Noname manuscript No.
(will be inserted by the editor)

DD-suite: A cross-platform package to build Decision Diagrams for optimization purposes

Antonia F. Blanco · Margarita Castro · Rodrigo Toro Icarte

Received: date / Accepted: date

Abstract Decision diagrams (DDs) have become a powerful tool for discrete optimization, supporting a wide range of algorithms that span cut-generation procedures, decomposition methods, and specialized branch-and-bound searches. Despite this growth, their adoption remains limited, partly because most existing DD code is tailored to a specific algorithm or application and is therefore hard to reuse. We introduce *DD-suite*, a cross-platform, open-source software package for building and manipulating DDs for discrete optimization. DD-suite is available in both Python and C++ through a shared modeling interface, and lets users construct exact, restricted, and relaxed DDs for any discrete optimization problem expressed in recursive form. The package implements the DD reduction procedure, shortest-path routines for obtaining primal and dual bounds, a visualization tool, and an extensive automated test suite. Furthermore, it includes extensive documentation, a support webpage, and ready-to-use examples for four combinatorial problems. Rather than a closed solver, DD-suite is designed as an extensible building block: users can add new construction mechanisms or run custom algorithms on top of the resulting diagram, as we illustrate with a DD-based cutting plane procedure embedded in a state-of-the-art mixed-integer programming solver. Our numerical experiments show that the C++ implementation is 5–6 times faster than the Python one while producing identical diagrams, and remains within a small constant factor of `ddo`, a specialized Rust framework, confirming that DD-suite combines an accessible, extensible codebase with competitive performance.

A. Blanco
Department of Industrial and Systems Engineering, Pontificia Universidad Católica de Chile

M. Castro
Department of Industrial and Systems Engineering, Pontificia Universidad Católica de Chile
E-mail: margarita.castro@uc.cl

R. Toro Icarte
Department of Computer Engineering, Pontificia Universidad Católica de Chile

Keywords Decision diagrams · Discrete optimization · Open-source software · Cutting planes · Cross-platform implementation

Mathematics Subject Classification (2020) 90C10 · 90C27 · 90C39 · 90-04

1 Introduction

In recent years, decision diagrams (DDs) have become a powerful tool for solving discrete optimization problems. This graphical structure was first proposed in the mid-1900s to represent circuits and Boolean functions [2,14,41]. Still, it was not until the early 2000s that researchers started to use it for optimization purposes in mathematical programming [5,6]. Since then, the number of applications of DDs for optimization purposes has rapidly grown, with many applications and the creation of novel algorithms based on such structures [12,18,33].

Specifically, DDs are graphical structures that can represent the feasibility set of most discrete optimization problems. Such representation is quite amenable to mathematical programming techniques, mainly because we can obtain the convex hull of a discrete problem using a network flow model over its corresponding DD [6]. Many authors have leveraged this characteristic to develop novel optimization algorithms, such as cut-generating procedures [17,25,53], Benders decomposition [30, 42,51], and branch-and-price [45,49] approaches, to name a few. Moreover, some researchers have created novel optimization approaches based on DDs, such as specialized branch-and-bound (B&B) algorithms entirely based on DDs [9,24,29,44], and the column elimination procedure [38,32]. These techniques have been employed in a wide range of applications that include classic combinatorial problems (e.g., independent set and graph coloring [11,32]), sequencing and vehicle routing problems [22,39,16], and healthcare applications [21,30,49], to name a few.

Despite the wide range of algorithms and applications based on DDs, their use is still quite limited, and only a few research groups have adopted such techniques. We believe this lack of diversity is mainly because DD-based algorithms require extensive coding to create and manipulate such DDs, which, until now, have been developed independently for each specific project. While several researchers have made their code publicly available, such code is usually tuned to their particular algorithm and application, which makes it very hard to modify or extend for other usages. To the best of our knowledge, two of the more general-purpose codes available for DDs in optimization are the DD-based B&B solvers (e.g., `ddo` [29] and CODD [44]), where users can easily include new problems that can be solved with their solver. However, these codes are mainly developed as solvers and are hard to adapt for other purposes (see Section 2 for further details and discussion).

Given the lack of software tools for DD-based optimization research, this work presents *DD-suite*, a cross-platform open-source code[1] to create and manipulate DDs for discrete optimization purposes. This software suite is available in Python and C++ and allows users to easily create three types of DDs (exact, restricted, or relaxed) for any discrete optimization problem they prefer. Our main goal is to provide an

[1] Available at `https://github.com/MargaritaCastro/dd-suite`.

intuitive, clean, and clear code that researchers can use to build DDs and create novel algorithms. To do so, DD-suite follows several good coding practices from software engineering [43], including clear on-code documentation and a support webpage with detailed tutorials and examples.

DD-suite is designed to be easily adapted to different usages, so researchers can employ DDs in any form they like. As an illustrative example, we provide a simple cutting plane implementation using DD-suite, in which we implemented existing DD-based cut-generation procedures (see Section 5.4 for further details). Moreover, our implementation allows users to test and explore new ways to construct DDs by changing key functions of their problem specifications or extending DD-suite to incorporate additional construction mechanisms. Furthermore, we include extensive automatic testing so users can safely work with DD-suite without fearing breaking the existing code. Lastly, DD-suite includes examples for four different combinatorial problems (i.e., knapsack, independent set, set cover, and sequencing), together with a second-order cone (SOC) knapsack example used in our cutting plane experiments, all with intuitive main files for users to play with.

Our numerical experiments confirm that this flexibility does not come at the expense of performance: the C++ and Python implementations produce identical diagrams, with C++ being 5–6 times faster and remaining within a small constant factor of `ddo`, a state-of-the-art Rust framework heavily optimized for speed. We further use these experiments to showcase the extensibility of DD-suite (i.e., by adding an alternative DD construction mechanism) and its research value (i.e., by implementing DD-based cutting planes that improve a state-of-the-art commercial solver).

Contributions. Overall, the main contributions of this work are:

- A clean, intuitive, and easily extended Python and C++ code to create and manipulate DDs for optimization purposes. Our implementation supports exact, relaxed, and restricted DDs and includes the DD reduction procedure, a shortest-path implementation to obtain optimality bounds, a DD visualization tool, and many other functionalities to get information about the DD and the supported algorithms.
- On-code documentation and a support webpage with tutorials and useful information on DD-suite usage.
- Extensive automatic testing to safely adapt and modify DD-suite for different usages without jeopardizing existing functionalities.
- A cutting plane example to illustrate how to use DD-suite for advanced optimization purposes and how to integrate it with mathematical programming solvers (e.g., Gurobi).
- Experiments comparing the efficiency of both code alternatives (Python vs. C++) and existing DD-based solvers.

Paper Structure. The rest of the paper is organized as follows. Section 2 reviews the publicly available DD software most closely related to DD-suite and positions our work with respect to them. Section 3 provides a background on DDs for discrete optimization, where we describe the main algorithms implemented in DD-suite. Section

4 details the basic features of the software (i.e., creating and manipulating DDs) and provides documentation and webpage details. Section 5 covers advanced usage aimed at developers, including the automatic testing framework, the cutting plane example, and how to extend DD-suite for new problems and construction mechanisms. Section 6 presents our numerical experiments, comparing the efficiency of the Python and C++ implementations against each other and against existing DD-based solvers, and illustrates the use of DD-suite for cutting planes and for prototyping new relaxation mechanisms. Finally, Section 7 concludes the paper and outlines future work directions.

2 Literature Review

We now review available tools that are closest in spirit to DD-suite, namely those that provide a somewhat general interface for modeling problems and building DDs. We focus on the two general-purpose DD-based B&B solvers, `ddo` [29] and CODD [44], and on the DD-compilation framework Haddock [26]. Table 1 summarizes the comparison, where a check mark (✓) denotes a supported feature and a dash (–) an unsupported one.

The `ddo` framework [29] is a generic and efficient Rust library for optimization based on decision diagrams. It implements the DD-based B&B algorithm of [9], where relaxed and restricted DDs provide the dual and primal bounds that drive the search, and it supports parallel B&B. Subsequent works have improved its performance through better filtering rules [28] and a caching mechanism for dominance and suboptimality detection [24]. The framework includes several examples (e.g., knapsack, independent set, MaxSAT, and MaxCut) and is heavily optimized for raw performance, which makes it a natural performance baseline for our experiments (see Section 6.2); on the other hand, its design centers on the B&B solver rather than on exposing the DDs for general use.

Table 1: DD-suite versus the most closely related software.

	`ddo`	CODD	Haddock	DD-suite
Language(s)	Rust	C++	C++ (MiniCP)	Python & C++
Primary role	B&B solver	B&B solver	DD compilation / CP	DD library
Problem specification	DP recursion	DP (lambdas)	Transition system	DP recursion
Exact / restricted / relaxed DDs	✓	✓	✓	✓
Primal & dual bounds	✓	✓	✓	✓
Parallel search	✓	–	–	–
Exposes DD for reuse	–	–	–	✓
Cutting planes / MP integration	–	–	–	✓
Cross-language interface	–	–	–	✓
Documentation & tutorials	README	Limited	Limited	Extensive
Automatic test suite	✓	–	–	✓
Open source	✓	✓	✓	✓

CODD [44] is a more recent C++ DD-based solver. Like `ddo`, it performs a DD-based B&B search, but it lets users specify the dynamic programming (DP) model very concisely through C++ lambda functions, sitting conceptually between a general DP solver, such as the domain-independent dynamic programming (DIDP) framework of [40], and a specialized DD solver such as `ddo`. As with `ddo`, CODD is primarily designed to be used as a solver, and its DD components are not meant to be repurposed for other DD-based techniques.

Haddock [26] is a language and architecture for DD compilation built on top of the MiniCP system (C++). Instead of an explicit DP recursion, problems are described through a labeled transition system, from which DDs are compiled. Haddock is mainly focused on DD propagation for constraint programming (CP), with later work adding the computation of optimization bounds [27]. Its scope is therefore different from ours: it targets DD-based propagation within a CP solver rather than providing a general-purpose toolkit to build and manipulate DDs for mathematical-programming-style algorithms.

Beyond these general frameworks, DDs have been embedded in several other optimization and CP tools. For instance, the peel-and-bound algorithm [50] builds relaxed DDs through *separation* (i.e., starting from a small, weak diagram and iteratively refining it by splitting nodes) to generate strong dual bounds, an alternative construction paradigm to the top-down procedure used by the solvers above and by DD-suite. Performance has also been pushed along a different axis: [52] accelerates the construction of DDs for DP models on GPUs, expanding and filtering states in parallel to obtain substantial speedups on hard sequencing problems. In the CP community, DDs are a standard data structure, and efficient libraries of DD operations (e.g., reduction, intersection, and union) have been developed to build CP models [46]. These tools, however, are again specialized to their target use (dual-bound computation, GPU-accelerated search, or CP propagation) rather than offering a general toolkit to build and manipulate DDs for arbitrary optimization algorithms.

A common feature of all these tools, and of DD-suite, is that the user specifies a problem through a recursive, DP model (i.e., a state representation together with transition, merging, and discarding rules), from which the DD is automatically constructed. The main difference lies in their intended use: `ddo` and CODD are *end-to-end solvers* in which the DD machinery is internal to a B&B search, whereas DD-suite is a *library* that exposes the DD construction and the resulting graph so that they can be reused as a building block for arbitrary DD-based algorithms.

3 Background on Decision Diagrams

This section provides a brief overview of DDs for optimization. We refer the reader to [12,18,33] for further details on DDs and their usage. In what follows, we use calligraphic font for sets, uppercase letters for the sets' cardinality, and lowercase letters for the sets' elements if possible (e.g., $a \in \mathcal{A}$ is an element of set $\mathcal{A}$ with cardinality $|\mathcal{A}| = A$). We also employ bold letters for vectors and sans serif font for functions (e.g., $\boldsymbol{x} = (x_1, x_2) \in \mathbb{Z}^2$ and $\mathsf{a}(\cdot) : \mathbb{Z} \to \mathbb{R}$ is a function).

Given the strong relationship between DP and DDs [34], we consider recursive formulations of discrete problems to explain DDs and as the problem specification for our software (see Section 4 for more details). Specifically, we consider discrete problems that can be represented sequentially and have a finite number of variables with bounded domain (i.e., $\boldsymbol{x} = (x_1, ..., x_T) \in \mathscr{X} \subset \mathbb{Z}^T$, where $T \in \mathbb{Z}$ and $\mathscr{X}$ is bounded).

Consider $\mathscr{T} = \{1, ..., T\}$ as the set of stages for the recursive formulation, s^I as the initial state of the system, and $\mathscr{S}$ as the set of reachable states starting from s^I. Abusing the notation, we use $\mathscr{S}_t$ to represent reachable states at stage $t \in \mathscr{T}$ and $\mathscr{X}_t(s)$ as the set of feasible assignments of x_t at state $s \in \mathscr{S}_t$. Then, the Bellman equations describing a maximization problem for all $s \in \mathscr{S}$ are:

$$\mathsf{f}_t(s) = \begin{cases} \max_{x_t \in \mathscr{X}_t(s)} \{\mathsf{r}_t(s, x_t) + \mathsf{f}_{t+1}(\mathsf{g}_t(s, x_t))\}, & s \in \mathscr{S}_t,\ t \in \mathscr{T} \setminus \{T\}, \\ \max_{x_t \in \mathscr{X}_t(s)} \{\mathsf{r}_t(s, x_t)\}, & s \in \mathscr{S}_T,\ t = T. \end{cases} \tag{1}$$

The value function at stage $t \in \mathscr{T}$ is given by $\mathsf{f}_t(\cdot) : \mathscr{S} \to \mathbb{R}$, while $\mathsf{r}_t(\cdot, \cdot) : \mathscr{S}_t \times \mathscr{X}_t \to \mathbb{R}$ is the immediate reward of choosing a feasible value for variable $x_t \in \mathscr{X}_t(s)$ at state $s \in \mathscr{S}_t$. Lastly, $\mathsf{g}_t(\cdot, \cdot) : \mathscr{S}_t \times \mathscr{X}_t \to \mathscr{S}_{t+1}$ corresponds to the transition function of choosing a value for x_t at state s. Then, the optimal solution for this recursive model is given by $\mathsf{f}_1(s^I)$.

