TUTORIALS IN
OPERATIONS RESEARCH
INFORMS 2024

 | ISBN 978-0-xxxxxxx-x-x

An Introduction to Decision Diagrams for Optimization

Willem-Jan van Hoeve
Tepper School of Business, Carnegie Mellon University, vanhoeve@andrew.cmu.edu

Abstract This tutorial provides an introduction to the use of decision diagrams for solving discrete optimization problems. A decision diagram is a graphical representation of the solution space, representing decisions sequentially as paths from a root node to a target node. By merging isomorphic subgraphs (or equivalent subproblems), decision diagrams can compactly represent an exponential solution space. This ability can reduce solving time and memory requirements potentially by orders of magnitude. That said, exact decision diagrams can still be of exponential size for a given problem, which limits their practical applicability to relatively small instances. However, recent research has introduced a *scalable approach* by compiling polynomial-sized relaxed and restricted diagrams that yield dual and primal bounds, respectively. These can be combined in an exact search to produce a generic decision diagram-based branch-and-bound method. This chapter describes how this approach provides a scalable solution method for state-based dynamic programming models. In addition, the chapter shows how this approach can be applied to, and embedded in, other computational paradigms including constraint programming, integer programming, and column elimination. After this chapter, readers will have an understanding of the basic principles of decision diagram-based optimization, an appreciation of how it compares it to other optimization methods, and an understanding of what types of optimization problems are most suitable for this new technology.

Keywords discrete optimization; decision diagrams; integer programming; dynamic programming; constraint programming; network flows; branch and bound

1. Introduction

This tutorial provides an introduction to the use of decision diagrams for discrete optimization problems. Decision diagrams were first introduced in computer science to compactly represent Boolean functions through a graphical data structure for the purpose of Boolean circuit verification (Lee [57], Akers [2]). In the decades that followed, decision diagrams have been widely studied in the context of knowledge representation, Boolean satisfiability, formal methods, configuration problems, and many others. An overview of the foundations of decision diagrams as a data structure can be found in the books by Wegener [81] and Knuth [52].

Starting in the late 1990s, decision diagrams were applied to represent the discrete solution space of combinatorial optimization problems (Lai et al. [55], Wegener [81], Becker et al. [7], Hawkins et al. [43]). All these methods use decision diagrams to exactly represent the solution set to a given problem, which requires exponential memory in the worst case. In a seminal paper from 2007, Andersen, Hadžić, Hooker, and Tiedemann [3] introduced the concept of *relaxed* decision diagrams of polynomial size to define discrete relaxations for combinatorial problems, in the context of constraint programming. This started a new line of research by Bergman, Cire, Van Hoeve, and Hooker [13], who developed a novel branch-and-bound method where relaxed decision diagrams provide dual bounds, restricted decision

diagrams provide primal bounds, and an exact search is defined based on the nodes of the diagrams. Since then, decision diagram-based optimization methods have been applied to integer linear programming, integer nonlinear programming, constraint programming, stochastic programming, and to application areas ranging from machine scheduling and vehicle routing to portfolio optimization and workforce planning (Castro et al. [25]).

We will describe the main developments in this area, starting with the connection of decision diagrams and discrete optimization in Section 2. We then present the decision diagram-based branch-and-bound method of Bergman et al. in Section 3. This is followed by an overview of decision diagram-based constraint programming, in Section 4. Section 5 combines decision diagram-based constraint programming and decision-diagram based optimization for scheduling and routing problems. Section 6 discusses the use of decision diagrams in the context of integer programming. In Section 7 a new solution method called column elimination is presented that iteratively strengthens a decision diagram relaxation, and which is closely related to column generation. A summary is given in Section 8.

2. Decision Diagrams and Discrete Optimization Problems

We present basic decision diagram terminology, connect this to discrete optimization problems, and provide assumptions and useful properties of decision diagrams for optimization problems.

Decision Diagram Definitions We first provide the canonical definition of decision diagrams, following (Bryant [22]). While we will *not* directly use these definitions for the purpose of optimization, they are still relevant as the formal foundation. Additionally, it will help readers that are familiar with traditional decision diagrams make the connection to the use in an optimization context. Specifically, we will discuss why and how decision diagrams for optimization are different from the canonical description.

