CODD: A Decision Diagram-based Solver for Combinatorial Optimization

Laurent Michel[a,*] **and Willem-Jan van Hoeve**[b,**]

[a]University of Connecticut, Storrs CT 06269, USA
[b]Carnegie Mellon University, Pittsburgh PA 15213, USA
ORCID (Laurent Michel): https://orcid.org/0000-0001-7230-7130, ORCID (Willem-Jan van Hoeve): https://orcid.org/0000-0002-0023-753X

Abstract. We introduce CODD, a system for solving combinatorial optimization problems using decision diagram technology. Problems are represented as state-based dynamic programming models using the CODD language specification. The model specification is used to automatically compile relaxed and restricted decision diagrams that are embedded inside a branch-and-bound search process. We introduce abstractions that allow us to generically implement the solver components while maintaining overall execution efficiency. We demonstrate the functionality of CODD on a variety of combinatorial optimization problems and compare its performance to other state-based solvers as well as integer programming and constraint programming solvers. CODD provides competitive results and can outperform the other solvers, sometimes by orders of magnitude.

1 Introduction

Decision diagram-based optimization (DDO) was introduced by Bergman et al. [6] as a generic methodology for solving discrete optimization problems. A decision diagram provides an explicit, compact, graphical representation of the set of solutions. Because such diagrams can grow exponentially large, Bergman et al. introduce relaxed decision diagrams of polynomial size to obtain relaxation (dual) bounds to the problem. Likewise, restricted diagrams are used to obtain heuristic (primal) solutions. These bounds are embedded into a branch-and-bound style search method that branches on nodes inside the diagram rather than discrete variables. This methodology was shown to outperform or be competitive with the state of the art for several application domains [5, 10, 17, 9]. We refer to [9] and [31] for recent surveys on decision diagrams for optimization.

Decision diagrams provide a natural bridge between combinatorial optimization, mathematical programming, state-based search, and constraint reasoning. In particular, Bergman et al. [6] developed their methodology to solve *dynamic programming models* through branch-and-bound. This was followed by the system Ddo [14] that offers a generic library to implement a DDO solver, and the Peel-and-Bound DDO solver [28]. More recently, Kuroiwa and Beck [21] introduced the *domain-independent dynamic programming* modeling formalism, with the Python modeling interface DIDPPy and associated generic state-based solvers using A* and beam search [22]. The availability of different generic solver technologies for dynamic programming models offers great potential for cross-fertilization between different fields, including automated planning and scheduling, operations research, and constraint solving [4].

Contributions. We present CODD, a generic solver for combinatorial optimization problems formulated as dynamic programming models. It is based on decision diagram technology, using as core principles a state definition, a state transition function, a cost transition function, and a state merging operator. We deviate from existing DDO methodology that follows the design of *ordered decision diagrams* in which states are arranged in explicit layers each associated with a decision variable. Instead, we propose a generalization that is *state based* and does not require explicit layers. This aligns the design of CODD more closely with general dynamic programming and state-based search methods. Our second main contribution is a specification language that uses high-level abstractions to concisely represent the components of the model. As our third contribution, we experimentally demonstrate the computational efficiency of CODD on a number of combinatorial optimization problems. Specifically, we show that CODD can outperform the other generic solvers DIDPPy and Ddo, sometimes by orders of magnitude.

In the remainder of the paper we first describe related work in Section 2. We then introduce the basic principles of DDO in Section 3, around which our system is designed. Section 4 presents the specifications of the solver components, and our procedure to automatically compile the associated elements of the solver. Lastly, in Section 5 we study the performance of CODD on four combinatorial optimization applications: the knapsack problem, the Golomb ruler problem, graph coloring, and the maximum independent set problem.

2 Related Work

Binary decision diagrams have been applied for solving (combinatorial) optimization problems since the mid-1990s, including branch-and-bound for 0/1 integer programming [20, 23] and binate covering problems [32]. While some of these earlier techniques are also used in DDO, the main difference is that DDO is a generic state-based paradigm that utilizes relaxed and restricted decision diagrams.

Decision diagram-based optimization is closely related to *constraint programming*, as decision diagrams can compactly represent the explicit solutions to global constraints. In fact, relaxed decision diagrams were first introduced in the context of multi-valued decision diagram (MDD) based constraint propagation [3]. Hoda [16]

* Corresponding Author. Email: ldm@uconn.edu.
** Corresponding Author. Email: vanhoeve@andrew.cmu.edu.

developed a generic decision-diagram based constraint programming solver, with a low-level C++ interface for developing new constraint propagators. Perez and Régin [25, 24] developed a C++ library of algorithms for MDD-based constraint propagation. The system Haddock [13] provides a more abstract view of MDD-based constraint programming, and introduces a specification language to implement new constraint types. The system automatically compiles the associated decision diagrams, which are embedded into the constraint propagation process of the MiniCP solver. Haddock cannot directly be used to solve DDO models, however, as it follows a constraint programming formalism instead of a state-based formalism.

Decision diagrams have also been utilized for *automated planning and scheduling*. In particular, Horn et al. integrate state-based search and decision diagram methodology to solve a range of planning and scheduling applications [18, 19]. Castro et al. [8] use relaxed decision diagrams for computing admissible heuristics for the cost-optimal delete-free planning problem. Kuroiwa and Beck [21] introduced DyPDL, a high-level modeling formalism for dynamic programming problems with several generic solvers: CABS, CAASDy, ACPS, APPS, and LNBS [22]. The DyPDL parser is implemented in Rust, while DIDPPy is a more recent Python interface.

For *stand-alone DDO solvers*, Bergman et al. [5] implemented the first such solver in C++. To develop new applications users are required to implement low-level C++ code. Likewise, the Peel-and-Bound DDO solver [28] offers a low-level Julia interface. The system Ddo [14] offers a generic library to implement a DDO solver more easily for applications. It is implemented in Rust but also offers the Python interface Py-DDO to access the library. Ddo also introduced local bounds [15] and dominance rules [11] into the solver. The decision platform Nextmv[1] is the first industrial-grade DDO implementation, and is tailored for vehicle routing and scheduling applications. Optimization models in Nextmv are implemented in the Go programming language. Nextmv provides a dedicated modeling interface to develop routing applications.

CODD extends this line of work and integrates ideas from state-based search into a DDO solver. Compared to prior DDO systems, it implements a generalized state-based implementation of decision diagrams, and offers abstractions to concisely and intuitively represent models in C++. Like the DIDPPy and Ddo solvers it integrates local bounds (which are similar to admissible heuristics) and dominance rules; CODD applies both of these during the compilation of the decision diagrams and the branch-and-bound search.

