition, a function for a merge operator is required. A dual bound function and state dominance are optionally defined as first-order functions. Unlike ddo, CODD does not assume that the number of transitions in a solution is fixed and explicitly defines a single base state.

3 Rust Programmable Interface for DIDP

We propose the Rust Programmable Interface for DIDP (RPID), a DIDP implementation using a strategy similar to DD-based solvers: a DP model is formulated as a Rust program using traits. To solve a model, we implement heuristic search solvers, following didp-rs.

Listing 1 The DP model for TSPTW in RPID.

```
struct Tsptw { a: Vec<i32>, b: Vec<i32>, c: Vec<Vec<i32>> }
struct S { u: FixedBitSet, i: usize, t: i32 }

impl Dp for Tsptw {
    type State = S;
    type CostType = i32;

    fn get_target(&self) -> Self::State {
        let mut u = FixedBitSet::with_capacity(self.a.len());
        u.insert_range(1..);
        S { u, i: 0, t: 0 }
    }
    fn get_successors(&self, s: &Self::State)
        -> impl IntoIterator<Item = (Self::State, Self::CostType, usize)> {
        s.u.ones().filter_map(|j| {
            if s.t + self.c[s.i][j] > self.b[j] { return None; }
            let mut u = s.u.clone();
            u.remove(j);
            let t = cmp::max(s.t + self.c[s.i][j], self.a[j]);
            if u.ones().any(|k| t + self.c[j][k] > self.b[k]) {
                None
            } else {
                Some((S { u, i: j, t }, self.c[s.i][j], j))
            }
        })
    }
    fn get_base_cost(&self, s: &Self::State) -> Option<Self::CostType> {
        if s.u.is_clear() { Some(self.c[s.i][0]) } else { None }
    }
}
```

We show the DP model for TSPTW formulated with RPID in Listing 1. First, Tsptw, a struct containing data of a problem instance (a and b for the time windows and c for the travel time), is defined. Then, S is defined to represent a state of the DP model, where field u is the set of unvisited customers U, i is the current location i, and t is the current time t. Trait Dp is implemented for Tsptw. Associated type State is used for a state, and CostType is used for the objective value. Dp has three required methods:

1. get_target returns the state corresponding to the original problem, which we call the target state following the convention in DIDP.
2. get_successors returns the successor states, transition weights, and transitions labels, given a state.

3. get_base_cost returns the value of a given state if it is a base state and None otherwise.
The return type of get_successors is impl IntoIterator, which means that any data type implementing the IntoIterator trait can be used. The easiest way is to return a dynamic array in the Rust standard library (Vec), which implements IntoIterator. In our example, filter_map is used to avoid allocating memory with Vec. Inside filter_map, each customer j included in s.u is examined, and the transition is labeled with j, representing visiting j. The successor states are filtered by the constraint in Equation (3), assuming $c_{ij} = c^*_{ij}$ for simplicity.

By default, the objective value of a solution is the sum of transition weights and the value of the base state. The binary operator to combine the transition weights ($\circ$ in Equation (1)) can be changed by overriding method combine_cost_weights. Similarly, minimization is assumed by default, and maximization can be selected by overriding method get_optimization_mode.

Listing 2 State dominance in RPID.

```
impl Dominance for Tsptw {
    type State = S;
    type Key = (FixedBitSet, usize);

    fn get_key(&self, s: &Self::State) -> Self::Key {
        (s.u.clone(), s.i)
    }
    fn compare(&self, s1: &Self::State, s2: &Self::State) -> Option<Ordering> {
        Some(s2.t.cmp(&s1.t))
    }
}
```

Listing 3 Dual bound function in RPID.

```
impl Bound for Tsptw {
    type State = S;
    type CostType = i32;

    fn get_dual_bound(&self, s: &Self::State) -> Option<Self::CostType> {
        let sum_to = s.u.ones().map(|j| self.c_to[j]).sum::<i32>();
        let sum_from = s.u.ones().map(|j| self.c_from[j]).sum::<i32>();
        Some(cmp::max(sum_to + self.c_to[0], sum_from + self.c_from[s.i]))
    }
}
```

Implementing the Dominance trait defines a sufficient condition for state dominance (Listing 2). Our design is inspired by ddo. The associated type Key is the type of the key, the part of a state that must be the same if one state dominates another. Given two states, if their keys extracted by the required method get_key are the same, then dominance is checked by method compare. The return value is None or Some(Ordering), an enumerated type in the Rust standard library, with value of Equal, Less, or Greater. Given two states, Equal is returned if they dominate each other, Less if the first state is known to be dominated by the second and not vice versa, Greater if the first is known to dominate the second and not vice versa, and None if no dominance is detected. In our example, Greater is returned if the first state has smaller t.

In practice, Dominance is required by all solvers currently implemented. However, a user does not always need to come up with sufficient conditions to check state dominance. Implementing compare is optional, and it returns Equal by default, meaning two states have the same value if their keys are the same. Thus, the easiest way to implement Dominance is to let get_key return the state itself. In such a case, a solver uses Dominance just to detect duplicate states to avoid redundant work. A user can also use a customized implementation of get_key without overriding compare when using only parts of the state data structure is sufficient; a state may cache information resulting from expensive computation.

A dual bound function is defined by trait Bound, which has only one required method, get_dual_bound, returning the value of a dual bound function given a state. It returns None if it turns out that the state does not lead to a solution, i.e., $V(S) = \infty$ in minimization. In our example, assuming that fields c_to and c_from are added to Tsptw, corresponding to c^{to} and c^{from} in Inequality (5), the bound is computed as in Listing 3.

3.1 Discussion

We discuss the design of RPID. First, we argue the advantage of RPID over didp-rs. Then, we highlight the differences between RPID and existing DD-based solvers.