A *binary decision diagram* (BDD) represents a Boolean function as an acyclic directed graph, with the nonterminal vertices labeled by Boolean variables and the leaf vertices labeled with the values 1 and 0. Each nonterminal vertex v is associated with a Boolean variable $\text{var}(v)$ and has two outgoing edges, $(v, \text{hi}(v))$ labeled with value 1, and $(v, \text{lo}(v))$ labeled with value 0. We can represent the Boolean function by associating a function f_v with every vertex v in the graph. For the leaf nodes, we define $f_1 = 1$ and $f_0 = 0$. For a non-terminal vertex v, we define $f_v = (\text{var}(v) \wedge f_{\text{hi}(v)}) \vee (\neg\text{var}(v) \wedge f_{\text{lo}(v)})$. Lastly, we identify a *root vertex* r representing the value of the original Boolean function as f_r. The result is a recursive reformulation that is a systematic application of Boole's expansion theorem (Boole [19]), also known as the Shannon expansion of a Boolean function.

In an *ordered* binary decision diagram (OBDD), we enforce the condition that the variables follow an arbitrary but fixed ordering rule as they are associated with vertices from the root to a terminal. That is, for each edge (u, w) connecting two nonterminal vertices u and v, we have that $\text{var}(u)$ comes before $\text{var}(v)$ in the ordering. In a *reduced* ordered binary decision diagram (ROBDD), we additionally impose that there cannot be two vertices that represent the same function. A key result is that for a given Boolean function and a given variable ordering, there exists a *unique* ROBDD representation for that function (Bryant [20, 21]). Moreover, there exist efficient algorithms to convert an OBDD into an ROBDD.

Example 1. The BDD in Fig. 1 represents the Boolean function $(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_3)$. The root vertex v_1 is associated with the function $f_{v_1} = (x_1 \wedge f_{v_2}) \vee (\neg x_1 \wedge f_{v_3})$. The functions f_{v_2} and f_{v_3} are similarly recursively expanded until one of the terminal vertices v_5 or v_6 is reached. This is an ordered BDD because the variables associated with the vertices are ordered; in this case, lexicographically. The BDD is also reduced, because each vertex v correspond to a unique function represented by the subgraph rooted at v.

The concepts above for binary decision variables can be generalized to functions over variables with multiple values instead of only binary decisions. Namely, if a variable has a

FIGURE 1. The ROBDD representing $(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_3)$, taken from (Bryant [22]).

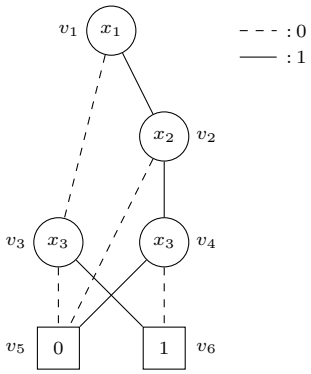
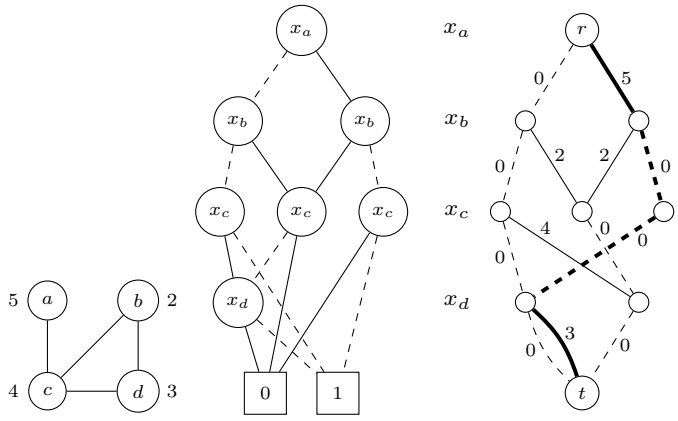


FIGURE 2. BDDs for the independent set problem.



a. Input graph b. ROBDD c. BDD for optimization

discrete set of possible values (its domain), we can define an edge $(v, \text{val}_j(v))$ for each value j in the domain of $\text{var}(v)$. The result is called a *multi-valued decision diagram* or MDD. Furthermore, decision diagrams can have more than two terminals to represent functions that distinguish multiple cases, e.g., for configuration problems (Wegener [81]).