3 Decision Diagram-based Optimization

We next describe the main elements of decision diagram-based optimization (DDO). The basic principles follow the original framework by Bergman et al. [6], but we present a generalization that does no longer require models and solutions to proceed in a fixed number of stages. Instead, we allow solutions to be sequences of arbitrary length, similar to problems encountered in AI planning or general dynamic programming.

Let $\mathcal{U}$ be a universe of elements, and let $\mathcal{D}$ be a set of ordered sequences of elements in $\mathcal{U}$, each of arbitrary but finite length. We consider discrete optimization problems of the form

$$\begin{array}{lll} P: & \max & f(y) \\ & \text{s.t.} & C_j(y), j = 1, \ldots, m, \\ & & y \in \mathcal{D}, \end{array}$$

where y represents the decision variable ranging over the domain of sequences $\mathcal{D}$, $f : \mathcal{D} \to \mathbb{R}$ is the objective function over y, and $C_j : \mathcal{D} \to \{0, 1\}$ is a constraint over y for $j = 1 \ldots, m$. If all sequences in $\mathcal{D}$ are assumed to be of the same length, we obtain the problem form considered in [6].

3.1 Dynamic Programming Model

The set of solutions to P can be modeled using a state-based *dynamic programming* (DP) formulation. In dynamic programming, a solution is represented as a sequence of state-based decisions. It uses 'state variables' and 'decision variables', where a decision variable transitions one state into another state depending on the decision value (or 'label'). A dynamic program is defined by:

- A set of states $\mathcal{S}$ based on a *state definition*. We identify a single *initial state* r and a single *target state* t.
- A *label generation function* $\lambda : \mathcal{S} \to \mathcal{U}$ representing the decisions that can be taken from a state.
- A *state transition function* $\tau : \mathcal{S} \times \mathcal{U} \to \mathcal{S}$ representing the transition out of a state using a decision label into another state.
- A *state cost function* $c : \mathcal{S} \times \mathcal{U} \to \mathbb{R}$ representing the cost of a transition out of a state using a decision label.
- A *value function* $v : \mathcal{S} \to \mathbb{R}$ for each state. The value function is implicit and is *not* specified as part of the model, except that the value function $v(r)$ for the initial state can be set to a constant K.

The DP model proceeds in *stages*, where stage i is associated with state variable s_i and decision variable x_i, and $s_1 = r$. The DP formulation of P has the following form:

$$\begin{array}{lll} \max & v(t) & \\ \text{s.t.} & v(s_{i+1}) = \max\limits_{\substack{x_i \in \lambda(s_i) \\ \tau(s_i, x_i) = s_{i+1}}} v(s_i) + c(s_i, x_i) & \forall s_i \in \mathcal{S} \setminus \{t\} \\ & v(r) = K & \end{array}$$

Observe that the constraints $C_1, \ldots, C_m$ are captured through appropriately defining the state definitions and transition functions. In particular, the set of possible labels for a decision x_i depends on the state s_i and is encoded by the label generating function $\lambda(s_i)$.

Example 1. *Given a complete weighted directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, edge set E, and edge distance d_{ij} for all $(i, j) \in E$, the traveling salesperson problem (TSP) asks to find a closed tour in G of minimum total distance, visiting each vertex exactly once. To formulate this problem as a DP, solutions are represented as an ordered sequence of n vertices (i.e., as permutations): variable x_j represents the j-th location to be visited, with domain $D_j = V$ for $j = 1, \ldots, n$. We arbitrarily select vertex 1 as the depot.*

- *State definition: Tuple $\langle S, e \rangle$ where $S \subseteq V$ represents the set of visited vertices, and $e \in V$ represents the last visited location. The initial state is defined as $r = \langle \{1\}, 1 \rangle$ with value $v(r) = 0$. The target state is defined as $t = \langle V, 1 \rangle$.*
- *Label generating function:* $\lambda(\langle S, e \rangle) = \begin{cases} V \setminus S & \text{if } |S| < n, \\ \{1\} & \text{if } |S| = n. \end{cases}$
- *State transition function:* $\tau(\langle S, e \rangle, \ell) = \langle S \cup \{\ell\}, \ell \rangle$.
- *State cost function:* $c(\langle S, e \rangle, \ell) = d_{e,\ell}$.

3.2 Exact, Relaxed, and Restricted Decision Diagrams

Given a DP model, we define a *decision diagram* as a weighted directed acyclic graph $D = (N, A)$ where node set N corresponds

[1] https://www.nextmv.io/

to the set of state variables and arc set A corresponds to the state transition function τ. That is, each state $s_i \in \mathcal{S}$ has an associated node in N (also named s_i). For each transition $\tau(s_i, \ell) = s_j$ in the DP model we define an arc $(s_i, s_j) \in A$ with associated label ℓ and weight $c(s_i, \ell)$.[2] The labels along each arc-specified path from node r to node t correspond to a solution to P (i.e., a sequence in $\mathcal{D}$) and vice versa. Furthermore, the longest r-t path (w.r.t. the weights) corresponds to the optimal solution to P.

Let $\text{Sol}(D)$ denote the set of solutions represented by the decision diagram D, and let $\text{Sol}(P)$ represent the set of solutions of P. For each arc-specified r-t path p, let $w(p)$ denote its total weight and let x^p denote its sequence of arc labels.

Definition 1. *A decision diagram D is* exact *w.r.t. P if $\text{Sol}(D) = \text{Sol}(P)$ and $w(p) = f(x^p)$ for all r-t paths p in D.*

Exact decision diagrams can be compiled in a 'top-down manner' starting at the initial state, by recursively expanding the state space according to the transition function, while maintaining a unique representation of states that are equal. If the exact decision diagram fits into computer memory, we can directly solve P by computing the longest path. In most practical cases, the exact decision diagram is exponentially large in n. The crucial element of decision-diagram based optimization is to utilize *relaxed decision diagrams* of polynomial size to obtain a relaxation bound on P.

Definition 2. *A decision diagram D is* relaxed *w.r.t. P if $\text{Sol}(D) \supseteq \text{Sol}(P)$ and $w(p) \geq f(x^p)$ for all r-t paths p in $\mathcal{D}$ such that $x^p \in \text{Sol}(P)$.*

Relaxed decision diagrams are obtained by merging states that are not equivalent. This is accomplished by applying a *merge* operator $\oplus(\cdot)$ to a set of states $S \subset \mathcal{S}$, resulting in a new 'merged' state $\oplus(S)$. In order to define a relaxed decision diagram, we must take care that the merged state does not exclude feasible solutions. The merge operator is iteratively applied until the size of the diagram meets a maximum size limit, typically defined as a maximum *width*, i.e., the maximum number of nodes at each layer.