Example 1 To better explain the main concepts related to DDs, we borrow the knapsack example in [18] as our running example throughout the paper. Consider a knapsack problem with four variables, weight vector $\boldsymbol{w} = (7, 5, 4, 1)$, and reward vector $\boldsymbol{c} = (4, 2, 5, 1)$, that is,

$$\begin{aligned} \max\ & 4x_1 + 2x_2 + 5x_3 + x_4 \\ \text{s.t. } & 7x_1 + 5x_2 + 4x_3 + x_4 \leq 8, \\ & \boldsymbol{x} = (x_1, ..., x_4) \in \{0, 1\}^4. \end{aligned}$$

A recursive formulation of the problem considers choosing one item per stage in lexicographic order. Each state represents the current load of the knapsack (i.e., an integer value), and the initial state is given by $s^I = 0$. For a given state $s \in \mathscr{S}_t$ and stage $t \in \mathscr{T}$, the set of feasible values of x_t is $\mathscr{X}_t(s) = \{x \in \{0, 1\} : \ s + w_t x \leq 8\}$. Then, the transition function for $s \in \mathscr{S}_t$ and $x_t \in \mathscr{X}_t(s)$ is $\mathsf{g}_t(s, x_t) = s + w_t x_t$. Note that the immediate reward function is independent of the state and, thus, is given by $\mathsf{r}_t(s, x_t) = c_t x_t$. Therefore, the Bellman equations governing this problem are:

$$\mathsf{f}_t(s) = \begin{cases} \max_{x_t \in \mathscr{X}_t(s)} \{c_t x_t + \mathsf{f}_{t+1}(s + w_t x_t)\}, & s \in \mathscr{S}_t,\ t \in \{1, ..., 3\}, \\ \max_{x_t \in \mathscr{X}_t(s)} \{c_t x_t\}, & s \in \mathscr{S}_4,\ t = 4, \end{cases} \quad \forall s \in \mathscr{S}.$$

□

3.1 Exact Decision Diagrams

DDs are graphical representations of the feasible set of a discrete problem. Given a recursive formulation, a DD can be seen as a compact representation of the transition graph of such a model, where nodes map to states and arcs map to feasible

variable assignments. We now describe the main components and properties of such a graphical structure.

Consider the recursive model given by equations (1). Formally, a DD is a directed acyclic layered graph $\mathscr{D} = (\mathscr{N}, \mathscr{A})$, where $\mathscr{N}$ and $\mathscr{A}$ are the set of nodes and arcs, respectively. The set of nodes is partitioned into $T+1$ layers $\mathscr{N} = \{\mathscr{N}_1, ..., \mathscr{N}_{T+1}\}$ such that each node $n \in \mathscr{N}_t$ is associated with a single state $s \in \mathscr{S}_t$, for all $t \in \mathscr{T}$. We use the function $\mathsf{s}(\cdot) : \mathscr{N} \to \mathscr{S}$ to represent the state associated with a node. In particular, the first node layer has a single node, called the *root node* (i.e., $\mathscr{N}_1 = \{\mathsf{r}\}$), which is associated with the initial state (i.e., $\mathsf{s}(\mathsf{r}) = s^I$). The last node layer also has a single node called the *terminal node* (i.e., $\mathscr{N}_{T+1} = \{\mathsf{t}\}$) and represents all the terminal states for the recursive model. For simplicity, we associate a dummy state s^D to the terminal node (i.e., $\mathsf{s}(\mathsf{t}) = s^D$).

The set of arcs is partitioned into T layers, $\mathscr{A} = \{\mathscr{A}_1, ..., \mathscr{A}_T\}$, such that an arc $a \in \mathscr{A}_t$ emanates from its parent node $\mathsf{p}_a \in \mathscr{N}_t$ and points to its child node $\mathsf{c}_a \in \mathscr{N}_{t+1}$, for all $t \in \mathscr{T}$. Each arc $a \in \mathscr{A}_t$, for $t \in \mathscr{T}$, has a value $\mathsf{v}_a \in \mathscr{X}_t(\mathsf{s}(\mathsf{p}_a))$ corresponding to a feasible assignment of x_t; and its child node is associated with the state given by the transition function (i.e., $\mathsf{s}(\mathsf{c}_a) = \mathsf{g}_t(\mathsf{s}(\mathsf{p}_a), \mathsf{v}_a)$). Given that the last node layer has a single node, the child node for all arcs in the last layer is the terminal node (i.e., $\mathsf{c}_a = \mathsf{t}$ for all $a \in \mathscr{A}_T$), and thus, the transition function should return the dummy state s^D in such cases. Abusing the notation, we use $\mathscr{A}^{\text{out}}(n)$ and $\mathscr{A}^{\text{in}}(n)$ to represent the set of outgoing and incoming arcs for any node $n \in \mathscr{N}$ (i.e., $a \in \mathscr{A}^{\text{out}}(n)$ and $a \in \mathscr{A}^{\text{in}}(n')$ if and only if $\mathsf{p}_a = n$ and $\mathsf{c}_a = n'$).

As illustrated in Example 2, a DD is indeed a compact representation of the state transition graph of a recursive model. Moreover, we say that a DD is *exact* if a one-to-one mapping exists between the feasible assignments of $\boldsymbol{x}$ and the $\mathsf{r}-\mathsf{t}$ paths on the DD (e.g., a DD following the description provided in this section). In such case, we can obtain the optimal solution of the recursive model by solving the longest-path problem from r to t where the length of each arc $a \in \mathscr{A}_t, t \in \mathscr{T}$, is given by the reward function as $\ell_a = \mathsf{r}_t(\mathsf{s}(\mathsf{p}_a), \mathsf{v}_a)$.

One of the main advantages of DDs is that we can further compress the graphical structure without sacrificing any feasible assignment. This is done using a reduction procedure [15] that merges isomorphic subgraphs by traversing each node layer once, starting from $\mathscr{N}_{T+1}$, and runs in polynomial time with respect to the DD size (i.e., number of nodes and arcs). The procedure returns the smallest exact DD at the cost of some nodes representing two or more states. Moreover, we can use any longest-path algorithm on the reduced graph to obtain the optimal solution if the reward function is not state-dependent (e.g., linear).

Example 2 Following the knapsack problem described in Example 1, Figure 1 presents the state transition graph for the model (left), an exact DD (middle), and its reduced version (right). In all graphs, the numbers inside the circles correspond to states, and arcs represent variable assignments (solid arcs for $x_t = 1$ and dashed arcs for $x_t = 0$). Also, numbers next to arcs are their associated length given by the reward function, and the bold paths correspond to the optimal solution to the problem.

We observe that the middle graph is an exact DD where each node is associated with a single state. The main difference with the transition graph is that there is a

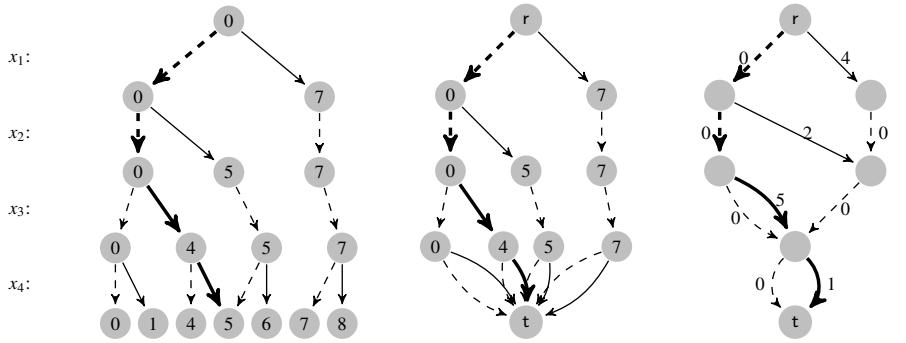


Fig. 1: State transition graph, exact DD, and its reduced version.

single node in the last layer (i.e., the terminal node t). The reduced DD is obtained by applying the reduction procedure [15] to the middle DD. This rightmost DD has considerably fewer nodes (i.e., at most two per layer), and it preserves all the paths (i.e., feasible assignments) present in the transition graph and the middle DD. Lastly, we can obtain the optimal solution from all graphical structures using a longest-path algorithm where $\ell_a = c_t \mathsf{v}_a$ for all $a \in \mathscr{A}_t, t \in \mathscr{T}$, which corresponds to $\boldsymbol{x}^* = (0,0,1,1)$ with value 6. □

3.2 Approximated Decision Diagrams

One of the main drawbacks of exact DDs is that they suffer from the *curse of dimensionality* [7]; that is, the number of nodes and arcs grows exponentially with the number of variables for most problems. While the reduction procedure can somewhat mitigate this effect, its final size can be exponential in the number of variables. To overcome this issue, researchers have proposed two types of DDs: underapproximations (i.e., restricted DDs) or overapproximations (i.e., relaxed DDs) of the feasibility set of the original problem. We now describe each approximated DD and provide general insights.

3.2.1 Restricted DDs

Restricted DDs were first introduced as a general procedure to obtain high-quality solutions for discrete optimization problems [10], but they can also be used, for instance, to create relaxations for bilevel problems [54]. The main idea is to construct a limited-size DD by discarding nodes during construction that could lead to poor-quality solutions. The most common approach to limit a DD size is through its maximum allowed width, that is, the maximum number of nodes per layer. Formally, given a DD $\mathscr{D} = (\mathscr{N}, \mathscr{A})$, we define the width of a node layer $\mathscr{N}_t$ as the number of nodes for such layer, that is, $w(\mathscr{N}_t) = |\mathscr{N}_t|$ for all $t \in \mathscr{T}$. Then, the maximum width of a DD is given by $W = \max_{t \in \mathscr{T}} \{w(\mathscr{N}_t)\}$.

For a given maximum width W, we build a restricted DD following a procedure similar to the one used for an exact DD (i.e., one node per state in each stage). If the number of nodes in a layer is larger than W, then we choose W nodes that might lead to an optimal solution and discard the rest. This discarding procedure is usually heuristic and problem-specific (see, for instance, Example 3). The resulting restricted DD has a limited size where the number of nodes per layer is at most W, and its paths represent a subset of all possible solutions to the original problem. Therefore, we can optimize over the resulting restricted DD and obtain a primal bound of the original problem. More details of the construction mechanism can be found in Section 3.3.

3.2.2 Relaxed DDs

Relaxed DDs are graphical structures that represent a discrete relaxation of an optimization problem and, thus, can be used to obtain dual bounds [3]. Similar to restricted DDs, relaxed DDs can be constructed by limiting the number of nodes in each layer (i.e., its maximum width). However, instead of discarding nodes when the size is exceeded, these DDs merge nodes such that a node represents two or more states. By doing so, the resulting DD might incorporate paths that represent infeasible assignments, thus overapproximating the original problem.

While the main idea of relaxed DDs is quite simple (i.e., merging nodes and states), it is crucial to define the merging mechanism and its implications on the recursive model (i.e., states, transition function, and reward function) to guarantee that the resulting DD indeed relaxes the original problem. Hooker [35] formalizes the key properties needed for a merging operator to obtain a relaxed DD. Specifically, given two states $s, s' \in \mathscr{S}_t$ for any $t \in \mathscr{T}$, we define a *merged state* $\tilde{s} = s \oplus s'$, where $\oplus$ is a merge operator. This merge operator defines a proper relaxation if the following conditions are satisfied:

(i) The feasibility set of $\tilde{s}$ contains all the feasible assignments associated with s and s', that is, $\mathscr{X}_t(s) \cup \mathscr{X}_t(s') \subseteq \mathscr{X}_t(\tilde{s})$.
(ii) The immediate reward at state $\tilde{s}$ is greater than or equal to the immediate reward at s and s' for any feasible assignment, that is, given $x \in \mathscr{X}_t(s)$ and $x' \in \mathscr{X}_t(s')$ we have $\mathsf{r}_t(s,x) \leq \mathsf{r}_t(\tilde{s},x)$ and $\mathsf{r}_t(s',x') \leq \mathsf{r}_t(\tilde{s},x')$.
(iii) If $\tilde{s}$ *relaxes* state s (i.e., $\tilde{s}$ satisfies (i) and (ii)), then $\mathsf{g}_t(\tilde{s},x)$ relaxes state $\mathsf{g}_t(s,x)$ for all $x \in \mathscr{X}_t(s)$.

As shown in [35], these three conditions are enough to guarantee that the merging operator $\oplus$ will provide a relaxed DD that indeed overapproximates the feasible set of the problem and, thus, can provide dual bounds. As in the restricted DD case, the merging operator is problem-specific, and choosing which nodes to merge is usually done heuristically (see Example 3 for its application to a knapsack instance).

Example 3 Following the knapsack instance described in the previous examples, Figure 2 depicts a restricted DD (left) and a relaxed DD (right) of maximum width $W = 2$ for such a problem. The restricted DD uses a discarding heuristic where we keep the nodes with the highest knapsack load. Indeed, the third layer has only two nodes and discards the node with $s = 0$. Note that every path in the restricted DD corresponds

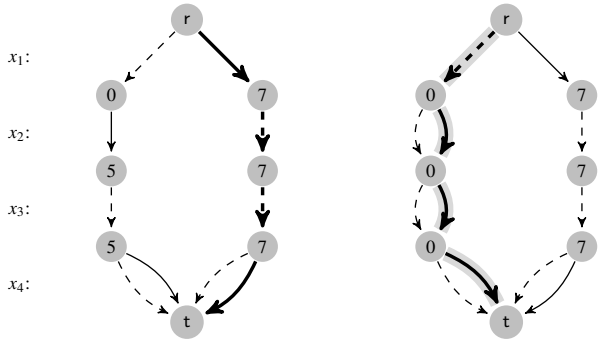


Fig. 2: Restricted and relaxed DDs for the knapsack example.

to a feasible assignment of the problem, but some solutions (e.g., $\boldsymbol{x} = (0,0,0,0)$) are not represented in the DD. The optimal solution (i.e., a primal bound for the original problem) is depicted by the bold path in the figure, which represents solution $\boldsymbol{x} = (1,0,0,1)$ with value 5.

To define a relaxed DD, the merging operator keeps the smallest load among the merged nodes, that is, $\tilde{s} = s \oplus s' = \min\{s, s'\}$ for any two states $s, s' \in \mathscr{S}_t$ (for some $t \in \mathscr{T}$). Since a smaller load leaves more room in the knapsack, the merged state contains all the feasible assignments of s and s'; and, since the reward function is state-independent, we keep it as it is. Note that tracking only the smallest load suffices to obtain a valid relaxation precisely because the reward does not depend on the state. This merging operator follows conditions (i)-(iii) and, thus, provides a relaxed DD.

The relaxed DD in Figure 2 utilizes a merging heuristic (which nodes to merge) that merges nodes with the lowest knapsack load. Note that all feasible solutions are represented in the graph, but some paths (i.e., the shaded path) are infeasible. Indeed, the optimal solution corresponds to the shaded path associated with $\boldsymbol{x} = (0,1,1,1)$ and value 8. Therefore, the restricted DD provides a primal bound of 5 and the relaxed DD a dual bound of 8 (recall that the optimal value is 6). □

The main advantage of these two DD approximations is that the user can easily control the quality of the bounds by simply adjusting the maximum width or trying different discarding/merging strategies [11]. Moreover, they can be used to find the optimal solution to a problem using a searching procedure such as B&B [9,24] or A^* search [19,20], or be used as a complement to existing mathematical programming solvers (e.g., to generate cuts [17,25,53] or solve the subproblems of a Lagrangian decomposition approach [8,16], among many others). We refer the reader to [18,33] for further recent examples of how to employ exact and approximated DDs to solve optimization problems.