Connection with Discrete Optimization There exists a natural connection between decision diagrams and discrete optimization problems, by interpreting the solution set of a discrete optimization problem as the result of a Boolean (or multi-valued) function returning value 1. We illustrate this with the classical NP-hard maximum independent set problem (Garey and Johnson [34]). Given a graph $G = (V, E)$ with vertex set V and edge set E, an *independent set* is a subset $S \subseteq V$ such that no two vertices in S share an edge, i.e., $(i, j) \notin E$ for all pairs $i, j \in S$. Given a weight w_i for $i \in V$, the *maximum independent set problem* asks to find an independent set of maximum total weight.

Example 2. Consider the graph in Fig. 2.a with vertex set $V = \{a, b, c, d\}$. We will focus first on enumerating all independent sets, and consider the optimization problem later. We can algebraically represent all independent sets as solutions to the integer linear model

$$\{(x_a, x_b, x_c, x_d) : x_i + x_j \leq 1 \ \forall (i, j) \in E, x_i \in \{0, 1\} \ \forall i \in V\}.$$

Alternatively, we can interpret the variables x to be Boolean-valued and define the Boolean function $f(x) = (\neg x_a \vee \neg x_c) \wedge (\neg x_b \vee \neg x_c) \wedge (\neg x_b \vee \neg x_d) \wedge (\neg x_c \vee \neg x_d)$. If $f(\boldsymbol{x})$ returns value 1 for a given truth assignment $\boldsymbol{x}$, this corresponds to the independent set $\{i : i \in V, \boldsymbol{x}_i = 1\}$. Fig. 2.b depicts the ROBDD representing f: any solution (truth assignment) $\boldsymbol{x}$ to the variables corresponds to a unique path from r to a terminal node, returning either false or true as the evaluation of $f(\boldsymbol{x})$. Specific for this example, paths from r to 1 represent all independent sets of G.

For our purposes, we are only interested in feasible solutions, and can therefore ignore the 0-terminal and all paths leading to it (see Fig. 2.c in which the 1-terminal is indicated by t). In addition, we need to express the objective function, being the sum of the weights of vertices in an independent set. We do this by associating a 'cost' value to each edge in the diagram: edge (u, v) has cost 0 if it represents decision 0, and value w_i if it represents value 1 and $\text{var}(u)$ is decision variable x_i for some $i \in V$. However, the ROBDD only represents the truth value of the Boolean function and arcs may skip variables; because many decisions are implicit we cannot express their costs. For this reason, the BDD in Fig. 2.c does not skip layers, but requires that edges are defined between consecutive layers. Doing so, each path from r to t is of the same length and represents an independent set of weight equal to the

sum of the associated edge costs. The longest r-t path corresponds to the optimal solution, in this case $S = \{a, d\}$ with weight 8 as indicated with bold edges in Fig. 2.c.

Assumptions for Optimization Problems There are important differences in how decision diagrams are built (or compiled) in the classical computer science literature and in the context of optimization. The classical definitions of BDDs were designed to efficiently manipulate and evaluate Boolean functions. In particular, as soon as a decision for vertex v evaluates to 1 or 0, the associated edge is directed to the respective terminal vertex, skipping any decision variable that has not yet been considered. We have seen in Example 2 that this may lead to challenges in representing the objective function for optimization problems, and we therefore imposed that edges must connect vertices in subsequent layers. More generally, imposing that edges are defined between consecutive layers gives the ability to provide an explicit mapping between solutions of an integer program and paths in the decision diagram, as each path is now of the same fixed length over the same ordered sequence of variables. This is particularly relevant when decision diagrams are integrated into integer programming or constraint programming solvers, in which case the decision diagram representation interacts with the original integer programming or constraint programming model.

This also relates to differences in how decision diagrams are *compiled*. Most classical methods apply a compositional approach to build larger BDDs from smaller ones, by first representing individual clauses as BDDs and then taking their conjunction. This is effective because each problem is a Boolean function, and the decisions are made without any semantic information. In contrast, in the context of decision diagram-based optimization, most methods follow a top-down compilation method that builds the diagram layer by layer from the root. One reason for this is that optimization problems are not presented as Boolean functions, but typically as state-based models or dynamic programming models. Such models do have semantic information, e.g., related to the evaluation of a cost function or resource constraints. By incorporating that information into the compilation method, the diagrams can be built much more efficiently.

For the above reasons, the decision diagram-based optimization literature usually mentions that edges can only connect subsequent layers. It is still possible to apply the concept of a reduced diagram in that case, by imposing that no two nodes *in a layer* can be equivalent. As an example, the BDD in Fig. 2.c is a reduced ordered BDD under the condition that edges only connect vertices in subsequent layers. However, when no direct link with an integer program or constraint program is required, it is often more efficient to compile decision diagrams with edges that do skip layers, as long as the appropriate problem structure can be represented.