Note that the merge operator changes the definition of the dynamic program: the newly merged states need to be added to $\mathcal{S}$ if they were not defined recursively before by the DP model. Also, in some cases the DP model may need to be adapted to be a useful basis for a relaxed decision diagram, as shown in the following example.

Example 2. *We continue the TSP from Example 1. Given two states $\langle S_1, e_1\rangle$, $\langle S_2, e_2\rangle$, we restrict the merge operator to be defined only if $e_1 = e_2$, as $\oplus(\langle S_1, e_1\rangle, \langle S_2, e_2\rangle) = \langle S_1 \cap S_2, e_1\rangle$. By taking the intersection of S_1 and S_2 we ensure that no solutions are excluded. The benefit of only defining (and allowing) the merge operator when e_1 equals e_2 is that the cost function remains exact. The relaxation will, however, allow for possible repetitions. Therefore we may no longer reach the target state $\langle V, 1\rangle$ after n decisions. To correct this, we introduce a 'counter' state variable h that is increased by 1 after each decision. The updated DP model is as follows:*

- *State definition: Tuple $\langle S, e, h\rangle$. The initial state is defined as $\langle\{1\}, 1, 0\rangle$, and the target state as $\langle V, 1, n\rangle$.*
- *Label generating function:*
$$\lambda(\langle S, e, h\rangle) = \begin{cases} V \setminus S & \text{if } h < n-1, \\ \{1\} & \text{if } h = n-1. \end{cases}$$
- *State transition function: $\tau(\langle S, e, h\rangle, \ell) = \langle V, \ell, n\rangle$ if $\ell = 1$ and $\tau(\langle S, e, h\rangle, \ell) = \langle S \cup \{\ell\}, \ell, h+1\rangle$ otherwise.*
- *State cost function: $c(\langle S, e, h\rangle, \ell) = d_{e,\ell}$.*
- *Merge operator: $\oplus(\langle S_1, e_1, h_1\rangle, \langle S_2, e_2, h_2\rangle) = \langle S_1 \cap S_2, e_1, h_1\rangle$ if $e_1 = e_2$ and $h_1 = h_2$.*

Observe that we choose to restrict the merge operator to be applied only if the states have the same last vertex e and counter h.

To find primal (heuristic) solutions, we can use *restricted decision diagrams* that represent a subset of the feasible solutions:

Definition 3. *A decision diagram D is* restricted *w.r.t. P if $\text{Sol}(D) \subseteq \text{Sol}(P)$ and $w(p) \leq f(x^p)$ for all r-t paths p in D such that $x^p \in \text{Sol}(P)$.*

Restricted decision diagrams are easier to compile than relaxed decision diagrams. Given a priority of the nodes and a maximum size limit (typically defined per layer), a restricted decision diagram is obtained by simply discarding the lowest priority nodes beyond the maximum size limit. The longest r-t path, if it exists, is a primal solution to P.

3.3 Branch-and-Bound Search

To obtain an exact solution to P, DDO solvers follow a branch-and-bound search process. Initially a relaxed and restricted decision diagram are compiled, resulting in a dual and primal bound. We then identify an *exact cutset* of nodes $L \subset N$ in the relaxed decision diagram. Typically L is selected to correspond to decision stage i where no nodes are the result of the merge operator, while decision stage $i+1$ contains at least one merged node.

Because of the Markovian property of the states in the DP model we can independently consider the nodes in L and use them to define new subproblems, where the state of each node in L corresponds to the initial state of the subproblem. This preserves optimality because the optimal solution must pass through one of the nodes in L. Each subproblem will again compile a relaxed and restricted decision diagram, as well as a last exact layer. We recursively continue this process until each subproblem is either exact or proven suboptimal because of the optimization bounds. Given enough time, this process terminates with the optimal solution to P. Otherwise, it provides a lower and upper bound on the optimal solution.

Remark. We note that our generalization changes several fundamental assumptions of decision diagram-based optimization. Most importantly, we no longer rely on explicit layers such that layer i corresponds to decision x_i. Instead, we define layers *implicitly* as the set of nodes that have the same length to the root. As a consequence, we can define *state-dependent* label generation functions, and even state-dependent decision variable selection. However, our framework is a true generalization in that the decision diagram definitions from the DP model carry over without any change, and that we recover the original architecture from [6] when all solution sequences are assumed to be of equal length.

4 Components of the CODD System

4.1 Language

CODD is implemented as a C++ library.[3] The design of the CODD modeling layer closely follows the concepts from the previous section. As an illustration, Figure 1 shows the CODD model for the

[2] We note that D is acyclic because the DP model is assumed to represent the sequences $\mathcal{D}$ in problem P. In general DP models could however lead to cyclic graphs; in this work we assume that all DP models are acyclic.

[3] See https://github.com/ldmbouge/CODD.

```
1  struct TSP { GNSet S;int e;int h; }; // state definition
2  const auto w = 64;                   // maximum DD width
3  auto [depot,n,V,d] = readInstance(fName);     // file I/O
4  auto init= [](){ return TSP { GNSet{depot},depot,1 };};
5  auto target=[n,&V]() { return TSP { V,depot,n };};
6  auto lgf   =[n,&V](const TSP& s)  {   // label generator
7     return (s.h >= n-1) ? GNSet {depot} : (V-s.S);
8  };
9  auto stf = [depot,n,&V,&d](const TSP& s,int label) {
10    if (label==depot)
11       return TSP {V,depot,n};
12    else
13       return TSP {s.S | GNSet{label},label,s.h+1};
14 };
15 auto scf = [&d](const TSP& s,int label) {        // cost
16    return d[s.e][label];
17 };
18 auto smf = [](const TSP& s1,const TSP& s2) {    // merge
19    if (s1.e == s2.e && s1.h == s2.h)
20       return TSP {s1.S & s2.S, s1.e, s1.h};
21    else return std::nullopt;           // empty optional
22 };
23 auto eqs = [depot,n](const TSP& s) {  // target equality
24   return s.e == depot && s.h == n;
25 };
26
27 BAndB engine(DD<TSP,Minimize<double>,...>
28     ::makeDD(init,target,lgf,stf,scf,smf,eqs,V),w);
29 engine.search(bnds);
```

Figure 1. CODD model for the Traveling Salesperson Problem. A GNSet is a set of natural numbers of arbitrary size.

TSP in Example 2. The *state* is modeled via a `struct` type. For the TSP, the structure on line 1 holds three attributes to model the visited vertices (`S`), the last visited vertex (`e`) and the counter (`h`) matching exactly the definition in Example 2.