Algorithm 1 DD Top-Down Construction

1: **procedure** TOPDOWNDD(W)
2: Create DD $\mathscr{D} = (\mathscr{N}, \mathscr{A})$ with $T+1$ empty node layers
3: Create the root and terminal node, i.e., $\mathsf{r} \in \mathscr{N}_1$ with $\mathsf{s}(\mathsf{r}) = s^I$ and $\mathsf{t} \in \mathscr{N}_{T+1}$ with $\mathsf{s}(\mathsf{t}) = s^D$
4: **for** $t \in \mathscr{T} \setminus \{T\}$ **do**
5: **for** $n \in \mathscr{N}_t$ **do**
6: Create an arc a emanating from n for each possible value in $\mathscr{X}_t(\mathsf{s}(n))$
7: **for** $a \in \mathscr{A}^{\text{out}}(n)$ **do**
8: **if** there exists node $n' \in \mathscr{N}_{t+1}$ with $\mathsf{s}(n') = \mathsf{g}_t(\mathsf{s}(n), \mathsf{v}_a)$ **then**
9: Direct arc a to node n', i.e., $\mathsf{c}_a = n'$
10: **else**
11: Create n' in $\mathscr{N}_{t+1}$ with $\mathsf{s}(n') = \mathsf{g}_t(\mathsf{s}(n), \mathsf{v}_a)$ and point arc a to n'
12: **if** $|\mathscr{N}_{t+1}| > W$ **then**
13: Apply MergeDDNodes($\mathscr{N}_{t+1}$) (relaxed DD) or DiscardDDNodes($\mathscr{N}_{t+1}$) (restricted DD)
14: **for** $n \in \mathscr{N}_T$ **do**
15: Create an arc a with $\mathsf{p}_a = n$ and $\mathsf{c}_a = \mathsf{t}$ for each value in $\mathscr{X}_T(\mathsf{s}(n))$
16: **return** $\mathscr{D}$

3.3 Construction Mechanism

We now formally present the top-down construction mechanism for exact and approximated DDs. We note that DD-suite uses this procedure to construct DDs, which can be easily tailored to different problems, as we discuss in Section 4. As previously mentioned, the procedure employs a recursive formulation for a discrete problem with a fixed number of variables (i.e., given by the Bellman equations (1)) and constructs all possible states at each stage.

Specifically, Algorithm 1 describes the complete top-down construction procedure. The algorithm first initializes the graphical structure and sets the states for the root and terminal node accordingly (lines 2–3). We then iterate over each node layer in lexicographic order, and for each node, we create an arc for every feasible value associated with that node's state (lines 4–6). We then create the nodes in the next layer using the recursive model's transition function, ensuring that two nodes cannot be associated with the same state (i.e., lines 7–11). If the maximum width of the following layer is exceeded, we can merge or discard nodes depending on the type of DD we want (lines 12–13). Lastly, we create the arcs in the last layer and direct them to the terminal node.

Note that this procedure can create exact and approximated DDs depending on the maximum width and the procedure to limit the width size. Moreover, this construction mechanism allows the user to obtain a wide range of approximated DDs by simply changing the merge and discarding functions.

4 DD-suite: basic features and documentation

DD-suite is an open-source cross-platform code for creating and utilizing DDs to solve discrete optimization problems. Both implementations (C++ and Python) share a similar structure, with identical file names, classes, and function names, albeit with minor differences due to language-specific semantics. We opt for these two lan-

guages because they are widely used in the mathematical programming community, and many open-source and commercial solvers support them (e.g., Gurobi [31] and CPLEX [36]).

While the two DD-suite implementations are equivalent, we develop each one with consideration for users' potential usage and coding capabilities. We consider the Python version to be a good starting point for people new to DDs and for prototyping purposes. On the other hand, the C++ version could be preferable for researchers interested in computational performance and extending DD-suite capabilities. Nonetheless, users can use any version depending on their preferences and expertise.

We now detail the basic features of DD-suite, that is, any usage that requires minimal intervention in the existing code. The primary purpose of DD-suite is to enable users to construct and utilize DDs to solve discrete optimization problems easily. Thus, the following sections explain how to specify a problem, build a DD, and obtain optimality bounds. We illustrate these basic features using the Python version for ease of exposition and use `typewriter` font to refer to code classes and functions. The last section discusses the code documentation and presents a webpage where users can find updated information and step-by-step tutorials.

4.1 Problem specification

We now explain how to specify and implement a discrete optimization problem in DD-suite, which requires a recursive formulation similar to the one introduced in Section 3. Specifically, DD-suite uses *abstract classes* and *inheritance* to specify new problems, allowing us to clearly instruct users on how to implement their recursive formulation and other problem specifications. Thus, the code includes a class called `AbstractProblem` that serves as the foundation for any problem we want to specify.

Users have to create a new class that inherits from `AbstractProblem` and complete the functions that detail their recursive model to introduce a new problem to DD-suite. Similarly to the recursive formulation (1), the constructor of the new problem class receives the initial state and the variables of the problem. DD-suite accepts any data structure as a state, including pre-defined (e.g., `int`, `vector`) and user-specified structures (e.g., `my_state`), as long as the user utilizes the same data structure for all states for a particular DD. Additionally, variables can have any bounded integer domain, which can be distinct from one another (e.g., $x_1 \in \{0,1\}$ and $x_2 \in \{0,1,2\}$). The class constructor can also receive any other type of argument that is needed to define the problem (e.g., items' weights in the knapsack problem).

Next, users need to specify the `transition_function()`, which has two primary purposes: (i) detail how to transition from one state to the other and (ii) identify feasible variable assignments (i.e., implement $\mathrm{g}_t(\cdot)$ and $\mathscr{X}_t$, respectively, for any stage $t \in \mathscr{T}$). Users have complete flexibility when implementing this transition function; however, we recommend a highly efficient implementation, as it is called multiple times during the DD construction procedure and, thus, directly affects the DD construction time.

The constructor and the `transition_function()` of the new problem class are the most crucial parts to specify a new problem properly. Our `AbstractProblem`

class also requires users to implement additional functions to ensure that everything works smoothly, which we discuss in more detail in the tutorial provided on our webpage (see Section 4.4). As an overview, users have to implement the functions `get_state_as_string()` and `get_state_copy()`, which are used during the DD construction procedure. Additionally, users must specify additional functions when creating relaxed and restricted DDs, which we detail in the following section.

Note that the environment for problem specification is quite general and can be applied to a wide range of problems. DD-suite provides four problem examples (i.e., knapsack, set cover, maximum independent set, and sequencing problems) for users to experiment with, as well as a SOC knapsack example used in the cutting plane experiments of Section 6.4. Specifying new problems requires minimal implementation and is quite simple given a recursive formulation, as illustrated in Example 4.

We note that the DD-suite problem specification only describes the feasibility set (i.e., reachable states and feasible variable assignments) and omits the reward function, which is encoded separately (see Section 4.3 for further details). Our implementation intentionally keeps this separation, allowing users to use the same DD graph (i.e., feasibility set) to optimize different objective functions, which is desirable in some DD applications, such as cut generation (see Section 5). Nonetheless, users can include reward function information into their problem class if desired (e.g., to guide merging and discarding heuristics) by introducing such information in the class constructor or even adding it to their state definition.

Example 4 Continuing with our running example, Listing 1 presents the constructor and transition function implementation for a knapsack problem (Listing 7 in Appendix A shows a complete implementation of the `KnapsackProblem` class). The `KnapsackProblem` class inherits from the `AbstractProblem` class and implements the recursive model as follows. The constructor receives a single problem-instance object whose attributes bundle the initial state, the variables (i.e., a list of the variable ids and their domain), and the parameters of the knapsack constraint (i.e., items' weights and knapsack capacity). This example uses a single integer as the state representation for the current knapsack load, exactly as in Examples 1 and 2. The transition function follows the definition of $g_t(\cdot)$ in Example 1: it writes the new state into a scratch buffer and returns whether the resulting load is feasible (i.e., it does not exceed the knapsack capacity).

```
class KnapsackProblem(AbstractProblem):

    def __init__(self, params: 'KnapsackStructure'):
        super().__init__(params.initial_state, params.variables)
        self.weights: list[int] = params.weights
        self.capacity: int = params.right_side_of_restrictions

    def transition_function(self, previous_state: 'State', variable_index:
     int, variable_value: int, scratch_state: list) -> bool:

        if variable_value == 0:
            scratch_state[0] = previous_state
            return True

        new_state: int = previous_state + self.weights[variable_index] *
     variable_value
        scratch_state[0] = new_state
```

```
        return new_state <= self.capacity
```

Listing 1: Partial knapsack problem class implementation in Python.

To create an instance of the problem, we simply create a class object and initialize the required parameters. Listing 2 shows the input parameters for our running example with four variables and how to create the problem instance (see Listing 8 in Appendix A for the complete main code of this running example). □

```
# Maximum width for the approximated DDs
width: int = 2

# Input data for the running example
params = SimpleNamespace(
    initial_state=0,
    variables=[('x_1', [0, 1]), ('x_2', [0, 1]), ('x_3', [0, 1]), ('x_4',
     [0, 1])],
    weights=[7, 5, 4, 1],
    right_side_of_restrictions=8,
)

# Objective coefficients and sense used later to optimize over the DDs
utility: list[int] = [4, 2, 5, 1]
objective: str = "max"

# Create the knapsack problem used to build the DD
knapsack_problem = KnapsackProblem(params)
```

Listing 2: Initialize and create a knapsack problem instance.

4.2 DD construction and personalization

Users can construct a DD by creating an object of the provided `DD` class, which takes as input an object representing the problem instance. Then, users can call any of the provided functions to create a DD, that is, `create_decision_diagram()` for exact DDs, `create_restricted_decision_diagram()` for restricted DDs, and `create_relax_priority_decision_diagram()` for relaxed DDs. DD-suite includes an alternative relaxed DD constructor, `create_relax_grouping_decision_diagram()`, which we further discuss in Section 5.2.

DD-suite utilizes the top-down construction procedure detailed in Algorithm 1. This procedure creates each layer sequentially, following the variable ordering provided by the problem class. Users can control this ordering through the problem class: by default, the variables are processed in the order given to the constructor, but users can define a custom ordering by overriding the `sort_variables()` method, which is applied when the problem is constructed with the `sort` flag enabled. Also, similarly to Algorithm 1, our implementation of the top-down construction procedure is general for all DD types (see Section 5 for further details) and only differs in whether we keep, discard, or merge nodes when a maximum width is exceeded (for exact, restricted, and relaxed DDs, respectively).

Specifically, function `create_decision_diagram()` constructs an exact DD by simply creating as many nodes as needed in every layer. In contrast, we can create a restricted DD via the command `create_restricted_decision_diagram()`,

which receives a maximum width as input and discards nodes when it is exceeded. Users can specify a discarding heuristic with the `get_priority_for_discard_node()` function available in the problem class. This function receives the state of a node and assigns it a priority, where states (i.e., nodes) with higher priority are discarded first.

Similarly, `create_relax_priority_decision_diagram()` receives a maximum width as input, and users can specify how to merge nodes in a relaxed DD with functions `get_priority_for_merge_nodes()` and `merge_operator()` in their problem class. The first function assigns a priority to each node based on its state information. Then, the construction procedure orders all the nodes by priority and repeatedly merges the two highest-priority nodes until the layer fits within the maximum width. The merging operator is specified by the user using the `merge_operator()` function, which returns the new merged state. We note that users have complete flexibility in implementing their priority and merging functions, but recall that `merge_operator()` must satisfy the properties detailed in Section 3.2.2 to ensure that the resulting DD is indeed relaxed.

While there are many ways to decide which nodes should be discarded or merged, we opt for this priority list type of implementation due to its simplicity and flexibility in accommodating a wide range of heuristics. We note that DD-suite can also be adapted to consider alternative methods for determining which nodes to discard or merge, which is further discussed in Section 5.2.

Once the DD is created, class `DD` provides the `reduce_decision_diagram()` function, which applies the reduction procedure mentioned in Section 3.3. Additionally, we can obtain the construction and reduction times using `get_building_time()` and `get_reduction_time()`, respectively, as well as graph information, including the number of nodes and arcs (i.e., `get_decision_diagram().get_node_count()` and `get_decision_diagram().get_arc_count()`, respectively). Lastly, users can export the DD graph with the `export_graph_file()` command and visualize the resulting DD using, for instance, the yEd software[2]. We refer the reader to the DD-suite webpage for further instructions on DD visualization.

```
dd_exact = DD(knapsack_problem)
dd_exact.create_decision_diagram()
dd_exact.reduce_decision_diagram()

# Print DD construction information
print("-- Exact DD information --")
print("\tConstruction time (sec): ", dd_exact.get_building_time())
print("\tReduction time (sec): ", dd_exact.get_reduction_time())
print("\tNumber of nodes: ", dd_exact.get_decision_diagram().get_node_count
    ())
print("\tNumber of arcs: ", dd_exact.get_decision_diagram().get_arc_count()
    )
```

Listing 3: Construct an exact DD, and print relevant information.

Example 5 Following up on our running example, Listing 3 illustrates the command to create and reduce an exact DD for our knapsack instance. Similar commands can be used to create relaxed and restricted DDs, which are available in Appendix A, Listing 8. The example code also shows how to obtain the building and reduction

[2] Free graph editor software https://www.yworks.com/products/yed.

times, as well as some key features of the DD graph (i.e., number of nodes and arcs). Lastly, the code illustrates how to export the DD graph into a `.gml` file using the `export_graph_file()` command, which can then be visualized using the free software yEd. Figure 8 in Appendix E shows the resulting visualizations of such software for all three DD types, which are equivalent to the DDs illustrated in Figures 1 and 2.

4.3 Obtain primal and dual bounds

As discussed in Section 3, we can obtain an optimal solution from a DD using a longest-path algorithm over the graph where arc lengths correspond to the reward function. In the case of approximated DDs, such a procedure provides either primal or dual bounds, depending on whether the DD is restricted or relaxed, respectively. DD-suite includes this optimization feature for cases where the objective function is linear, a common case in applications [18,33].

The `ShortestLongestPath` class provides algorithms for finding the longest or shortest path, depending on whether the problem is maximization or minimization. Its implementation is a simple Bellman–Ford shortest-path algorithm [1] that leverages the structure of the DD graph (i.e., a layered directed acyclic graph) to improve computational efficiency. To use this procedure, users must create an object of the `ShortestLongestPath` class that receives as input a `DD` object. Then, users set the arc lengths and the optimization sign (i.e., min or max) with the `set_parameters()` function and optimize with the `solve()` command. This command returns the solution in a data structure called `PathStructureSolution`, which includes the optimal value (`value`), a human-readable description of the optimal path (`path_print`), and the list of arcs that form such a path (`path_arcs`). The running time of the algorithm can be retrieved separately with the `get_time()` command.

We note that each object of `ShortestLongestPath` is associated with a single DD, but the objective function parameters can be updated as many times as the user wants. This characteristic is particularly useful when using a DD as a sub-routine of a more complex optimization approach, such as Benders decomposition, column generation, or cutting planes (see Section 5.4).