3.1.1 RPID vs. didp-rs

In didp-rs, expression trees are used for modeling. As of dypdl 0.8.0, expressions are designed for declarative definitions and are somewhat limited; loops cannot be used, and thus, implementing complicated algorithmic procedures or data structures is difficult. In the TSPTW example, as a dual bound function, we could use the minimum spanning tree (MST) weight in a complete graph, where U is the set of nodes and c_{jk} is the weight of edge (j, k). We can further improve this bound by constructing a 1-tree [10]: since U does not include the current location i and the depot 0, the cheapest edge from i to a customer in U and the cheapest edge from a customer in U to the depot are added to the MST. However, computing the MST weight using expressions is difficult in didp-rs. In contrast, RPID, ddo, and CODD are more flexible since we can directly write algorithms in the programming languages in which they are implemented.

Another potential advantage of RPID over didp-rs is performance. Methods in RPID are directly compiled Rust and so running them is substantially faster than evaluating expressions using the intermediate interpreter in didp-rs.

Note, however, that RPID complements didp-rs rather than replacing it; didp-rs is preferred in some use cases. For example, didp-rs provides a Python interface, DIDPPy. We could provide a Python interface for RPID similar to Py-DDO by defining a Rust struct with a Python object as a member. However, it may not be as efficient as DIDPPy since we need to call Python functions during solving. In addition, with explicit expression trees, a DIDP solver can analyze and exploit particular structures of expressions, as done in Kuroiwa and Beck [15]. In contrast, when components of a DP model are implemented as Rust methods, they become black-boxes for a solver, and such analysis and exploitation are difficult.

We could also port the didp-rs interfaces to RPID by implementing structs and traits that parse expressions. While it is unclear that such a system would have any computational advantages over didp-rs, it would facilitate a workflow of rapid prototyping in the didp-rs interface followed by production implementation in the lower-level Rust. Such a unification is one direction for future work.

3.1.2 Traits vs. Functions

A design advantage of traits is their explicitness: required methods and their signatures are clear for a user from the trait definitions. The downside is that the struct implementing the trait may have many fields to provide all necessary information to each method, e.g., a, b, c, c_to, and c_from in Tsptw. In contrast, CODD can avoid defining such a struct since each first-order function implemented as a C++ lambda function captures necessary information in addition to its arguments.

3.1.3 Successor Generation

Ddo and CODD use separate functions to identify applicable transitions, apply each transition to generate the successor state, and compute the transition weight. In contrast, RPID performs all of them at once in get_successors. This design choice was made for two reasons.

First, successor generation becomes more explicit. When reading a model, a user does not need to refer to multiple methods to understand how successor states are generated. While decomposing the successor generation function into pieces may be useful when the function is complicated, a user can do that in their own way, not forced by the trait definition.

Second, successor generation becomes more efficient when the same information is required in multiple places. Since it is not the case with TSPTW, we introduce single machine total weighted tardiness ($1||\sum w_i T_i$) as a motivating example. In this problem, a set of jobs N is processed on a single machine, and each job $j \in N$ has the processing time p_j, the due date d_j, and the weight w_j. The optimal solution is a sequence of the jobs that minimizes the total weighted tardiness $\sum_{j \in N} w_j \max\{C_j - d_j, 0\}$, where C_j is the completion time of job j. In our DP model, we represent a state by a single state variable S, representing the set of processed jobs, and process one job $j \in N \setminus S$ in each decision. We present the DP model in Equation (6) and its implementation with RPID in Listing 4.[3]

$$V(S) = \begin{cases} 0 & \text{if } S = N \\ \min\limits_{j \in N \setminus S} w_j \max\left\{\left(\sum_{k \in S} p_k\right) + p_j - d_j, 0\right\} + V(S \cup \{j\}) & \text{if } S \neq N. \end{cases} \tag{6}$$

Listing 4 The DP model for $1||\sum w_i T_i$ in RPID.

```
struct Wt { p: Vec<i32>, d: Vec<i32>, w: Vec<i32> }

impl Dp for Wt {
    type State = FixedBitSet;
    type CostType = i32;

    fn get_target(&self) -> Self::State {
        FixedBitSet::with_capacity(self.p.len())
    }
    fn get_successors(&self, s: &Self::State)
        -> impl IntoIterator<Item = (Self::State, Self::CostType, usize)> {
        let t = s.ones().map(|k| self.p[k]).sum::<i32>();
        s.zeroes().map(move |j| {
            let mut next_s = s.clone();
            next_s.insert(j);
            let tardiness = cmp::max(t + self.p[j] - self.d[j], 0);
            (next_s, self.w[j] * tardiness, j)
        })
    }
    fn get_base_cost(&self, s: &Self::State) -> Option<Self::CostType> {
        if s.is_full() { Some(0) } else { None }
    }
}
```

[3] Our actual DP model used in the experimental evaluation exploits precedence between jobs extracted by preprocessing following Kuroiwa and Beck [16, 17].

In this model, the time job j starts is $\sum_{k \in S} p_k$, which requires $O(|N|)$ time to compute. If we compute it for each j, we need $O(|N|^2)$ computation to generate all successors. To avoid such computation, we could introduce a redundant state variable t that is increased by p_j when j is processed, but this approach increases the amount of memory used for each state. In our implementation with RPID, we compute $\sum_{k \in S} p_k$ only once, save it as a Rust variable `t`, and use it for each `j`. We use similar approaches in the DP models for the minimization of open stacks problem and talent scheduling evaluated in Section 4.

A disadvantage of this design is that a solver has less flexibility in generating successor states: it cannot compute applicable transitions, the successor states, and the transition weights separately, potentially limiting the design of a solving algorithm. In the current implementation, our heuristic search solvers are not affected by this restriction.

4 Empirical Evaluation

We compare the performance of RPID against existing DIDP and DD-based solvers. Building on previous DIDP solvers [14, 16, 17], we implement cost-algebraic A* [6] and complete anytime beam search (CABS) [21]; A* is a fundamental algorithm in heuristic search, and CABS performs the best in the existing DIDP solvers due to its memory efficiency. We publish the source code for RPID[4] and DP models.[5] We use Rust 1.76.0 for all solvers implemented in Rust. For each problem instance, we use a 30-minute time limit, an 8GB memory limit, and a single core of Intel Xeon Gold 6418, run with GNU parallel [19].