3. Decision Diagram-Based Branch-and-Bound

This section describes the generic branch-and-bound framework based on decision diagrams by Bergman et al. [13]. Working from a state-based formulation (e.g., a dynamic program), this approach compiles relaxed decision diagrams to compute dual bounds (Bergman et al. [18, 14]), restricted decision diagrams to compute primal bounds (Bergman et al. [16]), and embeds these in an exact branch-and-bound search method (Bergman et al. [15]). We consider discrete optimization problems of the form

$$\begin{aligned} P: \quad & \max\ f(x) \\ & \text{s.t. } C_i(x), i = 1, \dots, m, \\ & \quad x \in D \end{aligned} \tag{1}$$

where $x = (x_1, \dots, x_n)$ is a tuple of n decision variables, f is a real-valued objective function over x, $C_1, \dots, C_m$ are constraints over x, and $D = D_1 \times \cdots \times D_n$ is the Cartesian product of the domains of the variables, i.e., $x_i \in D_i$ for $i = 1, \dots, n$. We assume that each domain is finite.

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 (Bellman [9]). It uses 'state variables' $s_1, s_2, \ldots, s_{n+1}$ and 'decision variables' $x_1, x_2, \ldots, x_n$, where a decision variable transitions one state into another state depending on the decision value (or 'label'). One typical form to represent DP models is a recursive algebraic formulation, which is common in operations research (Bellman [9]). Another form is a state-based formulation, i.e., a labeled transition system (Keller [50]). Bergman et al. [13] follow the latter approach, as it is more closely aligned with how decision diagrams are compiled. That is, a dynamic program for problem P consists of the following elements:

- A *state space* S with a *root state* r and a *terminal state* t. (In general we would allow multiple terminal states.) To facilitate notation, we also introduce an *infeasible state* $\hat{0}$ to represent infeasible solutions to P. The state space is partitioned into sets for each of the $n+1$ stages; i.e., S is the union of the sets $S_1, ..., S_{n+1}$, where $S_1 = \{r\}$ and $S_{n+1} = \{t, \hat{0}\}$, and $\hat{0} \in S_j$ for $j = 2, \ldots, n$.
- *Transition functions* $\tau_j : S_j \times D_j \to S_{j+1}$, for $j = 1, \ldots, n$, representing decision x_j as a transition between states in layers S_j and S_{j+1},
- *Transition cost functions* $h_j : S \times D_j \to \mathbb{R}$ for $j = 1, \ldots, n$, representing the cost of a decision,
- A root value $v_r \in \mathbb{R}$ accounting for objective function constants.

The DP formulation of P with variables $(s_1, \ldots, s_{n+1}, x_1, \ldots, x_n)$ has the following form:

$$\begin{aligned} \max\ & v_r + \sum_{j=1}^{n} h_j(s_j, x_j) \\ \text{s.t. } & s_{j+1} = \tau_j(s_j, x_j) && \text{for all } x_j \in D_j, j = 1, \ldots, n, \\ & s_j \in S_j && \text{for all } j = 1, \ldots, n+1. \end{aligned}$$

Observe that the objective function $f(x)$ and constraints $C_1, \ldots, C_m$ are captured through appropriately defining the state definitions and transition functions.

Example 3. Consider the maximum independent set problem on a weighted graph $G = (V, E)$ with node set $V = \{1, \ldots, n\}$ and edge set E, where each node $j \in V$ has a 'weight' w_j (see Example 2). We let $N(j)$ represent the set of neighbors of vertex j, i.e., $N(j) = \{j' : (j, j') \in E\} \cup \{j\}$. We have already seen an example BDD for this problem in Fig. 2.c. For the DP formulation, we maintain in each stage a set of *eligible* vertices $V_j \subseteq \{j, j+1, \ldots, n\}$ that can be added to the independent set so far constructed from the root up to stage j. When we include a vertex j, the transition removes that vertex and all its neighbors from the eligible set as we transition to stage $j+1$. To build the DP model, we introduce decision variables $x_j \in \{0, 1\}$ for $j \in V$ and state variables $s_1, \ldots, s_{n+1}$, and define the formulation as follows:

- State space: $S_j = 2^{V_j}$, root $r = V$, and terminal $t = \varnothing$.
- Transition functions: $\tau_j(s_j, 0) = s_j \setminus \{j\}$, and $\tau_j(s_j, 1) = \begin{cases} s_j \setminus N(j) & \text{, if } j \in s_j \\ \hat{0} & \text{, if } j \notin s_j \end{cases}$.
- Transition cost functions: $h_j(s_j, 0) = 0$, $h_j(s_j, 1) = w_j$.
- Root value: $v_r = 0$.