Example 6 Listing 4 illustrates how to use the `ShortestLongestPath` class to find an optimal solution over the exact DD for our running example (equivalent code for restricted and relaxed DDs can be found in Appendix A, Listing 8). The class object receives the DD and then updates its cost function with a reward vector (i.e., `utility`) and the optimization type (i.e., `objective = "max"`). Lastly, the code executes the longest-path procedure using the `solve()` command and illustrates how to obtain information about the solution, including its value and the optimal path, as well as the algorithm's execution time via `get_time()`.

```
# Optimal solution over the exact DD (longest path)
longest_path = ShortestLongestPath(dd_exact)
longest_path.set_parameters(utility, objective)
answer = longest_path.solve()

print("The optimal value is: ", answer.value)
print("Longest path: ", answer.path_print)
```

```
print("Solution time (sec): ", longest_path.get_time())
```

Listing 4: Obtain optimal solution and print relevant information.

4.4 Documentation, examples, and webpage

To facilitate DD-suite usage, our code provides thorough documentation of all classes and functions. This documentation follows standard software engineering documentation practices (i.e., descriptive docstrings for every class, method, and function, including their parameters and return values) and is included in the foundation code and all the provided examples. Moreover, DD-suite follows the Clean Code principles [43] (e.g., meaningful names, small functions and classes, etc.), so it is user-friendly and self-explanatory. We further detail the advantages of following such software engineering principles in Section 5.

Users can run the examples described above using the provided data instances and one of the main files provided to reproduce our experiments (see Section 6 for further details) or create their own main files to try out DD-suite functionalities. Moreover, these examples include automatic testing, so users can easily check if the code still runs smoothly if they make changes to it (see Section 5 for further details).

In addition, we have set up a webpage for DD-suite[3]. The site contains information on how to set up the Python and C++ versions and run the code to reproduce our experiments. Moreover, we provide a step-by-step tutorial that focuses on the basic features detailed in this section and some advanced features (i.e., cutting planes, see Section 5.4) for both the Python and C++ implementations. The webpage also includes essential information for developers, such as how to run and create new tests and a description of the most important classes of the software.

5 DD-suite for developers: advanced usage and extensions

As previously mentioned, DD-suite allows users to specify discrete optimization problems easily, create DDs for them, and obtain optimality bounds with just a few lines of code. However, this is just the tip of the iceberg given the diverse uses of DDs in the literature [18,33]. This section presents the main software characteristics of DD-suite that make it an ideal code base for DD-based algorithms and applications. We then discuss how users can extend and apply DD-suite for their research projects, including our DD-based cutting plane implementation as an illustrative example.

5.1 Software characteristics and tests

DD-suite is designed so that researchers can add new procedures, such as algorithms running on top of a DD or new DD construction mechanisms, without rewriting the

[3] https://dd-suite.dev/

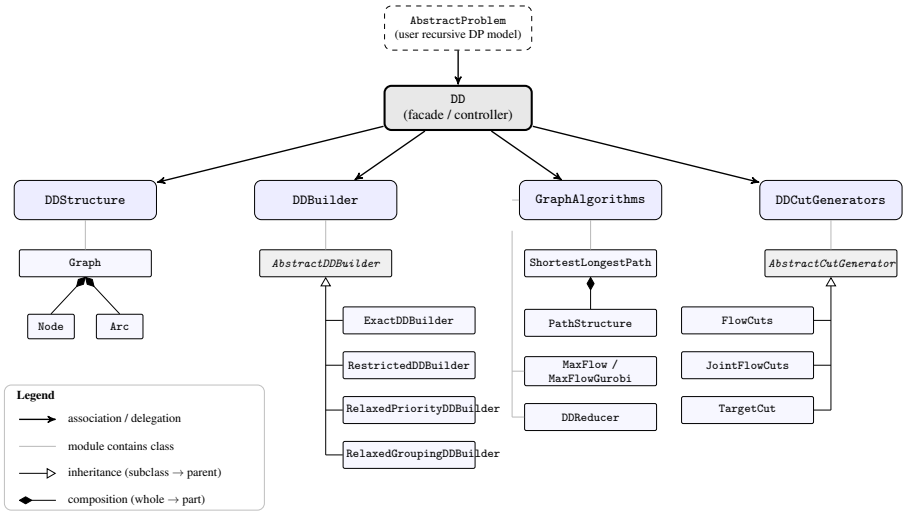


Fig. 3: Class-level architecture of DD-suite.

existing code. We achieve this by following Clean Code software engineering principles [43]: meaningful names, small single-purpose functions, and, most importantly, a modular architecture where every procedure is encapsulated in its own component. The code is organized into independent modules, namely the DD data structures (`DDStructure`), the construction procedures (`DDBuilder`), the algorithms that interact directly with the graphical structure (e.g., shortest/longest path and reduction), and advanced algorithms for optimization purposes (e.g., cutting planes). The `DD` class acts as a controller (i.e., a facade) that coordinates these modules and exposes a clean interface to the user. As a consequence, the components are decoupled: changing how a DD is constructed does not affect the algorithms that run on top of it, and vice versa, as long as the shared `Graph` interface exposed by `DDStructure` is respected. This separation is what makes DD-suite easy to extend, as we illustrate in the following sections. Figure 3 summarizes these modules and how they interact through the `DD` facade, together with the main classes inside each module and their relationships. In particular, the figure shows how the data structures are composed, where a `Graph` is built from `Node` and `Arc` objects and a `ShortestLongestPath` from `PathStructure` objects, as well as the abstract classes that new procedures specialize, which we discuss in the following sections.

A key ingredient that enables modularity is that all construction procedures share a single top-down construction template, implemented in the `AbstractDDBuilder` class following Algorithm 1. This template builds the diagram layer by layer and exposes two extension points (i.e., abstract methods) that each DD type implements: one invoked after every layer is completed and one invoked at the end of the construction. The exact, restricted, and relaxed builders are all concrete subclasses of `AbstractDDBuilder` that differ only in these two methods (i.e., whether they keep,

discard, or merge nodes when the maximum width W is exceeded), while reusing the rest of the construction machinery unchanged.

To safeguard this code base against future modifications, DD-suite includes an extensive automatic test suite for both implementations, using the `unittest`/`pytest` frameworks in Python and Google Test in C++. At the time of writing, the suite comprises more than 230 tests in Python and 260 in C++, and this number keeps growing as new problems and procedures are added. The tests cover each example problem and DD type (exact, restricted, and relaxed, with and without custom variable ordering), the graph operations (e.g., reduction and graph equivalence), and the cut algorithms. This testing environment serves a double purpose: it protects the existing functionality, and it gives developers a safety net to experiment with new ideas with the guarantee that they are not silently breaking the original code. We provide instructions on how to run and write new tests on the DD-suite webpage.

5.2 Extending the DD construction mechanism

As previously mentioned, the top-down construction procedure is implemented in the `AbstractDDBuilder` class, which builds the diagram following Algorithm 1 and delegates the type-specific behavior to two methods: `_specific_end_of_layer_function()`, called after each layer is built, and `_specific_end_of_construction_function()`, called once after the last layer. To incorporate a new construction mechanism, a developer only needs to create a new subclass of `AbstractDDBuilder` and implement these two methods, reusing all the underlying machinery.

We illustrate this extension mechanism with the alternative grouping-based relaxation `create_relax_grouping_decision_diagram()` routine. Recall that the default relaxed DD builder (i.e., `RelaxedPriorityDDBuilder`) merges the two highest-priority nodes. In contrast, the grouping variant (i.e., `RelaxedGroupingDDBuilder`) merges several nodes at once, as described in Algorithm 2: it sorts the nodes of the layer by merge priority and, in successive passes, merges every group of consecutive nodes whose priority gap lies within a tolerance δ. The tolerance starts at $\delta = 0$, so that only nodes with equal priority are merged, and grows after each pass until the layer fits within the maximum width W; to avoid passes that would merge nothing, δ jumps to the smallest priority gap still present in the layer whenever that gap is larger. Note that this procedure is a drop-in replacement for the MergeDDNodes routine of Algorithm 1 (line 13) and reuses the same user-defined priority and merge operators.

Listing 5 shows the skeleton of this new builder: implementing the alternative relaxation required overriding a single layer-level step (the grouped merge), while the rest of the construction (i.e., the node creation, the priority and merge operators defined in the problem class, and the final pruning) is inherited unchanged. The same pattern can be used to prototype other construction strategies discussed in the literature, such as iterative refinement or separation-based constructions [23,32].

```
class RelaxedGroupingDDBuilder(AbstractDDBuilder):

    def __init__(self, problem, max_width):
        super().__init__(problem)
        self._max_width = max_width

```

```
    def _specific_end_of_layer_function(self):
        # Called after every layer: shrink it if it exceeds the maximum
    width.
        if len(self.graph.structure[-1]) > self._max_width:
            self._merge_nodes_when_width_is_greater_than_w()
            self._delete_nodes_current_layer(self.graph.actual_layer)

    def _specific_end_of_construction_function(self):
        # Called once after the last layer: prune dead-ends and renumber
    nodes.
        self._bottom_up_pruner()
        self.adjust_node_number()
```

Listing 5: Skeleton of the grouping-relaxation builder (abbreviated).

The key difference with the default relaxed builder lies in the end-of-layer step: the grouped merge collapses many nodes in a single pass, which is cheaper on wide layers, at the cost of coarser control over which nodes are merged. All the remaining machinery (node and arc creation, the transition function, the user-defined priority and merge operators, and the final pruning) is reused from `AbstractDDBuilder` without modification.

5.3 Implementing new algorithms

Beyond constructing DDs, many DD-based techniques run algorithms on top of the resulting diagram (e.g., shortest paths for bounds, max-flow for separation, or reduction). DD-suite supports this usage directly: once a DD is built, the `DD` class exposes the underlying graph through `get_decision_diagram()` (and a deep copy through `get_decision_diagram_copy()`), so that any graph algorithm can be implemented against the same `Graph`, `Node`, and `Arc` data structures. The algorithms included in DD-suite are concrete examples of this pattern and live in the `GraphAlgorithms` module: the `ShortestLongestPath` class that computes optimality bounds (Section 4.3), the `MaxFlow` and `MaxFlowGurobi` routines used by the cut generators, and the `DDReducer` that performs the reduction procedure.

A design decision that is particularly convenient for developing new algorithms is the separation between the feasibility set and the reward function (see Section 4.3). Because the DD encodes only the reachable states and feasible assignments, the same graph can be reused to optimize over many different objective functions without

Algorithm 2 Grouping-based Merge of a Node Layer

procedure GROUPMERGEDDNODES($\mathcal{N}_t, W$)
  Sort $\mathcal{N}_t$ in non-decreasing order of the merge priority $\mathsf{p}(\cdot)$
  $\delta \leftarrow 0$ ▷ tolerance: merge only nodes with equal priority
  while $|\mathcal{N}_t| > W$ **do**
    Partition $\mathcal{N}_t$ into maximal groups of consecutive nodes whose priority gap is at most δ
    for each group $\mathcal{G}$ with $|\mathcal{G}| > 1$ and while $|\mathcal{N}_t| > W$ **do**
      Create node $\tilde{n}$ with $\mathsf{s}(\tilde{n}) = \bigoplus_{n \in \mathcal{G}} \mathsf{s}(n)$
      Redirect to $\tilde{n}$ every arc pointing to a node of $\mathcal{G}$, and replace $\mathcal{G}$ by $\tilde{n}$ in $\mathcal{N}_t$
    $\delta \leftarrow \max\{\delta + 1,$ smallest priority gap in $\mathcal{N}_t\}$ ▷ skip passes that merge nothing
  return $\mathcal{N}_t$

rebuilding it. This is exactly what advanced procedures (e.g., Benders decomposition, column generation, and cutting planes) require: a DD is queried repeatedly with changing objectives.

5.4 Example: A cutting plane implementation

To demonstrate how DD-suite serves as a building block for more sophisticated algorithms, we implement a DD-based cutting plane procedure and embed it into a state-of-the-art mixed-integer programming (MIP) solver. This functionality is encapsulated in the `DDCutGenerators` module, which follows the same abstract-class design used throughout the code. The `AbstractCutGenerator` class, shown in Listing 6, defines the interface that every cut generator must implement: given a (fractional) point, `generate_cut()` attempts to separate it and returns whether a violated inequality was found, while `get_cut()` returns the coefficients and right-hand side of the resulting inequality. This uniform interface lets the host solver treat all cut families interchangeably.

```python
class AbstractCutGenerator(ABC):

    def __init__(self, tolerance=1e-4):
        self.tolerance = tolerance

    @abstractmethod
    def generate_cut(self, x_values, verbose=False) -> bool:
        # Separate the point x_values; return True if a violated cut was found.
        ...

    @abstractmethod
    def get_cut(self) -> tuple[list[float], float]:
        # Coefficients and right-hand side of the last separated inequality.
        ...

    @abstractmethod
    def get_name(self) -> str: ...

    @abstractmethod
    def get_time(self) -> float: ...
```

Listing 6: The `AbstractCutGenerator` interface (abbreviated).

DD-suite provides three families of cut generators built on top of a DD, each as a subclass of `AbstractCutGenerator`: combinatorial flow cuts (`FlowCuts`) and their dual max-flow counterpart (`JointFlowCuts`), both based on the combinatorial cut-and-lift procedure of [17], and target cuts (`TargetCut`) following [53]. In addition, the `CutStrengthening` class strengthens a generated inequality following a disjunctive coefficient strengthening strategy that leverages the DD structure [17].

Finally, we integrate these generators into the Gurobi [31] B&B search through a user-cut callback. The `AbstractProblemGurobiClass` sets up the MIP model and, when DD cuts are enabled, registers a callback that, at the root node, retrieves the current fractional solution (`cbGetNodeRel`), invokes the active cut generators over the corresponding DDs, and injects any violated inequality back into the solver (`cbCut`). We emphasize that the entire implementation is built on top of the abstractions described above, with no modification to the DD construction or graph algorithms.

6 Numerical Experiments

This section presents an empirical evaluation of DD-suite with three main objectives. First, we assess the *computational efficiency* of our cross-platform implementation by comparing the construction time of its Python and C++ versions on a large set of instances and benchmarking them against `ddo` [29] (Section 6.2). Second, we showcase the *extensibility* of the code by introducing an alternative relaxed-DD construction mechanism and comparing it against the default one (Section 6.3). Third, we illustrate the *research value* of DD-suite as a building block for DD-based algorithms by implementing a DD-based cutting plane procedure and embedding it into a state-of-the-art MIP solver (Section 6.4). All scripts needed to reproduce the experiments, including data-visualization routines, are provided with our code[4].

6.1 Experimental setup

We now describe the experimental setup common to all three experiments, namely the problems and instances, and the computational environment used throughout.