The dynamic programming concepts introduced earlier as well as the merge operator $\oplus$ are *all* captured with first order functions (i.e., *lambda* in `C++`). The elements of the CODD language are:

- Initial state: the `init` lambda on line 4,
- Target state: the `target` lambda on line 5,
- λ: the label generating `lgf` lambda on lines 6-8,
- τ: the state transition `stf` lambda on lines 9-14,
- c: the state cost `scf` lambda on lines 15-17,
- $\oplus$: the merge `smf` lambda on lines 18-22,
- Target equality: the equality to t `eqs` lambda on lines 23-25,
- Objective: (defined as `Minimize` or `Maximize`) on line 27.

We added the optional `eqs` lambda to implement a more efficient test for whether a state is the target. Observe that the DP definitions in Example 2 are perfectly mirrored by the C++ fragment in Figure 1. For instance, line 13 returns the output state (an instance of `TSP`) with the first attribute set to be equal to `s.S | GNSet {label}`, the second attribute set to `label` and the third to `s.h+1`, i.e., $\tau(\langle S, e, h\rangle, \ell) = \langle S \cup \{\ell\}, \ell, h+1\rangle$. Indeed, `GNSet` creates a singleton with the provided value `label` and the binary operator `|` implements set union.

CODD offers a lightweight domain specific layer (DSL) atop C++. This choice confers to the DSL a familiarity that makes it convenient and compact. Note how lines 27-28 instantiate the solver with all the required lambdas and then invokes the branch-and-bound search (line 29). The last argument in `makeDD` is the set of possible labels, which is V in the case of the TSP.

Extensions It is possible to augment the core CODD model with state *dominance rules* as well as *local bounding* procedures.[4] We discuss here the local bounds and provide details on the dominance rules in Section 5 in the context of the knapsack problem.

A local bound is defined on a state and provides an approximation of the remaining value to reach the target, similar to an admissible heuristic in AI planning. Local bounds are helpful to quickly discard states that cannot improve upon the incumbent solution. As an illustration for the TSP, for a given state `s` the weight of a minimum spanning tree (MST) that connects unexplored vertices to the current partial solution (a path from the `depot` to `s.e`) provides a local bound. The C++ API is:

```
auto local = [depot,&d,&V](const TSP& s) -> double {
   return mst(d,V,s.S,depot,s.e,s.h);
};
```

The C++ lambda captures the `depot` as well as references to the distance matrix `d` and the full set of vertices `V`. It then calls an implementation of Kruskal's MST algorithm using the approximation of the set of visited vertices `s.S` as well as its true cardinality: `s.h`. The complete implementation, including all the IO code to read instances is less than 150 lines of code with the model itself not exceeding 30 lines. While one should *not* expect a naive 30-line model devoid of any sophisticated bound to compete with state-of-the-art dedicated solvers, it exhibits notable behaviors. When tested on the asymmetric TSP instance `br17` from TSPLIB, one observes the value that relaxed and restricted decision diagrams bring to the branch-and-bound search:

1. The 30-line model based on decision diagrams finds the optimum and proves it in under 2 seconds (with local bounding).
2. Decision diagrams deliver dramatic reductions in the size of the branch-and-bound tree and the runtime as width is increased, going from 20 millions nodes at width=2 to 4,594 nodes at width=16,384.
3. The local bound used *within* the diagram acts as an *amplifier*. When used, the number of nodes drops to 136,407 at width=2 and to 181 at width=512. Increasing the maximum width from 2 to 512 delivers a branch-and-bound tree 753 times smaller.

4.2 *Implementation*

The implementation of CODD leverages features found in modern C++. In particular it embraces the standard template library, generic types (aka "Templates") and the now ubiquitous first-order functions adopted from functional languages. This bestows a declarative and functional style for creating models. The CODD library provides *generic* functionality which, when instantiated with model-specific state description and manipulation functions produces a full-blown implementation. Several design choices are reviewed in this section. Those were governed both by the desire to retain elegant models and to deliver excellent performance.

State Genericity *A* CODD *state is represented by a generic type.* The library does not know anything about state representation and operates on states through user-provided operations (e.g., transition or cost functions). Two requirements exist for a type to represent a decision diagram state. First, it must support equality testing to recognize a state and to ensure a unique representation. Second, it must support hashing to be efficiently looked up and implement uniqueness. The `STL` provides the necessary interfaces: `std::equal_to<T>` and `std::hash<T>`.

Rich Data Types Attributes of a state in CODD can be values of any legitimate C++ data type (as long as they are immutable). That includes scalars of any type, enumerated types, structure (compound types), generic arrays (i.e., `FArray<T>`) or sets of values. Sets of natural numbers can be heap-allocated and of arbitrary size (`GNSet`) or embedded and of fixed size (`NatSet`). The TSP example illustrated the use of dynamically allocated sets of naturals. The data structures themselves are mutable, but once encapsulated in a CODD

[4] These functions are also present in the Ddo and DIDPPy solvers.

state, they can no longer change as the state itself is immutable. One can even use any STL type to define attributes. Given their usefulness, the set types `GNSet` and `NatSet` provide constant time operations (for a fixed number of 64-bit words) for set intersection, union, difference as well as the vanilla membership tests or the construction via insertions or removals.

First-Order Function *All state operations use first-order functions.* CODD models use exclusively first order functions (C++ lambdas) to create states (initial state, target state, transitions) and assess their cost. Those functions are bona fide functional closures that capture ambient states in the program to refer to data structures holding distance matrices, graphs, or any other artifacts part of the model. Restricting modelers to adhere to state and functional transitions is key to discourage convoluted models (e.g., with side-effects). It compels end-users to adhere to the mathematical definition of their problem when designing the model.

Immutability *A* CODD *state never changes after its creation.* The requirement makes it possible to conform to a strictly functional world view. States never change. If a merge occurs, it creates a new state that replaces those that spawned it. It permits the implementation to use a simple hashing scheme to hold onto states and preserve a unique representation at all times, allowing to leverage simple pointer equality for equality testing afterwards.

Layer-free Design *State layers are implicit in* CODD. Decision diagrams are usually presented (and implemented) as stratified graphs where nodes are organized in $n+1$ layers and arcs connect states from layer $\mathcal{L}_i$ to layer $\mathcal{L}_{i+1}$. The initial state belongs to $\mathcal{L}_1$ and the target to $\mathcal{L}_{n+1}$, and layer $\mathcal{L}_i$ has an associated decision x_i for each of its nodes. This design is adopted in most decision diagram-based systems mentioned in Section 2. Yet, dynamic programming does not mandate such a strict organization. CODD creates diagrams as a directed graphs where layers are *implicit* and rather resemble 'stages' as in dynamic programming: We define a layer (or stage) i to be all states that are i steps away from the initial state, and we restrict these to be of a maximum width when building relaxed and restricted decision diagrams. In CODD one can transition out of states that appear in the same "layer" by making decisions about different variables if one so chooses. However, modelers are free to impose a layer structure if they so decide by choosing a suitable state representation that encodes, in an attribute, the layer of a state (that monotonically increases with transitions). As an illustration, the counter h in the state definition for the TSP in Example 2 can serve this purpose.