Exact, Relaxed, and Restricted Decision Diagrams Given a DP model, we can compile the associated decision diagrams similar to the state-transition graph in dynamic programming. That is, we define the decision diagrams as a weighted directed acyclic graph $\mathcal{D} = (N, A)$ where node set N corresponds to the set of state variables and arc set A corresponds to the state transition function τ. Each state $s \in S$ has an associated node in N (also

named s). For each transition $\tau(s,\ell) = s'$ with $s \in S_j$, $s' \in S_{j+1}$ in the DP model we define an arc $(s,s') \in A$ with associated label ℓ and weight $w(s,s') = h_j(s,\ell)$. The labels along each arc-specified path from the root node to the terminal node t correspond to a solution to P and vice versa. Furthermore, the longest r-t path corresponds to the optimal solution to P. By construction $\mathcal{D}$ is a layered graph, where layer $j = 1,2,\ldots,n$ corresponds to decision x_j and layer $n+1$ is the terminal layer.

Let $\mathrm{Sol}(\mathcal{D})$ denote the set of solutions represented by the decision diagram $\mathcal{D}$, and let $\mathrm{Sol}(P)$ 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 $\mathcal{D}$ is *exact* w.r.t. P if $\mathrm{Sol}(\mathcal{D}) = \mathrm{Sol}(P)$ and $w(p) = f(x^p)$ for all r-t paths p in $\mathcal{D}$.

Exact decision diagrams can be compiled in a 'top-down' manner starting at the root state, by recursively expanding the state space according to the transition function, while merging equivalent states. If the exact decision diagram fits into computer memory, we can directly solve P by computing the longest path. However, because the exact decision diagram can grow exponentially large in n, its size often prohibits a complete compilation. 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 $\mathcal{D}$ is *relaxed* w.r.t. P if $\mathrm{Sol}(\mathcal{D}) \supseteq \mathrm{Sol}(P)$ and $w(p) \geq f(x^p)$ for all r-t paths p in $\mathcal{D}$ such that $x^p \in \mathrm{Sol}(P)$.

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 $S' \subset S$, 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 nor underestimates the objective function. 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 in each layer. Note that the merge operator changes the definition of the dynamic program: the newly merged states need to be added to S if they were not defined recursively before by the DP model.

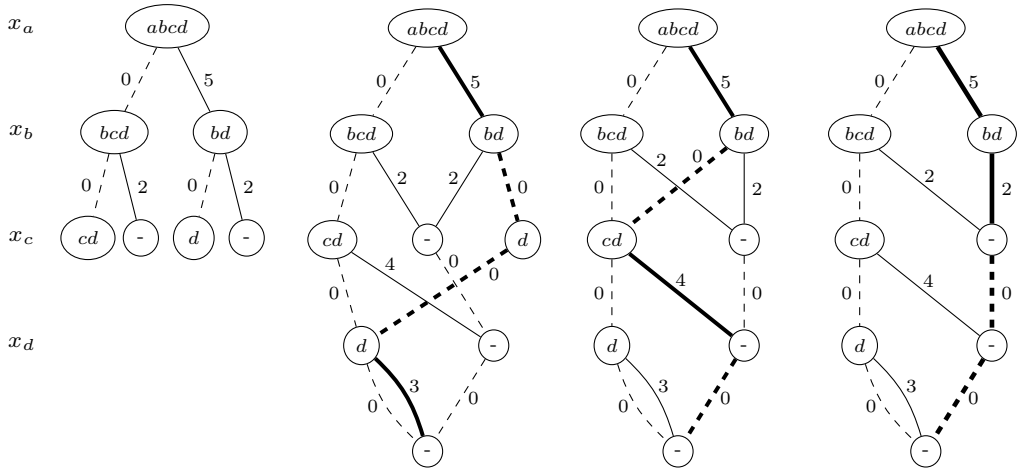
To find primal (heuristic) solutions, we can use *restricted decision diagrams* that represent a subset of the feasible solutions:

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

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.