Problems and instances. We evaluate DD-suite on the five discrete optimization problems included as examples in the code: knapsack, maximum independent set, set cover, sequencing, and SOC knapsack problems. The latter is used only in our cutting plane experiment of Section 6.4. For each problem, we consider up to two families of instances. The *standard* family collects benchmarks taken from the literature: the knapsack instances of [47], the DIMACS clique instances for the independent set problem [37], the SOC knapsack benchmark (SOC-K) of [4] composed of 90 instances, and, for the sequencing problem, 40 instances whose setup time matrices are taken from the ATSP set of TSPLIB [48]. The *custom* family consists of randomly generated instances created using the generators in `DataInstances/`, which allow us to control problem size and difficulty (e.g., number of variables, density, and seed). All instances of the set cover problem are randomly generated. In total, the benchmark comprises 608 instances across all problems and families. Appendix B details each problem, the specific instance sets used, and how they were generated.

Computational environment. All experiments were run on a high-performance computing cluster managed by SLURM, on compute nodes equipped with two Intel® Xeon® E5-2630 v4 processors (2.20 GHz, 20 physical cores) and 755 GB of RAM, running Rocky Linux 9.7. Each instance was solved on a single thread to enable a fair comparison across implementations. Unless stated otherwise, we build approximated DDs with a maximum width $W \in \{500, 1000, 2000, 5000, 10000, 20000\}$ and impose a time limit of five minutes per DD construction; runs exceeding this limit are reported as timeouts. The cutting plane experiments in Section 6.4 use a one-hour time limit, a common choice in the MIP literature. We use Gurobi 13 [31] both as a baseline MIP solver and as the host solver for our cutting plane procedure.

[4] GitHub repository: https://github.com/MargaritaCastro/dd-suite

6.2 Cross-language efficiency of DD-suite

Our first experiment evaluates the computational performance of the two DD-suite implementations and positions them relative to an existing high-performance tool. For every instance and DD type (exact, restricted, and relaxed), we construct the corresponding diagram in Python and C++ and measure its construction time. The two DD-suite implementations share the same algorithms and produce identical diagrams and optimality bounds; thus, comparing them isolates the overhead of the programming language rather than algorithmic differences. We verified this equivalence by checking that the optimization value, the number of nodes, and the number of arcs coincide across both languages on every configuration, for all DD types, and that the value obtained from the exact DDs matches the optimal value reported by Gurobi (see Appendix C). We further benchmark DD-suite against `ddo` [29], a state-of-the-art compiled (Rust) framework for DD-based optimization, to gauge how our implementation compares against a specialized, performance-oriented tool.

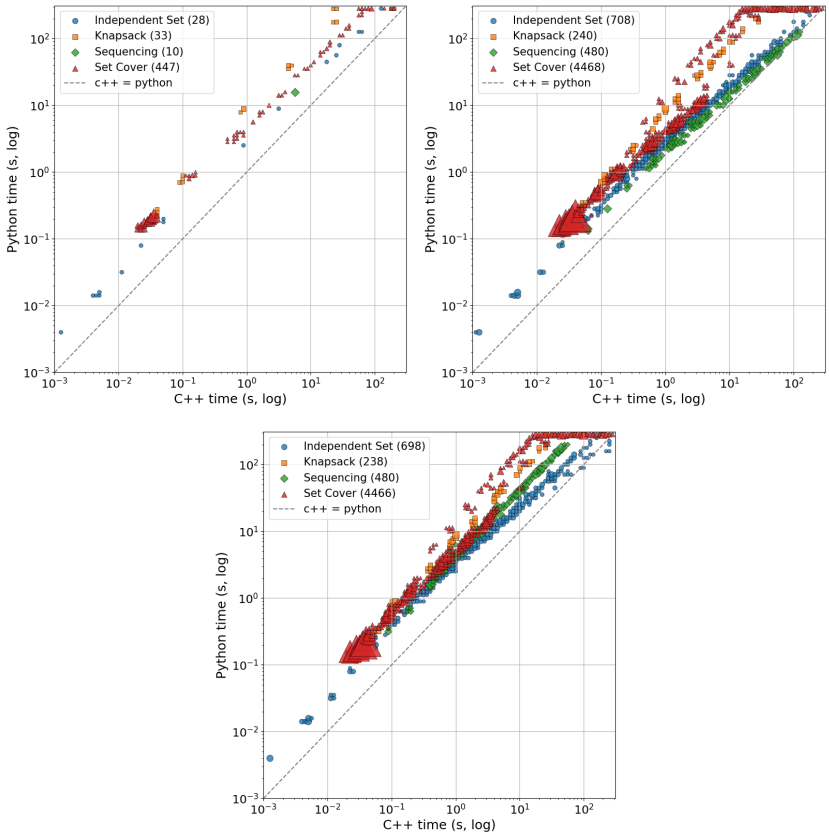


Fig. 4: C++ versus Python construction time, per DD type.

Table 2: Average ratio of construction times per DD type.

Comparison	DD type	Average construction-time ratio
Python / C++	Exact	5.9
Python / C++	Restricted	5.2
Python / C++	Relaxed	5.7
C++ / `ddo`	Exact	2.1
C++ / `ddo`	Restricted	2.3
C++ / `ddo`	Relaxed	1.8

Figure 4 compares the construction time of C++ against Python on a logarithmic scale, with one panel per DD type (exact, restricted, and relaxed) and colors denoting the problem type. To avoid overlaps, instances are binned by their (rounded) construction times, so each marker aggregates all instances that fall on approximately the same coordinate; the size of a marker is proportional to the number of instances it represents (i.e., a larger marker indicates that more instances accumulate at that position). Since the x-axis corresponds to C++ and the y-axis to Python, points above the diagonal indicate instances for which C++ is faster. The C++ implementation is consistently faster than the Python implementation across all problems and DD types, with an average speedup of 5 to 6 times. Table 2 summarizes these results, reporting the average construction-time ratio for each DD type, aggregated over all problems and widths, where a construction-time ratio of k means that the slower implementation takes k times longer.

Figure 5 compares DD-suite's C++ implementation against `ddo`, with one panel per DD type and points below the diagonal indicating that `ddo` is faster, following the same style as in Figure 4. We can see that `ddo` is faster than our C++ implementation with a factor of approximately 2 for all DD types on average (see Table 2). This gap is expected, as `ddo` is a specialized framework written in Rust and heavily optimized for raw performance, whereas DD-suite prioritizes a clean, extensible, and cross-platform code base. Specifically, DD-suite includes additional data structures for manipulating the underlying graph (e.g., both incoming and outgoing arcs), which makes it amenable to algorithm development but compromises its speed. In contrast, `ddo` solely focuses on B&B optimization, so it only includes the necessary structures to favor speed (e.g., only outgoing arcs).

6.3 Extending DD-suite: an alternative relaxation mechanism

This experiment illustrates how DD-suite can be extended within the code with minimal effort. As described in Section 4, the default relaxed-DD construction merges node pairs following a user-defined priority. We complement it with an alternative routine that merges nodes in groups, which only required overriding a single construction step, as explained in Section 5.2.

Figure 6a compares the two relaxation mechanisms in terms of construction time, with the priority mechanism on the x-axis and the grouping mechanism on the y-axis (points below the diagonal indicate that grouping is faster). The grouping variant

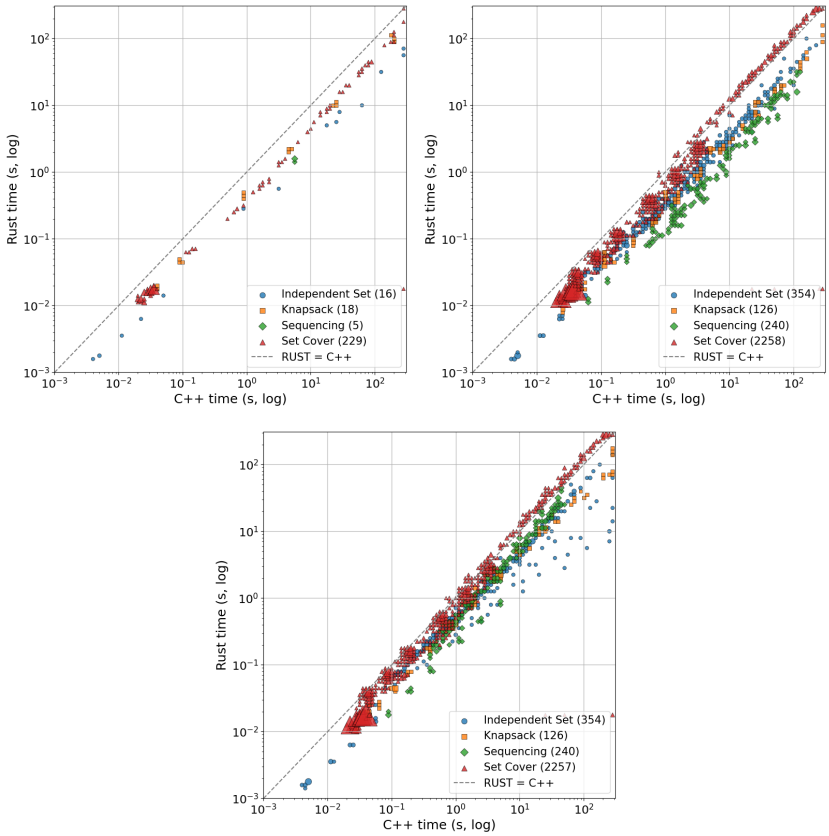


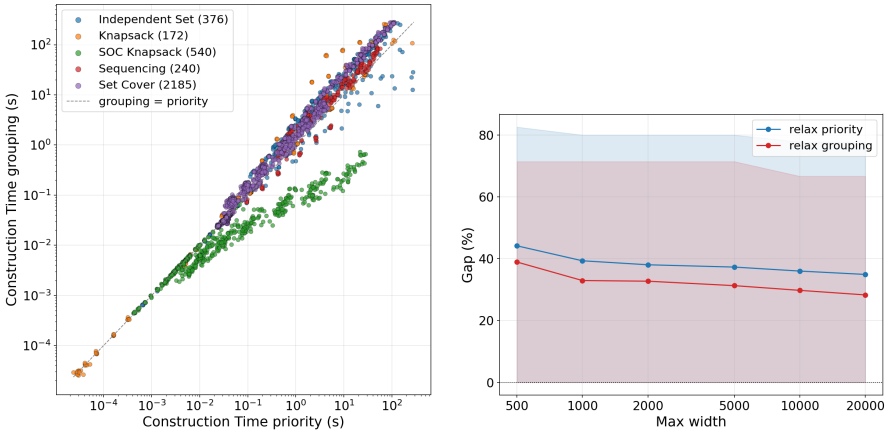
Fig. 5: C++ versus `ddo` construction time, per DD type.

merges several nodes at once and is therefore especially effective on wide layers. While the two mechanisms have comparable construction times on most instances, grouping is faster on average (with an average construction-time ratio of about 1.8), with the largest gains concentrated on the instances and widths that produce the widest layers.

Figure 6b compares the two mechanisms in terms of bound quality, measured as the optimality gap of the relaxed bound relative to the exact optimum. For each maximum width, the figure reports the average gap (after removing outliers using the IQR rule) and the interquartile range (shaded band) across the set-cover instances for both mechanisms. The two mechanisms yield identical bounds for roughly 60% of the (instance, width) pairs and are comparable on the remainder, with neither dominating the other. The analogous gap plots for all problems and for both the C++ and Python implementations are collected in Appendix D.

We stress that these two construction heuristics were not heavily engineered: our goal is to illustrate that alternative construction mechanisms can be prototyped within DD-suite with minimal effort and that they lead to observable differences in construc-

tion time and bound quality, rather than to identify the best possible relaxation. A thorough study of which construction strategy is preferable, and under which conditions, is an interesting future work direction. Overall, the grouping mechanism offers a useful speed–quality trade-off that users can consider.



(a) Construction time comparison.

(b) Gap comparison for set cover instances.

Fig. 6: Time and gap comparison between priority and grouping relaxed DDs.

6.4 DD-based cutting planes

We now illustrate how DD-suite can be used as a component of more sophisticated optimization algorithms and, in particular, how it enables reproducing results from the literature. Concretely, we recreate the 0–1 second-order conic (SOC) programming experiment of Castro, Cire, and Beck [17], who introduced the combinatorial cut-and-lift procedure that underlies our *FlowCuts* and *JointFlowCuts* generators and their strengthening; we additionally evaluate the target cuts of Tjandraatmadja and van Hoeve [53] as a third cut family on the same benchmark.

Concretely, we implement a DD-based cutting plane procedure and embed it into the Gurobi B&B search through a user-cut callback at the root node (see Section 5.4). Following the setup of [17], for each SOC knapsack instance we build one relaxed DD per SOC constraint and, given a fractional point, iterate over the DDs to separate valid inequalities that cut off the point; enabling Gurobi's `PreCrush` parameter ensures that the resulting user cuts are applied to the presolved model. We evaluate three families of cuts available in DD-suite: combinatorial flow cuts (*FlowCuts*), dual flow cuts derived from a max-flow formulation (*JointFlowCuts*), and target cuts (*TargetCuts*). For each family, we consider a plain variant and a strengthened variant (+*Strengthening*) that lifts every generated inequality into a tighter one. Note that we omit the TargetCuts+Strengthening alternative since TargetCuts are known to yield

facet-defining inequalities for this application [53]. As a baseline, we run Gurobi with its default configuration.

Table 3 reports aggregated results over the SOC-K benchmark for both the C++ and Python implementations, under a one-hour time limit. For each variant, we report the number of instances solved to optimality (# Solv), the average final MIP gap over the unsolved instances (Gap), the average solution time over the solved instances (Time), the average number of B&B nodes explored (# Nodes), and the average number of cuts generated (# Cuts). Averages are taken over the instances solved by both implementations to ensure a fair comparison. In each column and for each implementation, the best values are highlighted in bold.

Table 3: Aggregated results on the SOC-K benchmark.

	C++					Python				
Method	# Solv	Gap	Time	# Nodes	# Cuts	# Solv	Gap	Time	# Nodes	# Cuts
Gurobi	81	0.68%	423.2	66,363	–	81	0.67%	376.0	63,819	–
FlowCuts	83	0.73%	417.1	41,492	66	83	0.72%	393.5	40,847	66
+Strengthening	84	0.62%	417.3	39,648	47	84	0.62%	418.3	39,658	46
JointFlowCuts	82	0.61%	333.1	30,883	174	**86**	0.61%	331.6	30,877	174
+Strengthening	84	**0.48%**	295.6	30,951	112	85	**0.47%**	299.1	31,142	112
TargetCuts	**85**	0.54%	**267.9**	**29,299**	82	**86**	0.48%	**294.6**	**28,127**	83

The results show that all DD-based cut families improve upon the plain Gurobi baseline, which solves 81 of the 90 instances. Adding DD cuts increases the number of solved instances (up to 86) and substantially reduces the size of the search tree: the number of B&B nodes drops from roughly 66,000 for Gurobi to about 30,000 for the JointFlowCuts and TargetCuts variants, a reduction of more than 50%. The strengthening procedure further improves performance: for FlowCuts it reduces both the final gap (from 0.73% to 0.62%) and the number of cuts needed (from 66 to 47), while for JointFlowCuts it attains the smallest final gaps of the whole table (0.48% in C++ and 0.47% in Python). TargetCuts, in turn, solves the largest number of instances (85 in C++ and 86 in Python) and yields the smallest average solution time (268 seconds in C++). Across the board, the JointFlowCuts and TargetCuts variants are the strongest, attaining the smallest gaps (0.47%–0.61%) and the smallest search trees. The C++ and Python implementations produce very similar results, as expected from their shared algorithmic core, with small differences attributable to solver-internal timing and tie-breaking.