Explicit Diagrams *All states are kept until a diagram is discarded.* CODD derives diagrams by holding onto an explicit representation of *all* the nodes in the diagram. CODD keeps the entire diagram in memory when processing a sub-problem. This choice allows to easily compute both forward and backward bounds (from the initial state and from the target) that are essential for states to be transferred to the branch and bound or even compute local bounds.

4.3 Discussion

The layer-free design decision matches dynamic programming and is a departure from traditional decision diagrams that associate a decision variable to each layer. When processing layer i, all arcs leaving states are related to values in the domain of decision variable x_i. In a layer-free design, which variable to branch on next is a function of the parent state. It means that paths from r to t can have different *lengths*. Indeed, in a layer-free setup, a decision may entail jumping multiple layers in a classic setup. The adoption of layers is *a decision entirely left to the modeler*. They can choose to stratify their diagram with an additional attribute in the state to track a layer number, or they can go layer-free.

Care must be taken in the definition of the diagram to prevent *cycles*. In a layered graph, arcs always proceed from layer i to layer $i+1$ and the question does not arise. CODD refuses to create arcs that link a state s_1 that is i steps from r to a state s_2 that is j steps from r if $j < i$, to prevent the creation of cycles. Nonetheless, sequences of decision in CODD always have a bounded length.

5 Applications and Empirical Evaluations

We next study the performance of CODD on several combinatorial optimization problems: knapsack, Golomb ruler, graph coloring, and maximum independent set. The details of all CODD models are available on the GitHub repository. All experiments we conducted with all solvers were carried out on the same Macbook Pro with an M1Pro Processor and 32G of RAM.

Knapsack Given a set of N items $I = \{0, \dots, N-1\}$ each with a weight $w_i \in \mathbb{N}$ and profit $p_i \in \mathbb{N}$ and a capacity C, the knapsack problem is to select a subset of items with total weight at most C while maximizing the total profit. This problem is well suited for dynamic programming and models for Ddo [14] and DIDPPy [21] are available. We use a collection of 29 standard benchmarks used by both Ddo and DIDPPy that scale in size and hardness, with some instances featuring correlated items.

The CODD specification for knapsack is as follows. We assume that the items I are sorted by decreasing profit-to-weight ratios. Let the state be a tuple $\langle n, c\rangle$ where $n \in I$ is the index of the next item to consider and c is the remaining capacity. The model is:

- The initial and target states are $r = \langle 0, C\rangle$ and $t = \langle N, -\rangle$.
- $\lambda(\langle n, c\rangle) = \{0, 1\}$ if $c \geq w_n$ and $\{0\}$ otherwise.
- $\tau(\langle n, c\rangle, \ell) = \langle n+1, c-\ell \cdot w_n\rangle$ if $n < N-1$ and $\tau(\langle n, c\rangle, \ell) = \langle N, -\rangle$ otherwise.
- $c(\langle n, c\rangle, \ell) = p_n \cdot \ell$.
- The merge operator is $\oplus(\langle n_1, c_1\rangle, \langle n_2, c_2\rangle) = \langle \max(n_1, n_2), \max(c_1, c_2)\rangle$ if $|c_1 - c_2| \leq \frac{2 \cdot C}{100}$. The intent is to *reject* state merges when the capacities are too dissimilar (more than 2%).
- The target equality is $\texttt{eqs}(\langle n, c\rangle) = (n = N)$.

Two *optional* functions can improve model performance. First, one can use a straightforward local bound that fills the remainder of the knapsack with the most profitable items per unit of weight and finish with a fractional item. Second, the knapsack problem exploits a dominance rule to avoid exploring states that are dominated (its solutions being necessarily inferior). This capability is also present in Ddo and DIDPPy. In CODD it is conveyed with a simple function:

$$dom(\langle n_1, c_1\rangle, \langle n_2, c_2\rangle) = c_1 \geq c_2$$

that returns `True` when the capacity of the first state exceeds or matches that of the second state. CODD automatically uses the primal bound for a state and calls its dominance rule (if given) only if the state is of a higher quality. The C++ implementation of this dominance rule is:

```
auto dom = [](const SKS& a,const SKS& b) -> bool {
   return  a.c >= b.c;
};
```

where `SKS` is the C++ structure holding the knapsack state. The entire CODD model with the optional functions is just 40 lines of code, including the C++ functions for state hashing and equality testing.

Width	#B&B Nodes	Dom	Time Proof (s)
2	42	6/1	0.278
4	9	0/0	0.083
8	21	0/0	0.062
16	18	1/1	0.048
32	30	10/7	0.051
64	53	24/17	0.088
128	201	87/162	0.162
256	701	293/481	0.344
512	1	0/0	0.609

Table 1. CODD behavior on `knapPI_1_1000_1000`.

Impact of Width We first study the impact of the *maximum width* on the performance of CODD. Larger widths results in stronger decision diagrams with better bounds, but are also more costly to compute. Table 1 shows the behavior of CODD when the width increases from 2 to 512, on instance `knapPI_1_1000_1000` that uses 1000 items. First, CODD finds the optimum (and proves it) at all widths. Note that the number of nodes in the branch-and-bound search (#B&B Nodes) is not monotonically increasing or decreasing with the width. This is problem dependent, and inherent in the design of DDO solvers. Second, past width=512, the problem is solved at the root node (in 609ms). Third, the dominance rule (column Dom) can discard a meaningful number of nodes. At width 256 for instance, 293 new nodes are discarded since some other nodes in the queue dominate them, while newly added nodes prune 481 dominated nodes that were in the branch-and-bound queue. Lastly, the best runtime happens to be at width=16 with just 48ms demonstrating that extensive use of the merge operator at lower width does carry a cost.