4.1 RPID vs. didp-rs

With didp-rs, previous work [17] used DP models for eleven problem classes: TSPTW, the capacitated vehicle routing problem (CVRP), the multi-commodity pickup and delivery traveling salesperson problem (m-PDTSP), the orienteering problem with time windows (OPTW), the multi-dimensional knapsack problem (MDKP), bin packing, the simple assembly line balancing problem (SALBP-1), $1||\sum w_i T_i$, talent scheduling, the minimization of open stacks problem (MOSP), and graph-clear. DIDP solvers in didp-rs outperformed commercial CP and mixed-integer programming solvers in TSPTW, m-PDTSP, OPTW, SALBP-1, $1||\sum w_i T_i$, talent scheduling, MOSP, and graph-clear. The original models are available in YAML-DyPDL and DIDPPy in a public repository,[6] and we reimplement them in RPID. As a baseline, we also reimplement these models in Rust using didp-rs 0.8.0: we define the DP models by writing expressions with the dypdl library and solve them using the dypdl-heuristic-search library in Rust. We call this baseline "didp-rs."

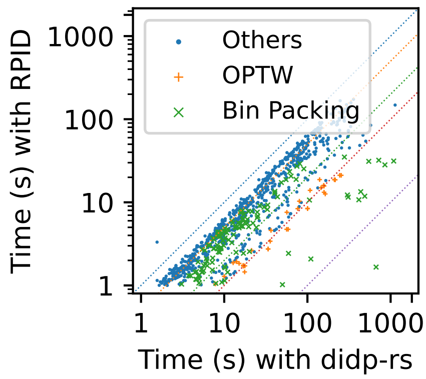
We confirmed that the numbers of expanded states are the same in RPID and didp-rs for the same solver and the same problem instance. Thus, we compare the number of optimally solved instances and the average time to solve an instance in each problem class. As shown in Table 1, RPID dominates didp-rs: solving more instances than didp-rs in six problem classes with A* and eight with CABS and never failing to solve an instance solved by didp-rs. On average, RPID is faster than didp-rs in all problem classes and reduces the solving time by at least half in five classes with A* and eight with CABS. We also show the number of solved instances against time for TSPTW, m-PDTSP, and OPTW in Figure 2 and for other problem classes in Appendix B.

---

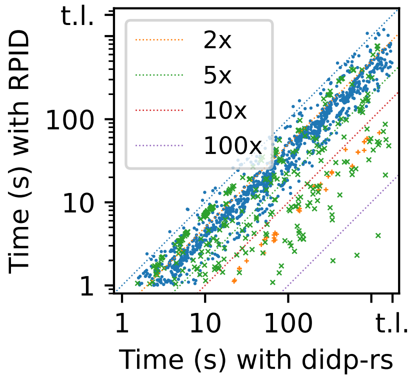
[4] https://github.com/domain-independent-dp/rpid/releases/tag/v0.1.0
[5] https://github.com/Kurorororo/didp-rust-models
[6] https://github.com/Kurorororo/didp-models

Table 1 RPID vs. didp-rs with the same DP models. '#opt' is the number of optimally solved instances, and "time" is the time in seconds to solve an instance optimally, averaged over instances where didp-rs takes at least 1 second. The higher value of '#opt' is in bold, and "time" by RPID is in bold if less than half of that of didp-rs.

	A*				CABS			
	didp-rs		RPID		didp-rs		RPID	
	#opt	time	#opt	time	#opt	time	#opt	time
TSPTW (340)	257	64	257	**17**	259	196	**267**	**44**
CVRP (207)	6	54	6	29	6	118	6	**46**
m-PDTSP (1178)	953	24	953	14	1034	149	**1049**	**66**
OPTW (144)	64	54	64	**5**	64	212	**85**	**13**
MDKP (276)	4	3	4	2	5	63	5	33
Bin Packing (1615)	922	42	**939**	**4**	1168	140	**1230**	**33**
SALBP-1 (2100)	1657	31	**1667**	**8**	1801	195	**1821**	**71**
$1\|\|\sum w_i T_i$ (375)	270	31	**277**	18	288	173	**299**	**66**
Talent Scheduling (1000)	207	62	**225**	**26**	235	277	**257**	**88**
MOSP (570)	483	8	**487**	4	527	172	527	142
Graph-Clear (135)	78	17	**80**	11	103	173	**104**	135



(a) A*.



(b) CABS.

Figure 1 Time in seconds to solve each instance optimally (only instances with at least 1 second).

We present an instance-wise comparison of the solving time by RPID and didp-rs in Figure 1. In the majority of instances, RPID achieves a speedup of more than two times. For A*, RPID is slower in one SALBP-1 instance, where we observe that RPID proves optimality faster than didp-rs but takes more time for termination, including freeing allocated memory. For CABS, RPID is slower than didp-rs in 19 MOSP instances. This performance degradation seems to be due to the implementation of state variables. The DP model for MOSP has two set state variables, and each transition performs set operations on them. Both our RPID model and didp-rs use the same library, fixedbitset, to represent set state variables. However, while didp-rs 0.8.0 uses fixedbitset 0.4.2, our model uses 0.5.7 (the latest version as of writing). We observe that using 0.4.2 with our RPID model results in better performance than didp-rs. We suspect that the overhead of didp-rs is relatively low in MOSP, sometimes outweighed by the difference in the set variable implementation.

Table 1 and Figure 1 present large performance gains in bin packing and OPTW. In particular, RPID achieves more than 100 times speedup in some bin packing instances. In bin packing, we minimize the number of bins to pack a set of items N, where a bin has the

Table 2 Comparison of models with the new dual bound functions vs. the models by Kuroiwa and Beck [17] using RPID. We use the 1-tree bound for TSPTW, CVRP, and m-PDTSP and the Dantzig bound for OPTW and MDKP. '#opt' is the number of optimally solved instances, "time" is the time in seconds to solve an instance optimally, and '#expanded' is the number of expansions before the last f-layer. "time" and '#expanded' are averaged over instances where the model by Kuroiwa and Beck [17] takes at least 1 second. Better values are bolded.