We note that these results and conclusions are consistent with the ones obtained in [17], despite the latter using a specialized DD implementation in C++ with an older commercial solver (i.e., IBM ILOG CPLEX 12.9). Overall, this experiment demonstrates that DD-suite is not only a tool for building and querying DDs in isolation but also a practical foundation for developing competitive DD-based algorithms.

7 Conclusions and Future Work

This work introduced *DD-suite*, a cross-platform, open-source code for building and manipulating decision diagrams (DDs) for discrete optimization. Unlike existing DD codes, which are typically tailored to a specific algorithm or application, DD-suite is designed to be clean, intuitive, and easily extended, providing the same modeling interface in both Python and C++. The suite supports the three standard DD types (exact, restricted, and relaxed), along with the DD reduction procedure, shortest-path routines for obtaining optimality bounds, a visualization tool, and extensive automated testing. Moreover, it includes thorough documentation, a support webpage, and ready-to-use examples for four combinatorial problems, as well as a SOC knapsack example used in our cutting plane experiments. Our goal is to lower the entry barrier to DD-based research, so that researchers can prototype and develop novel DD-based algorithms without reimplementing the underlying machinery.

Our numerical experiments support this design. The C++ and Python implementations build identical diagrams and bounds, with C++ being 5–6 times faster and remaining within a small constant factor (about 2 times) of `ddo` (i.e., a specialized, performance-oriented Rust framework). We further showed that DD-suite is easy to extend by adding an alternative grouping-based relaxation mechanism through a single overridden construction step, and that it serves as a practical building block for more sophisticated algorithms by implementing DD-based cutting planes that improve a state-of-the-art commercial solver on the SOC knapsack benchmark.

There are several promising directions for future work. A natural extension is to support additional DD types, such as zero-suppressed DDs, which are well suited to binary problems like the maximum independent set problem. Another is to incorporate additional DD construction mechanisms, such as iterative refinement or separation, which yield stronger relaxations and serve as building blocks for the column elimination algorithm. By releasing DD-suite as an open-source, well-documented, and extensively tested package, we hope to make DD-based optimization more accessible and to encourage its adoption by a broader research community.

Acknowledgements This research was partially supported by the National Center for Artificial Intelligence CENIA FB210017 (Basal ANID), and the *Agencia Nacional de Investigación y Desarrollo* of Chile under grants ANID-FONDECYT-11230797 and ANID-FONDECYT-1261833.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall (1993)
2. Akers, S.B.: Binary decision diagrams. IEEE Transactions on computers **C-27**(6), 509–516 (1978)

3. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13, pp. 118–132. Springer (2007)
4. Atamtürk, A., Narayanan, V.: Conic mixed-integer rounding cuts. Mathematical Programming **122**(1), 1–20 (2010)
5. Becker, B., Behle, M., Eisenbrand, F., Wimmer, R.: BDDs in a branch and cut framework. In: S.E. Nikoletseas (ed.) Proceedings of the International Workshop on Experimental and Efficient Algorithms, pp. 452–463. Springer, Berlin, Heidelberg (2005)
6. Behle, M.: Binary decision diagrams and integer programming. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2007)
7. Bellman, R.E.: Dynamic Programming. Princeton University Press, Princeton, NJ (1957)
8. Bergman, D., Cire, A.A., van Hoeve, W.J.: Lagrangian bounds from decision diagrams. Constraints **20**(3), 346–361 (2015)
9. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Discrete optimization with decision diagrams. INFORMS Journal on Computing **28**(1), 47–66 (2016)
10. Bergman, D., Cire, A.A., van Hoeve, W.J., Yunes, T.: BDD-based heuristics for binary optimization. Journal of Heuristics **20**(2), 211–234 (2014)
11. Bergman, D., Cire, A.A., Hoeve, W.J.v., Hooker, J.N.: Optimization bounds from binary decision diagrams. INFORMS Journal on Computing **26**(2), 253–268 (2014)
12. Bergman, D., Cire, A.A., Van Hoeve, W.J., Hooker, J.: Decision diagrams for optimization, vol. 1. Springer (2016)
13. Bergman, D., van Hoeve, W.J., Hooker, J.N.: Manipulating MDD relaxations for combinatorial optimization. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2011), *Lecture Notes in Computer Science*, vol. 6697, pp. 20–35. Springer (2011)
14. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **100**(8), 677–691 (1986)
15. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992)
16. Castro, M.P., Cire, A.A., Beck, J.C.: An MDD-based Lagrangian approach to the multi-commodity pickup-and-delivery TSP. INFORMS Journal on Computing **32**(2), 263–278 (2020)
17. Castro, M.P., Cire, A.A., Beck, J.C.: A combinatorial cut-and-lift procedure with an application to 0–1 second-order conic programming. Mathematical Programming **196**(1), 115–171 (2022)
18. Castro, M.P., Cire, A.A., Beck, J.C.: Decision diagrams for discrete optimization: A survey of recent advances. INFORMS Journal on Computing **34**(4), 2271–2295 (2022)
19. Castro, M.P., Piacentini, C., Cire, A.A., Beck, J.C.: Relaxed BDDs: An admissible heuristic for delete-free planning based on a discrete relaxation. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 77–85 (2019)
20. Castro, M.P., Piacentini, C., Cire, A.A., Beck, J.C.: Solving delete free planning with relaxed decision diagram based heuristics. Journal of Artificial Intelligence Research **67**(1), 607–651 (2020)
21. Cire, A.A., Diamant, A., Yunes, T., Carrasco, A.: A network-based formulation for scheduling clinical rotations. Production and Operations Management **28**(5), 1186–1205 (2019)
22. Cire, A.A., van Hoeve, W.J.: Multivalued decision diagrams for sequencing problems. Operations Research **61**(6), 1411–1428 (2013)
23. Ciré, A.A., Hooker, J.N.: The separation problem for binary decision diagrams. In: ISAIM (2014)
24. Coppé, V., Gillard, X., Schaus, P.: Decision diagram-based branch-and-bound with caching for dominance and suboptimality detection. INFORMS Journal on Computing (2024)
25. Davarnia, D., van Hoeve, W.J.: Outer approximation for integer nonlinear programs via decision diagrams. Mathematical Programming **187**(1), 111–150 (2021)
26. Gentzel, R., Michel, L., van Hoeve, W.J.: Haddock: A language and architecture for decision diagram compilation. In: Principles and Practice of Constraint Programming: 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7–11, 2020, Proceedings 26, pp. 531–547. Springer (2020)
27. Gentzel, R., Michel, L., van Hoeve, W.J.: Optimization bounds from decision diagrams in haddock. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 150–166. Springer (2023)

28. Gillard, X., Coppé, V., Schaus, P., Cire, A.A.: Improving the filtering of branch-and-bound mdd solver. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 231–247. Springer (2021)
29. Gillard, X., Schaus, P., Coppé, V.: DDO, a generic and efficient framework for MDD-based optimization. In: International Joint Conference on Artificial Intelligence (IJCAI-20) (2020)
30. Guo, C., Bodur, M., Aleman, D.M., Urbach, D.R.: Logic-based Benders decomposition and binary decision diagram based approaches for stochastic distributed operating room scheduling. INFORMS Journal on Computing **33**(4), 1551–1569 (2021)
31. Gurobi Optimization, LLC: Gurobi optimizer reference manual. https://www.gurobi.com (2024)
32. van Hoeve, W.J.: Graph coloring with decision diagrams. Mathematical Programming **192**(1), 631–674 (2022)
33. van Hoeve, W.J.: An introduction to decision diagrams for optimization. INFORMS TutORials in Operations Research pp. 1–28 (2024)
34. Hooker, J.N.: Decision diagrams and dynamic programming. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings 10, pp. 94–110. Springer (2013)
35. Hooker, J.N.: Job sequencing bounds from decision diagrams. In: Principles and Practice of Constraint Programming: 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 23, pp. 565–578. Springer (2017)
36. IBM: IBM ILOG CPLEX optimization studio. https://www.ibm.com/products/ilog-cplex-optimization-studio
37. Johnson, D.S., Trick, M.A. (eds.): Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 26. American Mathematical Society (1996)
38. Karahalios, A., van Hoeve, W.J.: Column elimination for capacitated vehicle routing problems. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 35–51. Springer (2023)
39. Kinable, J., Cire, A.A., van Hoeve, W.J.: Hybrid optimization methods for time-dependent sequencing problems. European Journal of Operational Research **259**(3), 887–897 (2017)
40. Kuroiwa, R., Beck, J.C.: Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), vol. 33, pp. 236–244 (2023)
41. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell System Technical Journal **38**(4), 985–999 (1959)
42. Lozano, L., Smith, J.C.: A binary decision diagram based algorithm for solving a class of binary two-stage stochastic programs. Mathematical Programming **191**(1), 381—-404 (2022)
43. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2008)
44. Michel, L., van Hoeve, W.J.: Codd: A decision diagram-based solver for combinatorial optimization. In: ECAI 2024, pp. 4240–4247. IOS Press (2024)
45. Morrison, D.R., Sewell, E.C., Jacobson, S.H.: Solving the pricing problem in a branch-and-price algorithm for graph coloring using zero-suppressed binary decision diagrams. INFORMS Journal on Computing **28**(1), 67–82 (2016)
46. Perez, G., Régin, J.C.: Efficient operations on mdds for building constraint programming models. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI), pp. 374–380 (2015)
47. Pisinger, D.: Where are the hard knapsack problems? Computers & Operations Research **32**(9), 2271–2284 (2005)
48. Reinelt, G.: TSPLIB—a traveling salesman problem library. ORSA Journal on Computing **3**(4), 376–384 (1991)
49. Riascos-Álvarez, L.C., Bodur, M., Aleman, D.M.: A branch-and-price algorithm enhanced by decision diagrams for the kidney exchange problem. Manufacturing & Service Operations Management **26**(2), 485–499 (2024)
50. Rudich, I., Cappart, Q., Rousseau, L.M.: Improved peel-and-bound: Methods for generating dual bounds with multivalued decision diagrams. Journal of Artificial Intelligence Research **77**, 1489–1538 (2023)
51. Salemi, H., Davarnia, D.: On the structure of decision diagram–representable mixed-integer programs with application to unit commitment. Operations Research **71**(6), 1943–1959 (2023)

52. Tardivo, F., Michel, L., van Hoeve, W.J.: Complete anytime decision diagram search with GPU-accelerated state expansion. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR). Springer (2026)
53. Tjandraatmadja, C., van Hoeve, W.J.: Target cuts from relaxed decision diagrams. INFORMS Journal on Computing **31**(2), 285–301 (2019)
54. Vasquez, S., Lozano, L., van Hoeve, W.J.: A single-level reformulation of binary bilevel programs using decision diagrams. Mathematical Programming (2025)

A DD-suite Python code for our running example

```python
from SourceCode.Problems.AbstractProblemClass import AbstractProblem


class KnapsackProblem(AbstractProblem):

    def __init__(self, params: 'KnapsackStructure'):
        super().__init__(params.initial_state, params.variables)
        self.weights: list[int] = params.weights
        self.capacity: int = params.right_side_of_restrictions

    def transition_function(self, previous_state: 'State', variable_index: int, variable_value: int, scratch_state: list) -> bool:

        if variable_value == 0:
            scratch_state[0] = previous_state
            return True

        new_state: int = previous_state + self.weights[variable_index] * variable_value
        scratch_state[0] = new_state
        return new_state <= self.capacity

    def get_priority_for_discard_node(self, state: 'State') -> int:
        return -state

    def get_priority_for_merge_nodes(self, id: int, state: 'State') -> int:
        return -state

    def merge_operator(self, state_one: 'State', state_two: 'State') -> 'State':
        return min(state_one, state_two)

    def get_state_as_string(self, state: 'State') -> str:
        return str(state)

    def get_state_copy(self, state: 'State') -> 'State':
        return state
```

Listing 7: Complete knapsack problem class implementation in Python

```python
from types import SimpleNamespace

from SourceCode.DD import DD
from Examples.KnapsackInstance.KnapsackProblem import KnapsackProblem
from SourceCode.GraphAlgorithms.ShortestLongestPath.ShortestLongestPath import ShortestLongestPath

## ============================
## SETUP PROBLEM AND PARAMETERS
## ============================

# Maximum width for the approximated DDs
width: int = 2

# Input data for the running example
params = SimpleNamespace(
    initial_state=0,
    variables=[('x_1', [0, 1]), ('x_2', [0, 1]), ('x_3', [0, 1]), ('x_4', [0, 1])],
    weights=[7, 5, 4, 1],
    right_side_of_restrictions=8,
)

# Objective coefficients and sense used later to optimize over the DDs
utility: list[int] = [4, 2, 5, 1]
```

```
objective: str = "max"

# Create the knapsack problem used to build the DD
knapsack_problem = KnapsackProblem(params)

## ============================
## CREATE EXACT DD
## ============================

dd_exact = DD(knapsack_problem)
dd_exact.create_decision_diagram()
dd_exact.reduce_decision_diagram()

# Print DD construction information
print("-- Exact DD information --")
print("\tConstruction time (sec): ", dd_exact.get_building_time())
print("\tReduction time (sec): ", dd_exact.get_reduction_time())
print("\tNumber of nodes: ", dd_exact.get_decision_diagram().get_node_count
    ())
print("\tNumber of arcs: ", dd_exact.get_decision_diagram().get_arc_count()
    )

## ============================
## CREATE RELAXED DD
## ============================

dd_relaxed = DD(knapsack_problem)
dd_relaxed.create_relax_priority_decision_diagram(max_width=width)
dd_relaxed.reduce_decision_diagram()

print("-- Relaxed DD information --")
print("\tConstruction time (sec): ", dd_relaxed.get_building_time())
print("\tReduction time (sec): ", dd_relaxed.get_reduction_time())
print("\tNumber of nodes: ", dd_relaxed.get_decision_diagram().
    get_node_count())
print("\tNumber of arcs: ", dd_relaxed.get_decision_diagram().get_arc_count
    ())

## ============================
## CREATE RESTRICTED DD
## ============================

dd_restricted = DD(knapsack_problem)
dd_restricted.create_restricted_decision_diagram(max_width=width)
dd_restricted.reduce_decision_diagram()

print("-- Restricted DD information --")
print("\tConstruction time (sec): ", dd_restricted.get_building_time())
print("\tReduction time (sec): ", dd_restricted.get_reduction_time())
print("\tNumber of nodes: ", dd_restricted.get_decision_diagram().
    get_node_count())
print("\tNumber of arcs: ", dd_restricted.get_decision_diagram().
    get_arc_count())

# Export graph in .gml format
dd_exact.export_graph_file("knapsack_exact_dd_file")
dd_relaxed.export_graph_file("knapsack_relaxed_dd_file")
dd_restricted.export_graph_file("knapsack_restricted_dd_file")

## ============================
## OPTIMIZATION: PRIMAL AND DUAL BOUNDS
## ============================

# Optimal solution over the exact DD (longest path)
longest_path = ShortestLongestPath(dd_exact)
longest_path.set_parameters(utility, objective)
answer = longest_path.solve()

```

```
print("The optimal value is: ", answer.value)
print("Longest path: ", answer.path_print)
print("Solution time (sec): ", longest_path.get_time())

# Dual (upper) bound over the relaxed DD
longest_path_relaxed = ShortestLongestPath(dd_relaxed)
longest_path_relaxed.set_parameters(utility, objective)
answer_relaxed = longest_path_relaxed.solve()

print("The dual (upper) bound is: ", answer_relaxed.value)
print("Longest path over relaxed DD: ", answer_relaxed.path_print)
print("Solution time (sec): ", longest_path_relaxed.get_time())

# Primal (lower) bound over the restricted DD
longest_path_restricted = ShortestLongestPath(dd_restricted)
longest_path_restricted.set_parameters(utility, objective)
answer_restricted = longest_path_restricted.solve()

print("The primal (lower) bound is: ", answer_restricted.value)
print("Longest path over restricted DD: ", answer_restricted.path_print)
print("Solution time (sec): ", longest_path_restricted.get_time())
```

Listing 8: Complete DD construction and optimization code

B Problems and data instances

This appendix details, for each problem, the specific instance set used in the experiments of Section 6. For each problem, we first state the classical integer programming (IP) formulation and then describe the instances themselves. In every case the recursive model assigns one variable per stage, so the DD horizon equals the number of variables (i.e., $T = n$).