Solver Comparison We next compare the performance of CODD with Ddo and DIDPPy (using the recommended default solver CABS), using their default settings. We note that Ddo uses a dynamic maximum width, based on the number of decision variables (items) in the remaining subproblem. For CODD we fixed the width to an arbitrary value of 64. We ran all 29 benchmark instances with all three solvers and present the results in Table 2. Dashes indicate that the solver could not run at all, while italics indicate that the solver ran up to its default timeout (of 30 seconds)[5]. Column B&B reports the number of nodes in the branch-and-bound tree for CODD at the specified width. For the `PI:3` instances with 5,000 and 10,000 items, the width had to be raised to 4096 and 8192, respectively, to close the instance at the root (and always under 3 seconds). Unsurprisingly, the solvers are virtually indistinguishable on the small instances (`f1` through `f10`). The situation changes for the 3 PI families where the size of the instances (number of items, column N) has a major impact. CODD solves all instances, even with a small width. While competing at a width of 64, it delivers a speedup ranging from 1 to 3 orders of magnitude against DIDPPy and 1 to 2 orders of magnitude against Ddo. Because Ddo uses a dynamic width, we report additional rows for large instances solved with CODD with similarly large widths (namely, 1024, 2048, 4096 and 8192). When wide enough, CODD finds the optimum and proves it with about 3 seconds even on the largest instances.

Golomb Ruler Problem Given an integer n, the Golomb Ruler Problem asks for integer locations of n marks along a ruler of minimum length such that no two pairs of marks are the same distance apart. The first mark is assumed to be at location 0. CODD models

[5] Though we note that the solver received up to 120 seconds but always aborted after 15 seconds.

	Benchmarks			CODD		DiDPPy	Ddo
	N	Width	Opt	B&B	T (s)	T (s)	T (s)
f1	10	64	295	1	0.000	0.025	0.000
f2	20	64	1024	1	0.000	0.033	0.000
f3	4	64	35	1	0.000	0.024	0.000
f4	4	64	23	1	0.000	0.023	0.000
f6	10	64	52	1	0.000	0.022	0.000
f7	7	64	107	1	0.000	0.024	0.000
f8	23	64	9767	1	0.001	0.041	0.006
f9	5	64	130	1	0.000	0.024	0.000
f10	20	64	1025	1	0.000	0.032	0.000
PI:1	100	64	9147	1	0.000	0.095	0.004
PI:1	200	64	11238	1	0.000	0.155	0.011
PI:1	500	64	28857	1	0.001	1.759	0.196
PI:1	1000	64	54503	53	0.088	9.286	1.039
PI:1	2000	64	110625	80	0.287	43.160	5.804
PI:1	5000	64	276457	104	1.189	314.370	*15.068*
PI:1	10000	64	563647	88	18.818	-	*15.048*
PI:2	100	64	1514	8	0.004	0.167	0.002
PI:2	200	64	1634	29	0.010	0.135	0.010
PI:2	500	64	4566	66	0.016	1.474	0.137
PI:2	1000	64	9052	59	0.060	7.479	0.804
PI:2	2000	64	18051	305	0.567	36.850	5.243
PI:2	5000	64	44356	143	3.125	229.670	*15.082*
PI:2	10000	64	90204	139	18.646	-	*15.066*
PI:3	100	64	2397	1	0.000	0.071	0.002
PI:3	200	64	2697	47	0.019	0.131	0.011
PI:3	500	64	7117	65	0.016	1.445	0.087
PI:3	1000	64	14390	222	0.152	6.516	0.563
PI:3	1000	1024	14390	1	0.022	6.516	0.563
PI:3	2000	64	28919	4093	464.119	31.129	4.157
PI:3	2000	2048	28919	1	1.484	31.129	4.157
PI:3	5000	64	72505	3321	3167.840	212.59	-
PI:3	5000	4096	72505	1	2.062	212.59	-
PI:3	10000	64	146919	760	1711.000	-	-
PI:3	10000	8192	146919	1	3.166	-	-

Table 2. Knapsack results for CODD, DIDPPy, and Ddo.

the states as tuples $\langle M, D, k, e, s\rangle$, where M is the set of marks already placed, D is the set of pairwise distances induced by M, k is the number of marks placed, e is the location of the most recently placed mark and s is the smallest unused distance.

- The initial and target states are $r = \langle\{0\}, \varnothing, 1, 0, 1\rangle$ and $t = \langle\varnothing, \varnothing, n, 0, L+1\rangle$ where L is a given upper bound on the position of the last mark (e.g., we use n^2).
- $\lambda(\langle M, D, k, e, s\rangle)$ returns a subset of values drawn from $\{1, \ldots, L\}$ that are *legal*. The legality condition uses the smallest unused distance s (see [12] and the model on GitHub).
- $\tau(\langle m, D, k, e, s\rangle, \ell) = t$ if $k = n - 1$, and $\langle M \cup \{\ell\}, w, k + 1, \ell, s'\rangle$ otherwise (adding a mark at position ℓ). Here, s' is the smallest value that does not appear in the new distance set $\{\ell - m | m \in M\} \cup D$.
- $c(\langle M, D, k, e, s\rangle, \ell) = \ell - e$, modeling the ruler length increase.
- $\oplus(\langle M_1, D_1, k_1, e_1, s_1\rangle, \langle M_2, D_2, k_2, e_2, s_2\rangle) = \langle M_1 \cap M_2, D_1 \cap D_2, k_1, e_1, \min(s_1, s_2)\rangle$ if $k_1 = k_2$ and $e_1 = e_2$.
- The target equality is $\texttt{eqs}(\langle M, D, k, e, s\rangle) = (k = n)$.

The total CODD model is less than 140 lines of C++.

Impact of Width Whereas the knapsack problem favors dynamic-programming style solvers (a large enough width can solve the problem at the root node), the Golomb ruler problem is a much more challenging computational problem. Figure 2 shows the behavior of CODD on the representative instance with 12 marks, for widths from 2 to 8192. As expected, the runtime does decrease as the maximum width increases, up to width 2048. After that, the cost incurred for a wide diagram overtakes the marginal benefits in the branch and bound effort. Indeed, in this case a larger width alone does not suffice to solve the instance at the root node.

Solver Comparison We next compare the performance of CODD with Ddo and the Comet constraint programming (CP) system, for

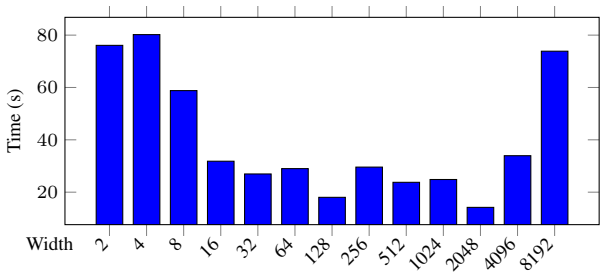

Figure 2. Golomb Ruler (12 marks). Time to find and prove the optimum.