	Kuroiwa and Beck [17]			1-Tree/Dantzig		
A*	#opt	time	#expanded	#opt	time	#expanded
TSPTW (340)	257	**25**	3,681,570	257	89	**3,344,561**
CVRP (207)	6	29	3,577,960	**8**	**5**	**390,277**
m-PDTSP (1178)	953	19	2,516,593	**1075**	**5**	**356,029**
OPTW (144)	64	8	2,897,171	**71**	**4**	**1,220,461**
MDKP (276)	4	2	735,507	**6**	**0**	**342**
CABS	#opt	time	#expanded	#opt	time	#expanded
TSPTW (340)	**267**	**161**	16,855,270	247	287	**4,420,454**
CVRP (207)	6	55	9,579,821	**8**	**19**	**989,025**
m-PDTSP (1178)	1049	129	18,352,633	**1074**	**34**	**820,919**
OPTW (144)	85	340	78,952,253	**92**	**97**	**20,692,072**
MDKP (276)	5	33	16,356,224	**6**	**0**	**29,756**

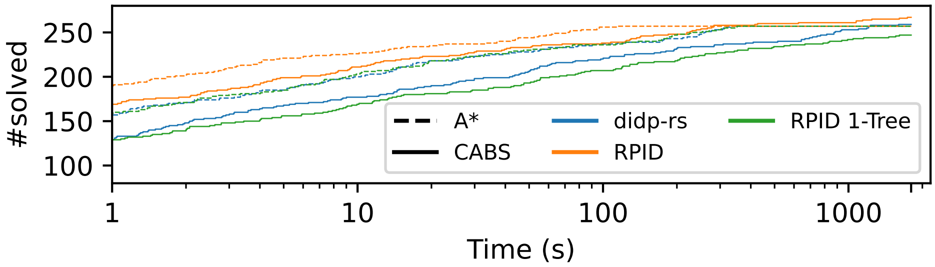
capacity c, and each item $i \in N$ has the weight w_i. In the DP model, a state is represented by the remaining capacity of the current bin r and the set of unpacked items U. Each transition packs an item $i \in U$ with $w_i \leq r$ and reduces r by w_i. If no such item exists, only one transition is applicable, which opens a new bin and packs an arbitrary item i, updating r to $c - w_i$. In didp-rs, two transitions are defined for each item i, one to pack it in the current bin and another to pack it in a new bin. The latter has a precondition $j \notin U \vee w_j > r$ for each $j \in N$ to confirm that no item can be packed in the current bin, requiring $O(|N|)$ time to check in the worst case. Since there are $O(|N|)$ such transitions, identifying applicable transitions requires $O(|N|^2)$ time. In RPID, we can avoid such computation by checking $\forall j \in U, w_j > r$ only once in the successor generator function, as presented in Appendix A.

For OPTW, the restrictions of the DyPDL syntax result in a large expression tree with less efficient execution than what can be done with Rust code. In the model, the dual bound function computes $\sum_{j \in U : \phi_j} c_j$, where $U \subseteq N$ is a set state variable, ϕ_j is a Boolean condition on j, and c_j is a constant depending on j. To represent this computation, the current implementation takes the sum of expressions representing "c_j if $j \in U \wedge \phi_j$ and 0 otherwise" for all $j \in N$, resulting in an expression tree with depth $O(|N|)$.

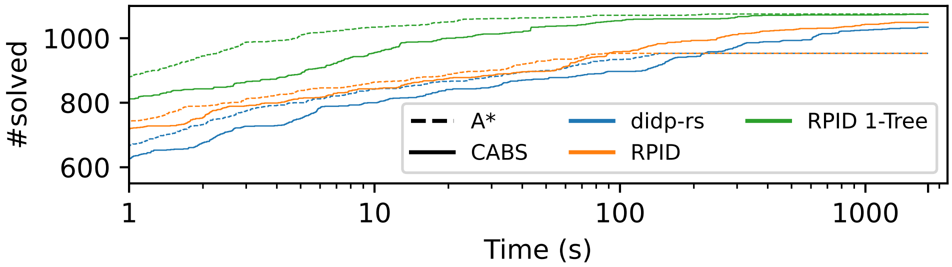
4.2 Comparison of Dual Bound Functions

As we discussed in Section 3.1.1, a dual bound function based on the 1-tree weight can be used for the DP model of TSPTW. The DP models for CVRP and m-PDTSP are similar to that of TSPTW, and dual bound functions similar to Inequality (5) are used, so they can also be replaced with the 1-tree bound. We implement such DP models in RPID, using Kruskal's algorithm to compute the MST weight, which requires sorting edges in the ascending order of the weights. In our implementation, sorting is performed in preprocessing, and Kruskal's algorithm ignores edges connected to visited customers in a given state.

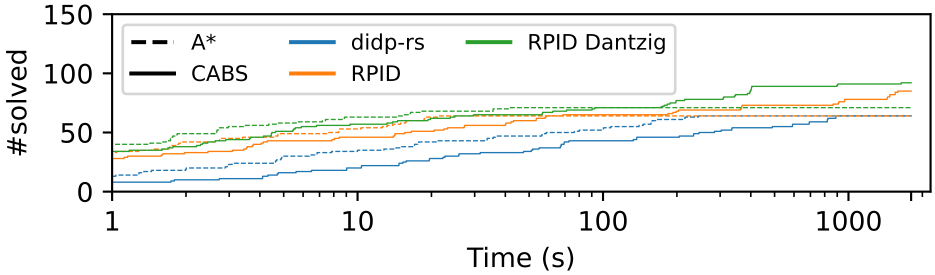
Since OPTW and MDKP can be considered generalizations of 0-1 knapsack, the Dantzig bound [4] can be used as a dual bound function. Due to restrictions of expressions in didp-rs, Kuroiwa and Beck [17] approximated the Dantzig bound in their DP models. In RPID, we



(a) TSPTW.



(b) m-PDTSP.



(c) OPTW.

Figure 2 Time in seconds vs. the number of optimally solved instances.

implement DP models using the Dantzig bound as a dual bound function. We compare the new DP models with the original models (the ones used in Section 4.1) using A* and CABS in RPID, measuring the number of state expansions by each heuristic search algorithm to solve an instance optimally. Our heuristic search algorithms maintain the global dual bound, a lower/upper bound of the optimal objective value in minimization/maximization. We subtract the number of expansions after the global dual bound matches the optimal objective value from the total number of expansions. This metric is called "expansions before the last f-layer" and is conventionally used to compare different heuristic functions for A*.