For knapsack, independent set, sequencing, and SOC knapsack we use established benchmarks from the literature; the sequencing instances are derived from a standard benchmark by the conversion described in Section B.4. For set cover, no standard benchmark fits our setting, so we generate the instances ourselves and report the generator below. Wherever random draws are involved, they use Python's `random` module (`numpy.random` for set cover) seeded with the value recorded in the file name, so every instance is reproducible from the listings. All the instances, together with the generation and conversion scripts, are available in the `DataInstances/` directory of the GitHub repository.

B.1 Knapsack

The 0–1 *knapsack problem* asks for a maximum-reward subset of n items subject to a single capacity constraint. Letting $x_i = 1$ if item i is placed in the knapsack, $c_i \in \mathbb{Z}$ its reward, $w_i \in \mathbb{Z}_{>0}$ its weight, and $b \in \mathbb{Z}_{>0}$ the capacity, the problem reads

$$\begin{aligned} \max \ & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.} \ & \sum_{i=1}^{n} w_i x_i \le b, \\ & \boldsymbol{x} \in \{0,1\}^n. \end{aligned} \tag{2}$$

The recursive formulation is that of Example 1: the state is the current load of the knapsack, and the resulting DD is binary.

We use the knapsack benchmark of [47], which comprises the 30 instances listed in Table 4 and combines two groups. The nine *low-dimensional* instances (prefix `f`) are small on purpose: the number of items (i.e., binary variables) ranges from 4 up to 23, so that the exact DD can be constructed in full. The 21 *large-scale* instances (prefix `knapPI`) cover three instance types and seven sizes, from 100 up to 10,000 items, and are used to stress the restricted and relaxed constructions well beyond the low-dimensional regime.

Table 4: Knapsack instances [47].

Low-dimensional		
`f1_l-d_kp_10_269`	`f2_l-d_kp_20_878`	`f3_l-d_kp_4_20`
`f4_l-d_kp_4_11`	`f6_l-d_kp_10_60`	`f7_l-d_kp_7_50`
`f8_l-d_kp_23_10000`	`f9_l-d_kp_5_80`	`f10_l-d_kp_20_879`
Large-scale		
`knapPI_1_100_1000_1`	`knapPI_2_100_1000_1`	`knapPI_3_100_1000_1`
`knapPI_1_200_1000_1`	`knapPI_2_200_1000_1`	`knapPI_3_200_1000_1`
`knapPI_1_500_1000_1`	`knapPI_2_500_1000_1`	`knapPI_3_500_1000_1`
`knapPI_1_1000_1000_1`	`knapPI_2_1000_1000_1`	`knapPI_3_1000_1000_1`
`knapPI_1_2000_1000_1`	`knapPI_2_2000_1000_1`	`knapPI_3_2000_1000_1`
`knapPI_1_5000_1000_1`	`knapPI_2_5000_1000_1`	`knapPI_3_5000_1000_1`
`knapPI_1_10000_1000_1`	`knapPI_2_10000_1000_1`	`knapPI_3_10000_1000_1`

B.2 Independent set

Given an undirected graph $G = (V, E)$ with vertex rewards $c_i \in \mathbb{Z}$, the *maximum weight independent set problem* looks for a set of pairwise non-adjacent vertices of maximum total reward. Using $x_i = 1$ to indicate that vertex i is selected, the formulation is

$$\begin{aligned} \max \ & \sum_{i \in V} c_i x_i \\ \text{s.t. } & x_i + x_j \leq 1, \qquad \forall \{i, j\} \in E, \\ & \boldsymbol{x} \in \{0,1\}^{|V|}. \end{aligned} \tag{3}$$

The problem is also known as the *maximum stable set* problem, and its unweighted version ($c_i = 1$ for all i) as the maximum independent set problem (MISP). It is equivalent to finding a maximum clique in the complement graph $\bar{G}$ and complementary to the minimum vertex cover problem, so care is needed when comparing objective values reported in the literature. In our recursive model, stage t decides x_t and the state records the set of vertices that are still eligible, which yields a binary DD with $n = |V|$ stages.

We use the DIMACS maximum-clique benchmark [37], a widely used set of graphs whose sizes range from a few dozen up to several hundred vertices. Since the maximum independent set of G is a maximum clique of $\bar{G}$, the repository stores the *complement* of each DIMACS graph and solves (3) on it directly; for example, `MANN_a9` has 45 vertices and 918 edges in DIMACS format and is stored with the 72 edges of its complement. All instances are unweighted ($c_i = 1$), so the reported objective is the cardinality of the largest independent set, which coincides with the DIMACS maximum-clique number. We use the 64 graphs listed in Table 5, spanning the `MANN`, `brock`, `c-fat`, `hamming`, `johnson`, `keller`, `p_hat`, `san`, and `sanr` families.

B.3 Set cover

Let $A \in \{0,1\}^{m \times n}$ be a 0–1 matrix whose n columns index candidate sets and whose m rows index the elements to be covered, and let $c_j \in \mathbb{Z}_{>0}$ be the cost of selecting column j. The *set covering problem* selects a minimum-cost family of columns covering every row:

$$\begin{aligned} \min \ & \sum_{j=1}^{n} c_j x_j \\ \text{s.t. } & \sum_{j=1}^{n} A_{rj} x_j \geq 1, \qquad r = 1, \ldots, m, \\ & \boldsymbol{x} \in \{0,1\}^{n}. \end{aligned} \tag{4}$$

Table 5: Independent set instances (DIMACS [37]).

brock200_1	c-fat500-10	MANN_a45	p_hat700-3
brock200_2	c-fat500-2	MANN_a9	san1000
brock200_3	c-fat500-5	p_hat1000-1	san200_0.7_1
brock200_4	hamming10-2	p_hat1000-2	san200_0.7_2
brock400_1	hamming10-4	p_hat1000-3	san200_0.9_1
brock400_2	hamming6-2	p_hat1500-1	san200_0.9_2
brock400_3	hamming6-4	p_hat1500-2	san200_0.9_3
brock400_4	hamming8-2	p_hat1500-3	san400_0.5_1
brock800_1	hamming8-4	p_hat300-1	san400_0.7_1
brock800_2	johnson16-2-4	p_hat300-2	san400_0.7_2
brock800_3	johnson32-2-4	p_hat300-3	san400_0.7_3
brock800_4	johnson8-2-4	p_hat500-1	san400_0.9_1
c-fat200-1	johnson8-4-4	p_hat500-2	sanr200_0.7
c-fat200-2	keller4	p_hat500-3	sanr200_0.9
c-fat200-5	keller5	p_hat700-1	sanr400_0.5
c-fat500-1	MANN_a27	p_hat700-2	sanr400_0.7

When all costs are equal the problem reduces to minimum cardinality set cover, which is the case for every instance we use. Stage t decides x_t and the state records which rows remain uncovered, giving a binary DD with n stages.

All the instances for this problem are generated by us, following the structured scheme of [13] implemented in Listing 9. The key property of this scheme is that the matrix is *banded*: row i may only be covered by columns in the window $[i, \min(i+\beta, n))$, where β is the bandwidth. Within that window the generator selects $k = \lfloor d\,n \rfloor$ columns uniformly without replacement (or the whole window when it contains fewer than k columns) and sets those entries to one. The band structure is what makes these instances suitable for DDs: because a row can only interact with the β columns that follow it, the number of rows that are simultaneously "open" at any stage is bounded, which keeps the exact DD width manageable and makes the effect of the maximum width W interpretable. All instances use unit costs, $c_j = 1$.

Instances follow the convention `set_cover_n{n}_m{m}_d{d}_b_w{β}_seed{S}`. As summarized in Table 6, they form two families, for a total of 384 instances. The *large* family fixes the bandwidth at $\beta = 165$ and sets $m = n - \beta + 1$, sweeping n over the 16 values $\{250, 500, \ldots, 4000\}$ in steps of 250. Here the density is not a free parameter: the generator sets $d = 75/n$, so that every row contains exactly $k = 75$ ones and the row density decreases as n grows (this is why the file names show d ranging from 0.30 down to 0.02). The *small* family fixes $n = 100$ and $m = 78$ and instead sweeps the bandwidth over $\beta \in \{17, 20, \ldots, 38\}$ and the density over $d \in \{0.05, 0.10, \ldots, 0.50\}$, which isolates the effect of the bandwidth on the diagram size. Both families use seeds $S \in \{1, 2, 3, 4\}$.

```python
import numpy as np

def set_cover_matrix_generator(n: int, m: int, d: float, bw: int, seed: int
    =1) -> np.ndarray:
    """
    Generate an m x n matrix A for a structured Set Cover instance.

    Parameters:
    - n: number of columns (variables).
    - m: number of rows (constraints).
    - d: density of ones per row (a proportion between 0 and 1).
    - bw: bandwidth (how many columns ahead can be chosen).

    Returns:
    - A: binary matrix of size m x n.
    """
    A = np.zeros((m, n), dtype=int)
    k = int(d * n)  # number of ones per row
    np.random.seed(seed)

```

```
    for i in range(m):
        start = i
        end = min(i + bw, n)  # avoid exceeding n
        candidates = list(range(start, end))

        if len(candidates) == 0:
            continue  # there are no valid columns for this row

        selection = candidates if len(candidates) <= k else np.random.choice(candidates, size=k, replace=False)
        A[i, selection] = 1

    return A

n_options = [250, 500, 750, 1000, 1250, 1500, 1750, 2000, 2250, 2500, 2750, 3000, 3250, 3500, 3750, 4000]
d_options = [0.1]
b_w_options = [165]

for seed in range(1, 5):
    for n in n_options:
        for d in d_options:
            d = 75 / n  # fixes the number of ones per row at k = int(d * n) = 75
            for b_w in b_w_options:
                m = n - b_w + 1
                custom_folder = './Standard/'
                file_name = f'set_cover_n{n}_m{m}_d{round(d,2)}_b_w{b_w}_seed{seed}.txt'

                objective_weights = [1 for _ in range(int(n))]
                matrix_of_weight = set_cover_matrix_generator(n, m, d, b_w, seed)

                print(f"n: {n}, m: {m}, d: {d}, b_w: {b_w}")
                print(matrix_of_weight)

                with open(custom_folder + file_name, 'w') as file:
                    file.write(str(n) + '\n')
                    file.write(str(m) + '\n')
                    file.write(' '.join(map(str, objective_weights)) + '\n')
                    for i, row in enumerate(matrix_of_weight):
                        file.write(' '.join(map(str, row)))
                        if i != len(matrix_of_weight) - 1:
                            file.write('\n')
```

Listing 9: Structured (banded) set cover instance generator.

Table 6: Structured set cover instance families.

	Large family	Small family
n (columns)	$250, 500, \ldots, 4000$	100
m (rows)	$n - \beta + 1$	78
Bandwidth β	165	$17, 20, \ldots, 38$
Density d	$75/n$	$0.05, 0.10, \ldots, 0.50$
Ones per row k	75	$\lfloor dn \rfloor$
Costs c_j	1	1
Seeds	$1, \ldots, 4$	$1, \ldots, 4$
# configurations	16	80
# instances	64	320

B.4 Sequencing

We consider the single-machine sequencing problem with sequence-dependent setup times and total weighted completion time objective, denoted $1\,|\,s_{ij}\,|\,\sum_j w_j C_j$ in the standard scheduling notation. A set of n jobs $J = \{1,\dots,n\}$ must be processed one at a time without preemption; job $j \in J$ has processing time p_j and weight w_j, and a setup time s_{ij} elapses when job j is started immediately after job $i \in J$. A dummy job 0 (the *depot*) represents the initial state of the machine, so s_{0j} is the setup required before the first job. The goal is to order the jobs so as to minimize $\sum_{j\in J} w_j C_j$, where C_j is the completion time of job $j \in J$.

Introducing binary variables $y_{ij} = 1$ if job $j \in J$ is processed immediately after job $i \in J$ (with $i = 0$ for the first job) and continuous completion times $C_j \geq 0$, the formulation used in our experiments is

$$
\begin{aligned}
\min\ & \sum_{j\in J} w_j C_j && && (5)\\
\text{s.t.}\ & \sum_{j\in J} y_{0j} = 1, && &&\\
& y_{0j} + \sum_{i\in J, i\neq j} y_{ij} = 1, && && \forall j \in J,\\
& \sum_{j\in J, j\neq i} y_{ij} \leq 1, && && \forall i \in J,\\
& C_j \geq s_{0j} + p_j - M(1-y_{0j}), && && \forall j \in J,\\
& C_j \geq C_i + s_{ij} + p_j - M(1-y_{ij}), && && \forall i,j \in J,\ i \neq j,\\
& C_j \geq 0, \quad y_{ij} \in \{0,1\}. && &&
\end{aligned}
$$

The first three constraint families state that exactly one job starts the sequence, that every job has exactly one predecessor, and that every job has at most one successor. The big-M constraints link the completion times to the selected sequence and simultaneously act as subtour elimination; we use $M = \sum_{j\in J} p_j + \sum_{j\in J} \max_{i\neq j} s_{ij}$.