	Comet		Ddo		CODD width=64		
n	T (s)	B&P	T (s)	T_U (s)	Opt	T_O (s)	T_P (s)
7	0.0	104	0.0	0.1	25	0.0	0.0
8	0.0	595	0.1	0.8	34	0.0	0.0
9	0.1	3149	0.9	6.6	44	0.1	0.1
10	0.6	19938	7.3	53.0	55	0.2	0.3
11	12.3	0.3M	*908.8*	1693.9	72	1.1	5.5
12	124.0	2.6M	-	-	85	24.7	29.0
13	40,285.0	459M	-	-	106	480.4	615.6
14	-	-	-	-	127	839.2	2567.7

Table 3. Solver comparison on Golomb Ruler with 7-14 marks.

which we use the available models from their respective distribution. Table 3 shows a comparison of the three solvers, all using the same machine. Column n identifies the instance size, the columns T and B&P for Comet report the runtime (in seconds) and branch-and-prune tree size (suffix M is for millions of nodes). Under Ddo, column T reports the runtime while T_U reports user CPU time (cumulative across all cores). Indeed, Ddo uses all the machine's cores to speed up the branch and bound. The time for instance n=11 is italicized to convey that the solver aborted with a time limit before producing an optimality proof. Under the CODD heading, Opt is the optimum, T_O is the time to find the optimum and T_P is the time to deliver the optimality proof. First, CODD is the only solver that scales to instance size 14 in a reasonable runtime. Second, the speed difference w.r.t. Ddo is remarkable (165 times faster at size 11), especially as we compare a strictly sequential solver (CODD) against a parallel one (Ddo). Third, the Comet [29] model includes symmetry-breaking and provides results that exceeded those from Chuffed [1] and OR-Tools [26, 27] on the canned MiniZinc model [2] for the Golomb ruler. In summary, CODD solves instances up to size 14, *on a single core* in under 2600 seconds. It outperforms CP as well as IP and QP models described in a prior computational study [7].

Graph Coloring We present the performance of CODD on graph coloring problems in Table 4, using instances from the DIMACS graph coloring benchmark that CODD can solve in 120s. We compare CODD with the approach in [30] that also uses decision diagrams to solve graph coloring problems. In that work, the diagram represents the independent sets of the graph, and a solution is a collection of r-t paths in the diagram found by solving an integer network flow problem. Some instances allow for a small enough exact diagram to be solved (IPDD-exact). Otherwise, a relaxed diagram is used that is iteratively refined (IPDD-relaxed). We can see that a pure DDO approach using CODD outperforms the IPDD approach on most of these instances. The IPDD results were obtained on an Intel Xeon E5345@2.33GHz CPU with a 1-hour timeout (TO) [30].

Maximum Independent Set Lastly, we compare the performance of CODD with Ddo on the maximum independent set problem. Table 5 reports on a set of 15 DIMACS independent set benchmark instances. For this problem, the Ddo solver uses an advanced layer-based variable selection heuristic while CODD follows a simple static ordering. Both solvers use 1 thread. While this CODD model performs reasonably well, it is apparent that the advanced variable selection heuristic pays off. Yet, this heuristic is layer-based and does not have a direct analogue in CODD due to its state-based design.

	CODD			IPDD-relaxed		IPDD-exact	
Instance	Obj	B&B	T (s)	Obj	T (s)	Obj	T (s)
myciel3	4	1	0.001	4	0.04	4	0.02
myciel4	5	549	0.163	5	7.03	5	0.85
2-Insertions_3	4	20526	8.746	[3,4]	TO	4	930.37
1-FullIns_3	4	1	0.002	4	0.06	4	0.09
2-FullIns_3	5	13043	24.328	5	0.83	5	2.73
r125.1	5	1	0.032	5	0.02	5	0.05
r125.1c	46	878	0.041	46	2.08	46	0.13
anna	11	1	0.151	11	0.01	-	TO
jean	10	37	1.303	10	0.01	10	0.73
david	11	1	0.037	11	0.01	11	5.07
queen5_5	5	1	0.001	5	0.01	5	0.03
queen6_6	7	2386	0.056	7	2.35	7	0.36
queen7_7	7	15184	0.381	7	2.77	7	0.88
DSJC125.1	5	22755	111.626	[5,6]	TO	-	TO
miles250	8	1	0.063	8	0.01	8	0.46
miles500	20	1	0.247	20	0.04	20	1.71
miles750	31	58	25.794	31	0.12	31	0.96
school1	14	20	0.844	14	1.67	-	TO
school1_nsh	14	2494	10.016	14	15.59	-	TO

Table 4. Coloring instances solvable by CODD in under 120s (width=64). TO stands for time-out (1 hour).

Benchmark		CODD	(w=128)	Ddo
Name	Opt	T_O (s)	T_P (s)	T (s)
johnson8-2-4	4	0.001	0.001	0.000
johnson8-4-4	14	0.005	0.005	0.004
johnson16-2-4	8	0.011	0.692	0.835
keller4	11	0.016	14.796	3.113
hamming6-2	32	0.005	0.005	0.004
hamming6-4	4	0.003	0.003	0.001
hamming8-2	128	0.051	0.051	0.049
hamming8-4	16	0.028	14.476	15.543
hamming10-2	512	0.474	0.475	0.666
brock200_1	21	929.975	1214.000	241.432
brock200_2	12	0.689	1.425	0.538
brock200_3	15	5.737	11.556	3.667
brock200_4	17	9.551	43.478	12.112
p_hat300-1	8	0.111	0.588	0.157
p_hat300-2	25	0.949	179.711	10.841

Table 5. Maximum Independent Set Problem. CODD uses width=128.

6 Conclusion

We introduced CODD, a modeling interface and solver for decision diagram-based optimization (DDO). Based on the principal DDO concepts of states, transition functions, and a state merging operator, CODD is an intuitive domain specific modeling layer and library in C++. It offers the ability of state-based modeling, thus complementing both integer programming and constraint programming modeling styles. We showed how CODD integrates elements from state-based search into the DDO solving paradigm, as it generalizes the compilation of decision diagrams to be state based rather than layer based. We demonstrated the flexibility and performance of the system on four combinatorial optimization problems, showing that CODD is competitive with, or outperforms, other generic exact solvers based on decision diagrams, state-based search, and constraint programming. CODD is implemented in 9,000 lines of pure C++. Its high-level specifications allow CODD to be easily be embedded in a high-level language like Python or Julia.

Acknowledgements

Laurent Michel was partially supported by Synchrony. Willem-Jan van Hoeve was partially supported by Office of Naval Research Grant No. N00014-21-1-2240 and National Science Foundation Award #1918102.

References

[1] URL https://github.com/chuffed/chuffed.