As shown in Table 2, our dual bound functions reduce the number of expansions in all problem classes. In CVRP, m-PDTSP, OPTW, and MDKP, the number of optimally solved instances is increased, and the time to solve an instance is reduced by a factor of two to six. We also present the number of solved instances against time for TSPTW, m-PDTSP, and OPTW in Figure 2. The result shows that the flexibility of the RPID modeling can contribute to significant performance improvement. However, in TSPTW, the time to solve an instance is increased, and the number of instances solved optimally by CABS is decreased.

Table 3 Comparison of the RPID model implementations for $1||\sum w_i T_i$ where $\sum_{k \in S} p_k$ is computed once for all successors (Original), separately computed for each successor (Separate), and cached as a state variable (StateCache). '#opt' is the number of optimally solved instances, and "time" is the time in seconds to solve an instance optimally, averaged over co-solved instances where Original takes at least 1 seconds. The best value is bolded.

	Original		Separate		StateCache	
	#opt	time	#opt	time	#opt	time
A*	**277**	**27**	**277**	28	274	**27**
CABS	**299**	**139**	295	154	298	144

One potential reason is the quadratic computational complexity of Kruskal's algorithm on a complete graph compared to the linear complexity of the dual bound function in Inequality (5). While the expensive dual bound function pays off in CVRP and m-PDTSP, it does not in TSPTW, possibly because many states are already pruned by Equation (3).

4.3 Impact of the Successor Generation Interface

As discussed in Section 3.1.3, the single successor generation function of RPID can be beneficial when the same information is required by multiple transitions. We evaluate the impact of this interface design using the DP model for $1||\sum w_i T_i$, presented in Equation (6). In our original implementation presented in Listing 4 and used in Section 4.1, the total processing time of already processed jobs, $\sum_{k \in S} p_k$, is computed once for all successor states. We consider two different implementations, "Separate", where $\sum_{k \in S} p_k$ is separately computed each time a successor state is generated, and "StateCache", where $\sum_{k \in S} p_k$ is stored as a state variable and increased by p_j when j is added to S.

We compare the three implementations in Table 3. With A*, Separate slightly increases the average time to solve an instance, and StateCache reaches the memory limit in three instances solved by Original and Separate, possibly due to increased memory usage per state. With CABS, both Separate and StateCache increase average solving time and solve fewer instances than Original due to the time limit. These results confirm that our interface design has a positive impact on performance.

4.4 RPID vs. Ddo and CODD

We compare RPID, didp-rs, ddo, and CODD using the problem classes with which previous work compared CODD with didp-rs and ddo [18]: 0-1 knapsack, Golomb ruler, and the maximum independent set problem (MISP). For 0-1 knapsack and MISP, instances are retrieved from the CODD repository.[7] For Golomb ruler, an instance is uniquely determined from a parameter n. While previous work compared heuristic search solvers and DD-based solvers [17, 18], the interface designs of the solvers are different; didp-rs uses expression trees and DD-based solvers use functions in the programming language in which they are implemented. With RPID, we conduct the first empirical evaluation comparing heuristic search solvers and DD-based solvers without such differences.

[7] https://github.com/ldmbouge/CODD/tree/main/data

Table 4 RPID vs. didp-rs, ddo, and CODD in time in seconds to optimally solve a 0-1 knapsack instance. "t.o." indicates time out, and "m.o." indicates memory out. The smallest value is in bold.

	Dantzig					No Dantzig			
	Ddo	CODD		RPID		didp-rs		RPID	
Instance	width=4	width	time	A*	CABS	A*	CABS	A*	CABS
PI:1 2000	0.37	64	0.76	**0.12**	0.25	2.72	2.96	1.13	0.92
PI:1 5000	0.47	64	2.73	**0.13**	0.28	175.14	106.83	41.58	32.77
PI:1 10000	0.71	64	m.o.	**0.16**	0.28	t.o.	1227.55	566.74	321.07
PI:2 2000	0.16	64	3.70	**0.02**	0.04	1.05	1.73	0.42	0.61
PI:2 5000	0.44	64	m.o.	**0.15**	0.29	52.65	64.20	13.97	21.09
PI:2 10000	0.76	64	m.o.	**0.16**	0.27	835.28	870.12	220.60	222.25
PI:3 1000	0.26	1024	**0.08**	0.14	0.27	0.90	1.53	0.50	0.48
PI:3 2000	3.23	2048	3.47	**0.34**	1.38	10.92	14.27	3.40	4.89
PI:3 5000	4.87	4096	6.29	**0.74**	4.83	243.29	297.93	66.10	81.09
PI:3 10000	4.32	8192	m.o.	**1.26**	8.73	t.o.	t.o.	651.70	656.50

We use ddo 2.0.0 and the latest model implementations available in its repository at the time of writing.[8] Ddo requires a parameter called a width as input, and each model code defines a default width. In 0-1 knapsack, we use a width of 4 as it performs better than the default width of 2. In the other two problem classes, we use the default parameters, 10 for Golomb ruler and the width automatically decided based on a problem instance in MISP.

For CODD, from models in its repository,[9] we use knapsack2 for 0-1 knapsack, gruler_midlb for Golomb ruler, and misp5 for MISP,[10] compiled with GCC 13.2.0. Similar to ddo, CODD also requires a width parameter as input. Following the previous work using the same instances [18], we use 64 for Golomb ruler and 128 for MISP. In 0-1 knapsack, we use the best width for each instance reported in the previous work [18].

For didp-rs and RPID, we implement models following the CODD models. For 0-1 knapsack, ddo, CODD, and RPID models use the Dantzig bound as a dual bound function. As mentioned earlier, the Dantzig bound is difficult to use with didp-rs, so we define a dual bound function similar to those used in the DP models for MDKP and OPTW, and we also evaluate a RPID model with such a dual bound function (No Dantzig).