In the recursive model, a state is the pair (set of already scheduled jobs, last scheduled job) and stage t selects the job placed in the t-th position. Since the decision at each stage is a job index rather than a yes/no choice, the resulting diagram is a *multivalued* DD (MDD) with n stages and domain $\{1,\dots,n\}$, unlike the binary diagrams of the other four problems. This is also why in our implementation the cut separation of Section 6.4 uses target cuts for this problem.

The setup time matrices are taken from the *asymmetric traveling salesman problem* (ATSP) set of TSPLIB [48], the standard library for this family of problems. Given an ATSP instance on n cities with distance matrix d, we take city 0 as the depot and cities $1,\dots,n-1$ as the jobs, so that a job sequence is an open Hamiltonian path rooted at the depot and the resulting instance has $n-1$ jobs. The setup matrix is then $s_{0j} = d_{0,j+1}$ for the depot row and $s_{i+1,j} = d_{i+1,j+1}$ otherwise. The diagonal of a TSPLIB matrix carries a big-M value (from 9999 to 10^8 depending on the instance) marking the arc a city cannot take to itself; under the mapping above it falls exactly on the entries $s_{i+1,i}$, which no feasible sequence ever traverses.

TSPLIB supplies distances only, so processing times and weights are drawn following the scheme used in the sequencing literature cited by [22], namely $p_j, w_j \sim \mathcal{U}\{1,\dots,10\}$, with the seed derived from the instance name so that the draw is reproducible. We keep the ATSP instances with fewer than 50 cities, which yields the eight matrices reported in Table 7, and generate 5 replications of (p,w) for each, giving $8 \times 5 = 40$ instances following the convention `sequencing_atsp_{name}_n{jobs}_seed{S}`. The setup time ranges reported in the table exclude the diagonal entries $s_{i+1,i}$, which no feasible sequence traverses and which carry the TSPLIB sentinel discussed above.

B.5 SOC knapsack

The *conic* or *second-order cone (SOC) knapsack problem* replaces the linear capacity rows of (2) by conic ones. Given rewards c_i, linear coefficients a_{ji}, conic coefficients d_{ji}, right-hand sides b_j, and a scalar

Table 7: Sequencing instances derived from the TSPLIB ATSP set.

ATSP instance	cities	jobs	$\min s_{ij}$	$\max s_{ij}$
`br17`	17	16	0	74
`ftv33`	34	33	7	332
`ftv35`	36	35	7	332
`ftv38`	39	38	7	332
`p43`	43	42	0	5160
`ftv44`	45	44	7	332
`ftv47`	48	47	7	348
`ry48p`	48	47	54	2782

$\Omega \geq 0$, the problem is

$$\begin{aligned} \max \ & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.} \ & \sum_{i=1}^{n} a_{ji} x_i \ + \ \Omega \sqrt{\sum_{i=1}^{n} d_{ji}^2 x_i} \ \leq \ b_j, \qquad j = 1, \ldots, m, \\ & \boldsymbol{x} \in \{0,1\}^n. \end{aligned} \tag{6}$$

Because $x_i \in \{0,1\}$ implies $x_i = x_i^2$, the radicand equals $\|D_j \boldsymbol{x}\|_2^2$ with $D_j = \operatorname{diag}(d_{j1}, \ldots, d_{jn})$, so each constraint can be written as $\boldsymbol{a}_j^\top \boldsymbol{x} + \Omega \|D_j \boldsymbol{x}\|_2 \leq b_j$ and is indeed a second-order cone constraint.

The instances are the SOC knapsack benchmark from [4], used exclusively in the cutting plane experiment of Section 6.4. They follow the convention `knapsack_n{n}m{m}o{Ω}b{B}_{S}`, where $n \in \{100, 125, 150\}$ is the number of variables, $m \in \{10, 20\}$ is the number of conic constraints, $\Omega \in \{1, 3, 5\}$ is the conic multiplier of (6), $B = 5$ is fixed throughout the benchmark, and $S \in \{0, \ldots, 4\}$ indexes five replications. This gives $3 \times 2 \times 3 \times 5 = 90$ instances.

The instance files store, in order: a line with n and m; the value of Ω; the reward vector $\boldsymbol{c}$; the right-hand sides $b_1, \ldots, b_m$; then m lines with the linear coefficients a_{ji}; and finally m lines with the conic coefficients d_{ji}.

C Verification across implementations

To support the equivalence claim of Section 6.2, we verify that: (i) the Python and C++ implementations build the same diagrams, (ii) the value of an exact DD is indeed the optimum of the problem, and (iii) the restricted and relaxed diagrams return valid primal and dual bounds.

The first check compares the two implementations across all possible configurations (i.e., combinations of instance, construction type, maximum width W, reduction flag, and objective sense). For each such configuration, we compare three quantities: the optimization value returned by the diagram, the number of nodes of the diagram, and the number of arcs. Table 8 reports the results broken down by problem and construction type. Specifically, each cell represents the number of configurations on which the two implementations agree, out of the total number of configurations where both alternatives finish within the time limit. The three quantities coincide in all 20,880 configurations, which span 608 instances, the five example problems, the four construction types, both the reduced and the non-reduced variants, and every maximum width used in Section 6.

The second check validates the diagrams against an external reference: we compare the value of the (non-reduced) exact DD with the optimal value reported by Gurobi. This comparison is meaningful only for exact diagrams, since the restricted and relaxed constructions return bounds rather than optima. Table 9 reports, for each problem, the number of instances on which the exact DD value matches Gurobi's optimum; it does so on every one of the 282 instances. Two remarks are in order about the coverage of this comparison, which spans 282 of the 518 instances of these four problems. First, in the remaining instances, the exact DD exceeds the five-minute construction limit, which is precisely the regime that motivates the

Table 8: Python vs. C++ agreement, by problem and construction type.

Problem	Exact	Restricted	Relaxed (priority)	Relaxed (grouping)	Total
Knapsack	45/45	319/319	318/318	295/295	977/977
Independent set	36/36	766/766	754/754	725/725	2,281/2,281
Set cover	422/422	4,127/4,127	4,125/4,125	4,078/4,078	12,752/12,752
SOC knapsack	180/180	1,080/1,080	1,080/1,080	1,080/1,080	3,420/3,420
Sequencing	10/10	480/480	480/480	480/480	1,450/1,450
Total	693/693	6,772/6,772	6,757/6,757	6,658/6,658	20,880/20,880

Table 9: Agreement of exact DD values with the optimum reported by Gurobi.

Problem	# instances	DD = Gurobi
Knapsack	27	27
Independent set	21	21
Set cover	229	229
Sequencing	5	5
Total	282	282

restricted and relaxed diagrams in Section 6. Second, the converse also holds: in eight instances, the exact DD certifies an optimum that Gurobi fails to prove within its one-hour limit. These are one independent set instance and the five sequencing instances, where Gurobi terminates with optimality gaps between 57% and 118% while having already found the same solution, and two large knapsack instances, where Gurobi stops at its default relative optimality tolerance of 10^{-4} with an incumbent one unit below the true optimum. The SOC knapsack instances are excluded from this second comparison because each SOC knapsack DD models a single conic constraint and is used as a cut generator rather than as a standalone exact solver.

The third check verifies that the restricted and relaxed constructions return valid bounds. A restricted DD represents a subset of the feasible solutions, so its optimization bound is a primal bound that can never be better than the optimum; a relaxed DD represents a superset, so its optimization bound is a dual bound that can never be worse. For a maximization problem, we therefore require the restricted value to be at most the optimum and the relaxed value to be at least the optimum; the reverse holds for a minimization problem. We take as the reference the optimal value of the non-reduced exact DD when it is available; otherwise, we use the value reported by Gurobi for the instances where it closes the gap. This yields 490 instances with a known optimum (405 certified by the exact DD and 85 by Gurobi), making 17,589 of the 21,297 restricted and relaxed configurations checkable; the remainder correspond to instances whose optimum is unknown. The check spans the six maximum widths W of Section 6 and both the reduced and the non-reduced variants, and it is applied with a relative tolerance of 10^{-6}, since the values are stored as floating-point numbers. Table 10 reports, in each cell, the number of configurations whose bound holds over the number of configurations checked: every one of the 17,589 bounds holds. As in the first check, only the C++ implementation is verified, since Table 8 already establishes that both implementations return the same values.

D Gap of the relaxed bound: priority vs. grouping relaxation

This appendix complements the extensibility experiment of Section 6.3. Figure 7 reports, for every example problem and for both the C++ and Python implementations, the optimality gap of the relaxed bound (relative to the exact optimum) as a function of the maximum width, for the two relaxation mechanisms (priority and grouping). Each row corresponds to a problem (from top to bottom: knapsack, independent set, set cover, sequencing, and SOC knapsack) and each column to an implementation (left: C++, right: Python). In each panel the line is the average gap after removing outliers with the IQR rule and the shaded

Table 10: Validity of the restricted and relaxed bounds.

Problem	Restricted	Relaxed (priority)	Relaxed (grouping)	Total
Knapsack	348/348	346/346	345/345	1,039/1,039
Independent set	492/492	484/484	490/490	1,466/1,466
Set cover	3,888/3,888	3,888/3,888	3,888/3,888	11,664/11,664
SOC knapsack	1,080/1,080	1,080/1,080	1,080/1,080	3,240/3,240
Sequencing	60/60	60/60	60/60	180/180
Total	5,868/5,868	5,858/5,858	5,863/5,863	17,589/17,589

band the interquartile range. These plots confirm the observation made in the main text: the two mechanisms yield comparable bound quality across all problems, with neither dominating the other.

E DD visualizations using yEd

Figure 8 shows yEd visualizations of the exact, restricted, and relaxed DDs for our running example. Note that the states shown for the exact and restricted DDs are not accurate, since these diagrams are reduced (i.e., nodes with different states are merged) and the merge operator is not used during the reduction procedure. Nonetheless, they encode the same set of feasible solutions as the diagrams in Figures 1 and 2.

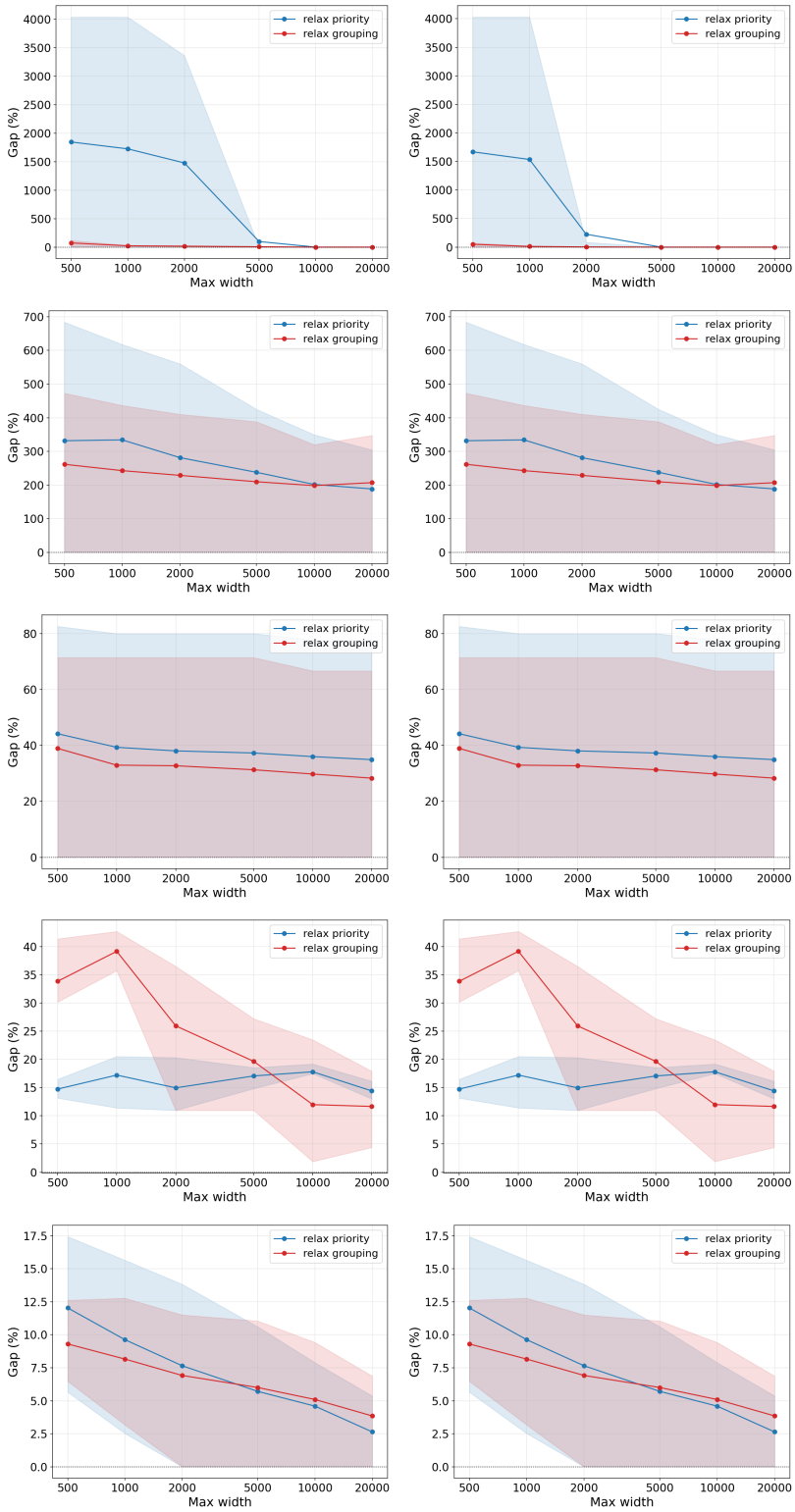


Fig. 7: Relaxed-bound gap versus maximum width (all problems).

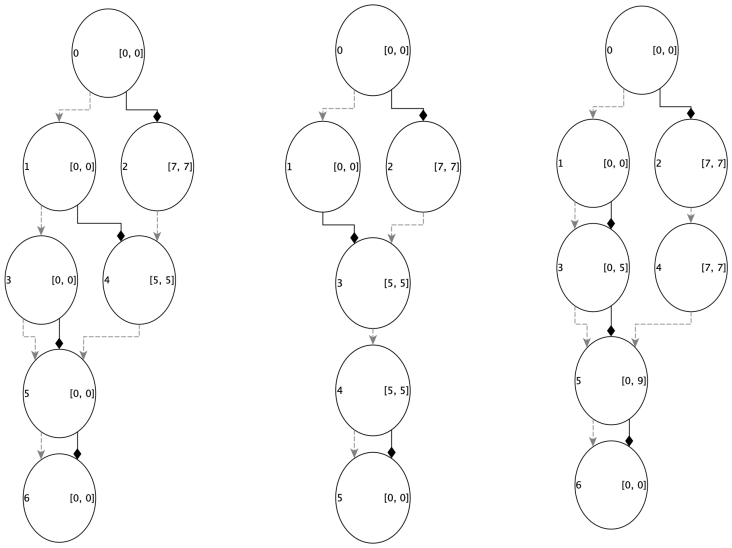


Fig. 8: yEd visualizations of the three DD types for our running example.