[2] URL https://www.csplib.org/Problems/prob006/models/golomb.mzn.html.

[3] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Proceedings of the 13th international conference on Principles and Practice of Constraint Programming*, pages 118–132. Springer-Verlag, 2007.

[4] J. C. Beck, D. Magazzeni, G. Röger, and W.-J. van Hoeve. Planning and operations research (dagstuhl seminar 18071). *Dagstuhl Reports*, 8(2):26–63, 2018. doi: 10.4230/DAGREP.8.2.26. URL https://doi.org/10.4230/DagRep.8.2.26.

[5] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Discrete optimization with decision diagrams. *INFORMS J. Comput.*, 28(1):47–66, 2016.

[6] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. *Decision diagrams for optimization*. Springer, 2016.

[7] B. Burak Kocuk and W.-J. van Hoeve. A computational comparison of optimization methods for the golomb ruler problem. In L. Rousseau and K. Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 409–425. Springer, 2019. URL https://doi.org/10.1007/978-3-030-19212-9_27.

[8] M. P. Castro, C. Piacentini, A. A. Ciré, and J. C. Beck. Solving delete free planning with relaxed decision diagram based heuristics. *J. Artif. Intell. Res.*, 67:607–651, 2020. doi: 10.1613/JAIR.1.11659. URL https://doi.org/10.1613/jair.1.11659.

[9] M. P. Castro, A. A. Cire, and J. C. Beck. Decision Diagrams for Discrete Optimization: A Survey of Recent Advances. *INFORMS J. Comput.*, 34 (4):2271–2295, 2022.

[10] A. A. Cire and W.-J. van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61:1411–1428, 2013.

[11] V. Coppé. *Advances in Discrete Optimization with Decision Diagrams: Dominance, Caching and Aggregation-Based Heuristics*. PhD thesis, UCLouvain, 2024.

[12] A. Dollas, W. Rankin, and D. McCracken. A new algorithm for Golomb ruler derivation and proof of the 19 mark ruler. *IEEE Transactions on Information Theory*, 44(1):379–382, Jan. 1998. ISSN 1557-9654. doi: 10.1109/18.651068. URL https://ieeexplore.ieee.org/document/651068.

[13] R. Gentzel, L. Michel, and W.-J. van Hoeve. HADDOCK: A Language and Architecture for Decision Diagram Compilation. In *Proceedings of CP*, volume 12333 of *LNCS*, pages 531–547. Springer, 2020.

[14] X. Gillard, P. Schaus, and V. Coppé. Ddo, a Generic and Efficient Framework for MDD-Based Optimization. In *Proceedings of IJCAI*, pages 5243–5245, 2020.

[15] X. Gillard, V. Coppé, P. Schaus, and A. A. Ciré. Improving the filtering of branch-and-bound MDD solver. In P. J. Stuckey, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5-8, 2021, Proceedings*, volume 12735 of *Lecture Notes in Computer Science*, pages 231–247. Springer, 2021. doi: 10.1007/978-3-030-78230-6_15. URL https://doi.org/10.1007/978-3-030-78230-6_15.

[16] S. Hoda. *Essays on Equilibrium Computation, MDD-based Constraint Programming and Scheduling*. PhD thesis, Carnegie Mellon University, 2010.

[17] J. N. Hooker. Job Sequencing Bounds from Decision Diagrams. In *Proceedings of CP*, volume 10416 of *LNCS*, pages 565–578, 2017.

[18] M. Horn. *Advances in search techniques for combinatorial optimization: New anytime A* search and decision diagram based approaches*. PhD thesis, Technische Universität Wien, 2021. https://doi.org/10.34726/hss.2021.96303.

[19] M. Horn, J. Maschler, G. R. Raidl, and E. Rönnberg. A*-based construction of decision diagrams for a prize-collecting scheduling problem. *Comput. Oper. Res.*, 126:105125, 2021. URL https://doi.org/10.1016/j.cor.2020.105125.

[20] S. W. Jeong and F. Somenzi. *A New Algorithm for 0-1 Programming Based on Binary Decision Diagrams*, volume 212 of *The Kluwer International Series in Engineering and Computer Science*, pages 145–165. Springer, Boston, MA, 1993.

[21] R. Kuroiwa and J. C. Beck. Domain-Independent Dynamic Programming: Generic State Space Search for Combinatorial Optimization. In S. Koenig, R. Stern, and M. Vallati, editors, *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, July 8-13, 2023, Prague, Czech Republic*, pages 236–244. AAAI Press, 2023. doi: 10.1609/ICAPS.V33I1.27200. URL https://doi.org/10.1609/icaps.v33i1.27200.

[22] R. Kuroiwa and J. C. Beck. Solving Domain-Independent Dynamic Programming Problems with Anytime Heuristic Search. In S. Koenig, R. Stern, and M. Vallati, editors, *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, July 8-13, 2023, Prague, Czech Republic*, pages 245–253. AAAI Press, 2023. doi: 10.1609/ICAPS.V33I1.27201. URL https://doi.org/10.1609/icaps.v33i1.27201.

[23] Y. Lai, M. Pedram, and S. B. K. Vrudhula. Evbdd-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 13(8):959–975, 1994.

[24] G. Perez. *Decision Diagrams: Constraints and Algorithms*. PhD thesis, Université Côte d'Azur, 2017.

[25] G. Perez and J.-C. Régin. Efficient Operations On MDDs for Building Constraint Programming Models. In *Proceedings of IJCAI*, pages 374–380. AAAI Press, 2015.

[26] L. Perron and V. Furnon. Or-tools. URL https://developers.google.com/optimization/.

[27] L. Perron, F. Didier, and S. Gay. The cp-sat-lp solver. In R. H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:2, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-300-3. doi: 10.4230/LIPIcs.CP.2023.3. URL https://drops.dagstuhl.de/opus/volltexte/2023/19040.

[28] I. Rudich, Q. Cappart, and L.-M. Rousseau. Peel-And-Bound: Generating Stronger Relaxed Bounds with Multivalued Decision Diagrams. In *Proceedings of CP*, volume 235 of *LIPIcs*, pages 35:1–35:20, 2022.

[29] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, Cambridge, Mass., 2009.

[30] W.-J. van Hoeve. Graph coloring with decision diagrams. *Math. Program.*, 192(1):631–674, 2022. doi: 10.1007/S10107-021-01662-X. URL https://doi.org/10.1007/s10107-021-01662-x.

[31] W.-J. van Hoeve. An Introduction to Decision Diagrams for Optimization. In *INFORMS TutORials in Operations Research*. INFORMS, 2024.

[32] T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Explicit and implicit algorithms for binate covering problems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 16(7):677–691, 1997.