Tables 4–6 present the results, omitting instances solved within 1 second by all solvers. There is no single winner: RPID with A* shows a clear advantage in 0-1 knapsack, CODD is the best in Golomb ruler, and ddo is almost always the best in MISP. While previous work reported that didp-rs with CABS fails to solve four 0-1 knapsack instances in their evaluation [18], it solves all but one in our evaluation. We suspect that previous work did not use any dual bound function with DIDP, following the model in the didp-rs repository.[11] In our evaluation, CODD reaches the 8GB memory limit in four instances of 0-1 knapsack. Given a larger memory limit, CODD solves all such instances in at most a few tens of seconds, but it is slower than DDO and RPID with the Dantzig bound. Further analysis of the performance difference between DIDP and DD-based solvers is left for future work.

[8] https://github.com/xgillard/ddo/tree/3b39798874b66ac965a0ce915c6f21f562ebaa6e/ddo/examples

[9] https://github.com/ldmbouge/CODD/tree/main/examples

[10] They seem to be the best models according to authors' description and our preliminary experiments.

[11] https://github.com/domain-independent-dp/didp-rs/blob/main/didppy/examples/knapsack.ipynb

Table 5 RPID vs. didp-rs, ddo, and CODD in time in seconds to optimally solve a Golomb ruler instance. "t.o." indicates time out, and "m.o." indicates memory out. The smallest value is in bold.

	Ddo	CODD	didp-rs		RPID	
n	width=10	width=64	A*	CABS	A*	CABS
8	0.71	**0.04**	2.61	5.73	1.88	1.19
9	6.89	**0.20**	26.25	51.49	14.02	13.85
10	50.55	**0.68**	m.o.	532.55	m.o.	143.84
11	m.o.	**10.51**	m.o.	t.o.	m.o.	m.o.
12	m.o.	**55.56**	m.o.	t.o.	m.o.	t.o.
13	m.o.	**1318.98**	m.o.	t.o.	m.o.	m.o.
14	m.o.	t.o.	m.o.	t.o.	m.o.	t.o.

Table 6 RPID vs. didp-rs, ddo, and CODD in time in seconds to optimally solve an MISP instance. "t.o." indicates time out, and "m.o." indicates memory out. The smallest value is in bold.

	Ddo	CODD	didp-rs		RPID	
Instance		width=128	A*	CABS	A*	CABS
johnson8-4-4	**0.17**	0.24	0.81	1.04	0.70	0.58
johnson16-2-4	2.64	1.43	1.04	1.11	0.92	**0.62**
keller4	**5.47**	29.63	27.55	65.46	15.62	39.02
hamming6-2	**0.17**	0.28	2.84	14.22	1.56	9.07
hamming8-2	**0.30**	63.69	m.o.	t.o.	m.o.	t.o.
hamming8-4	**29.65**	36.81	m.o.	m.o.	m.o.	m.o.
hamming10-2	**10.01**	1520.08	m.o.	t.o.	m.o.	t.o.
brock200-1	**403.20**	t.o.	m.o.	t.o.	m.o.	m.o.
brock200-2	**1.10**	3.68	5.68	15.09	3.41	8.72
brock200-3	**7.88**	22.21	m.o.	131.98	m.o.	80.07
brock200-4	**22.67**	102.52	m.o.	797.01	m.o.	455.68
p_hat300-1	**0.49**	1.25	2.09	1.53	1.42	1.01
p_hat300-2	**19.82**	317.83	m.o.	t.o.	m.o.	m.o.

5 Conclusion

We propose new software for domain-independent dynamic programming (DIDP): the Rust Programmable Interface for DIDP (RPID). We use traits in Rust for modeling, following an existing decision diagram-based (DD-based) solver. RPID is novel in that the successor generation is defined in a single function, motivated by readability and efficiency. As DIDP software, RPID enables flexible modeling and fast execution by using Rust functions. Our experiment shows that given the same models, RPID is faster than the existing DIDP implementation, didp-rs, in most cases, and further performance improvement is achieved with the better models facilitated by the flexibility of RPID. We also demonstrate that the relative performance of RPID and existing DD-based solvers changes by problem classes.

As we discussed, didp-rs is preferred to RPID in some use cases that are particularly related to model development and analysis. To close the performance gap between didp-rs and RPID, improving the flexibility and efficiency of expressions in didp-rs is an important direction. For example, we may want to implement specialized expressions for particular computations that are commonly used in DIDP models.

References

1 Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957.

2 David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and J. N. Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016. doi:10.1287/ijoc.2015.0648.

3 Vianney Coppé, Xavier Gillard, and Pierre Schaus. Modeling and exploiting dominance rules for discrete optimization with decision diagrams. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research – 21st International Conference, CPAIOR 2024*. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-60597-0_15.

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

5 Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995. doi:10.1287/opre.43.2.367.

6 Stefan Edelkamp, Shahid Jabbar, and Alberto Lluch Lafuente. Cost-algebraic heuristic search. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, pages 1362–1367. AAAI Press, 2005. URL: http://www.aaai.org/Library/AAAI/2005/aaai05-216.php.

7 Xavier Gillard, Vianney Coppé, Pierre Schaus, and André Augusto Cire. Improving the filtering of branch-and-bound MDD solver. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research – 18th International Conference, CPAIOR 2021*, pages 231–247. Springer International Publishing, 2021. doi:10.1007/978-3-030-78230-6_15.

8 Xavier Gillard, Pierre Schaus, and Vianney Coppé. Ddo, a generic and efficient framework for MDD-based optimization. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 5243–5245. International Joint Conferences on Artificial Intelligence Organization, 2020. Demos. doi:10.24963/ijcai.2020/757.

9 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi:10.1109/TSSC.1968.300136.

10 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970. doi:10.1287/opre.18.6.1138.

11 John N. Hooker. Decision diagrams and dynamic programming. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – 10th International Conference, CPAIOR 2013*, pages 94–110. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38171-3_7.

12 Ryo Kuroiwa. DIDP Rust Models. Software, swhId: swh:1:dir:810758e7707cef89647835b7e209d2c94e7bdfd2 (visited on 2025-07-28). URL: https://github.com/Kuroroororo/didp-rust-models, doi:10.4230/artifacts.24206.

13 Ryo Kuroiwa and J. Christopher Beck. RPID. Software, version 0.1.0., swhId: swh:1:dir:9f071b7f5f3daf914c19752ae76692b3a8d1b108 (visited on 2025-07-28). URL: https://github.com/domain-independent-dp/rpid, doi:10.4230/artifacts.24205.

14 Ryo Kuroiwa and J. Christopher Beck. Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 236–244. AAAI Press, 2023. doi:10.1609/icaps.v33i1.27200.

15 Ryo Kuroiwa and J. Christopher Beck. Large neighborhood beam search for domain-independent dynamic programming. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.23.

16 Ryo Kuroiwa and J. Christopher Beck. Solving domain-independent dynamic programming problems with anytime heuristic search. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 245–253. AAAI Press, 2023. doi:10.1609/icaps.v33i1.27201.

17 Ryo Kuroiwa and J. Christopher Beck. Domain-independent dynamic programming, 2025. arXiv:2401.13883. doi:10.48550/arXiv.2401.13883.

18 Laurent Michel and Willem-Jan van Hoeve. CODD: A decision diagram-based solver for combinatorial optimization. In *ECAI 2024 – 27th European Conference on Artificial Intelligence*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, pages 4240–4247. IOS Press, 2024. doi:10.3233/FAIA240997.

19 Ole Tange. GNU parallel – The command-line power tool. *;login: The USENIX Magazine*, 36:42–47, 2011. URL: https://www.usenix.org/publications/login/february-2011-volume-36-number-1/gnu-parallel-command-line-power-tool.

20 Mark Wallace. Languages versus packages for constraint problem solving. In *Principles and Practice of Constraint Programming – CP 2003*, pages 37–52. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-45193-8_3.

21 Weixiong Zhang. Complete anytime beam search. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pages 425–430. AAAI Press, 1998. URL: http://www.aaai.org/Library/AAAI/1998/aaai98-060.php.

A DP Model for Bin Packing

Listing 5 The DP model for bin packing in RPID.

```
struct BinPacking { c: i32, w: Vec<i32> }
struct S { u: FixedBitSet, r: i32 }

impl Dp for BinPacking {
    type State = S;
    type CostType = i32;

    fn get_target(&self) -> Self::State {
        let mut u = FixedBitSet::with_capacity(self.w.len());
        u.insert_range(..);
        S { u, r: 0 }
    }
    fn get_successors(&self, s: &Self::State)
        -> impl IntoIterator<Item = (Self::State, Self::CostType, usize)> {
        let candidates = s.u.ones().filter(|&i| s.r >= self.w[i])
            .collect::<Vec<_>>();
        if candidates.is_empty() {
            let i = s.u.ones().next().unwrap();
            let mut next_u = s.u.clone();
            next_u.remove(i);
            vec![(S { u: next_u, r: self.c - self.w[i] }, 1, i)]
        } else {
            candidates.into_iter().map(|i| {
                let mut next_u = s.u.clone();
                next_u.remove(i);
                (S { u: next_u, r: s.r - self.w[i]}, 0, i)
            }).collect()
        }
    }
    fn get_base_cost(&self, s: &S) -> Option<Self::CostType> {
        if s.u.is_clear() { Some(0) } else { None }
    }
}
```

In bin packing, a set of items N is given, and each item $i \in N$ has the weight w_i. The objective is to minimize the number of bins to pack all items, where each bin has the capacity c. In our DP model, as state variables, we use the set of unpacked items U and the remaining capacity r of the current bin. Each transition packs item $i \in U$ in a bin. If i fits in the current bin, i.e., $w_i \leq r$, we can pack it in the current bin, and r is decreased by w_i. If $w_i > r$, we need to open a new bin to pack i, and r becomes $c - w_i$ in such a case. Without loss of optimality, we open a new bin only when no item can be packed in the current bin. In addition, any item can be selected as the first item packed in the newly opened bin.[12] We show the DP model in Equation (7) and its implementation with RPID in Listing 5.

$$V(U, r) = \begin{cases} 0 & \text{if } U = \emptyset \\ 1 + V(U \setminus \{i\}, c - w_i) & \text{if } \forall j \in U, w_j > r \land \exists i \in U \\ \min\limits_{i \in U : w_i \leq r} V(U \setminus \{i\}, r - w_i) & \text{otherwise.} \end{cases} \tag{7}$$

B Number of Optimally Solved Instances Against Time

We present the number of optimally solved instances against time for CVRP, MDKP, bin packing, SALBP-1, $1||\sum w_i T_i$, talent scheduling, MOSP, and talent scheduling.

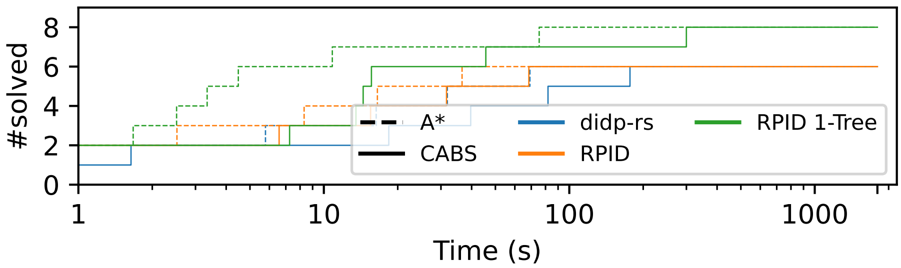


Figure 3 Time in seconds vs. the number of optimally solved instances for CVRP.

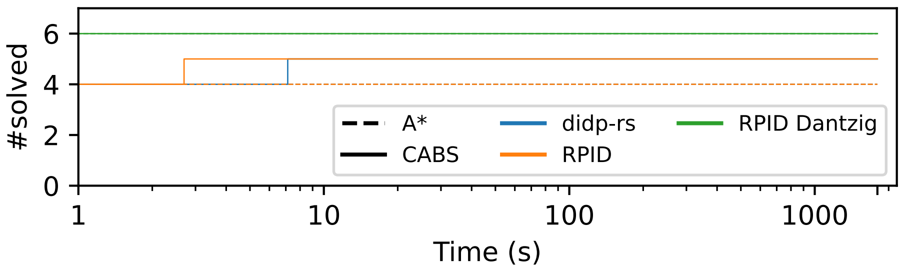


Figure 4 Time in seconds vs. the number of optimally solved instances for MDKP.

[12] In our DP model used in the experimental evaluation, we further break symmetries by enforcing that item i is packed in the i-th or earlier bin, following Kuroiwa and Beck [14, 17].

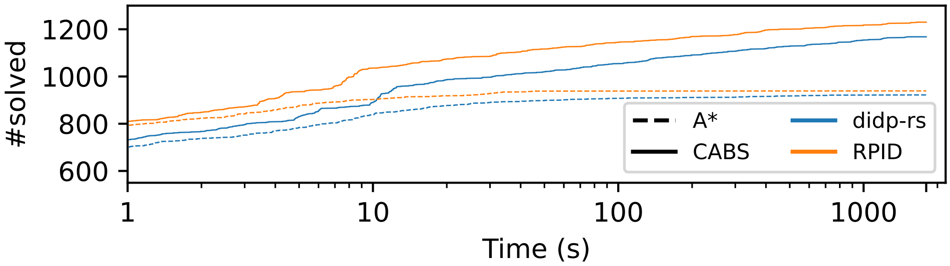


Figure 5 Time in seconds vs. the number of optimally solved instances for bin packing.

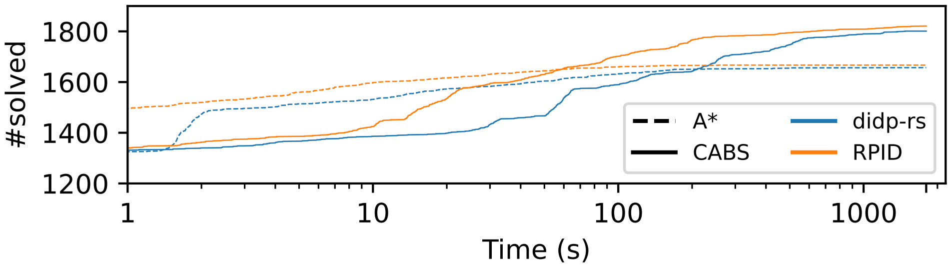


Figure 6 Time in seconds vs. the number of optimally solved instances for SALBP-1.

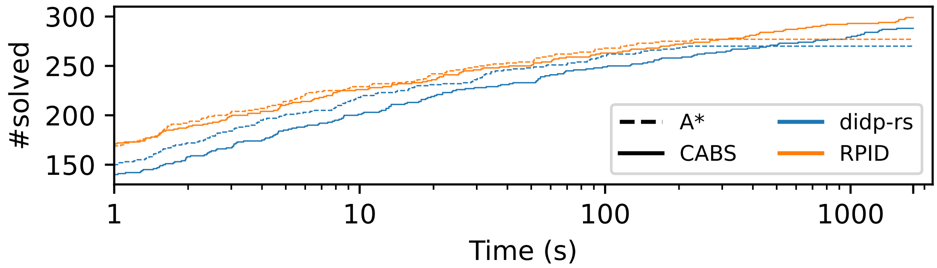


Figure 7 Time in seconds vs. the number of optimally solved instances for $1||\sum w_i T_i$.

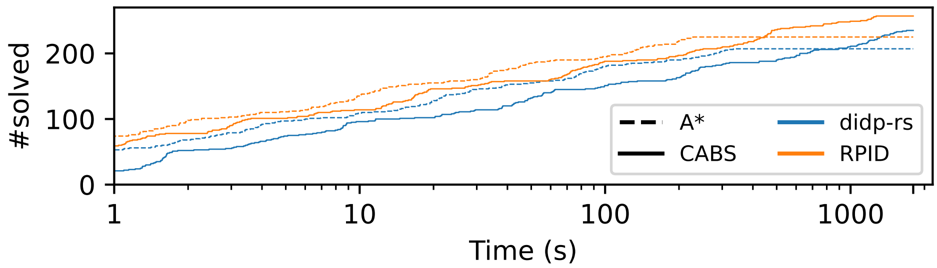


Figure 8 Time in seconds vs. the number of optimally solved instances for talent scheduling.

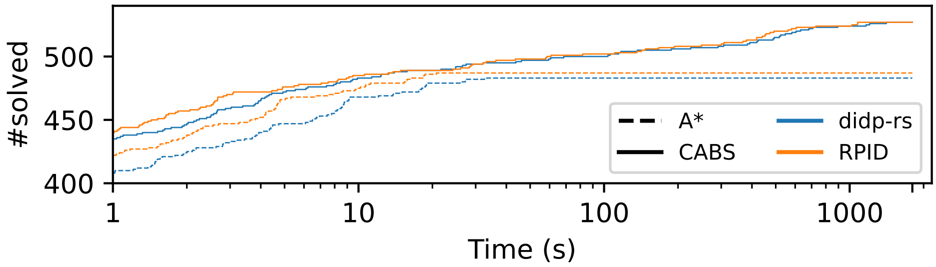


Figure 9 Time in seconds vs. the number of optimally solved instances for MOSP.

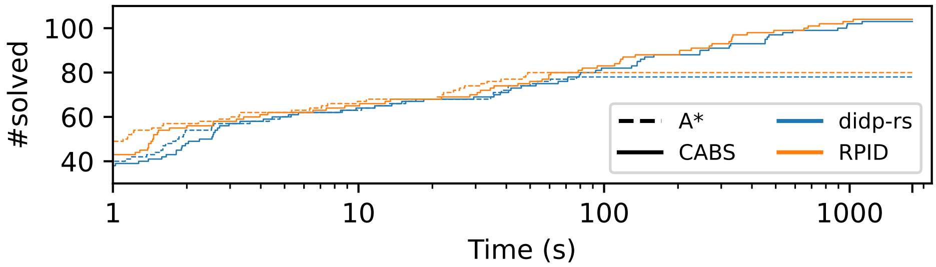


Figure 10 Time in seconds vs. the number of optimally solved instances for graph-clear.