Example 4. We illustrate the top-down compilation of the decision diagram for the DP model in Example 3 in Fig. 3.a. The compilation applies the fixed lexicographic variable ordering x_a, x_b, x_c, x_d. The root node, representing decision x_a, has associated state variable $V = \{a,b,c,d\}$ representing the initial set of eligible vertices. Applying the transition function to this state yields the two states $\{b,c,d\}$ (for $x_a = 0$) and $\{b,d\}$ (for $x_a = 1$) in the layer associated with decision x_b. By recursively applying the transition function, we arrive at the layer for x_c in which two states are equivalent (representing the empty set) and can hence be merged. The merged state is depicted in Fig. 3.b, which also shows the complete compilation of the decision diagram. (In these diagrams, we do not depict arcs leading to the infeasible state $\hat{0}$.) Each arc has an associated weight that is indicated in the figure. The longest path, depicted with bold edges, corresponds to the optimal solution $(x_a, x_b, x_c, x_d) = (1,0,0,1)$ with total weight 8.

FIGURE 3. Top-down compilation and the exact, relaxed, and restricted decision diagrams for the maximum independent set problem of Example 3 using the input graph in Fig. 2a.



a. Top-down compilation b. Exact diagram c. Relaxed diagram d. Restricted diagram

Observe that the width (the size of the largest layer) of the exact diagram is 3. Suppose we now impose a maximum width limit of two nodes and wish to create a relaxed decision diagram. Fig. 3.a shows that we first exceed the width limit for the layer associated with x_c, having four nodes. Because the two states representing the empty set are merged, we are left with three nodes. We can heuristically select any two states to be merged to reach the maximum width 2, for example $\{c, d\}$ and $\{d\}$. As merging operator, we define the *union* of their respective sets:

$$\oplus(\{s_1, s_2, \ldots, s_k\}) = \cup_{i=1}^{k} s_i.$$

This operator ensures that no solution is lost, but we may introduce non-solutions. Applying it to our example, our merged state is $\{c, d\} \cup \{d\} = \{c, d\}$. Fig. 3.c shows how the arc with label 0 from state $\{b, d\}$ in the layer for x_b is now directed to the new merged state $\{c, d\}$ in the next layer. Further top-down compilation does not exceed the width limit, resulting in the presented relaxed diagram. The longest path is now $(x_a, x_b, x_c, x_d) = (1, 0, 1, 0)$ providing a dual bound of value 9. It corresponds to $\{a, c\}$ which is not an independent set.

Lastly, we compile a restricted diagram of maximum width 2. As before, we consider the layer for decision x_c in Fig. 3.c. We can heuristically select any state, say $\{d\}$ and simply discard it from the layer. Further top-down compilation does not exceed the width limit, resulting in the restricted diagram presented in Fig. 3.d. Its longest path is $(x_a, x_b, x_c, x_d) = (1, 1, 0, 0)$ providing a primal bound of value 7. It corresponds to $\{a, b\}$ which is indeed an independent set but not optimal.

An alternative to the top-down compilation method above is *compilation by separation*, first proposed by Hadzic et al. [42] under the name 'incremental refinement'. In this method, we start with a decision diagram of width one and iteratively refine the diagram by separating out paths via new nodes. The initial decision diagram can be defined by applying the top-down compilation method with a limit of one node per layer, i.e., in each layer we apply the merging operator until one node remains. We then iteratively compute an optimal solution (the longest r-t path) and inspect whether it violates a constraint or overestimates the objective function. If not, we found the optimal solution to the original problem. Otherwise, we separate the path as follows. Starting from the root, we follow the arc-specified path. In each layer, we split the path by redirecting the solution arc (u, v) from node v to a newly created node v', thus creating arc (u, v'). We copy all arcs (v, w) to become arcs

(v', w), but remove any arc (v', w) that is not feasible. We will see examples of this algorithm in Section 4 (constraint programming), Section 5 (scheduling and routing), and Section 7 (column elimination).

Branch-and-Bound Search To obtain an exact solution, we follow a branch-and-bound search process. We start by compiling a relaxed and restricted decision diagram, resulting in a dual and primal bound. We then identify an *exact r-t cutset* of nodes $C \subset N$ in the relaxed decision diagram. Typical choices are the *frontier cutset* or the *last exact layer* (Bergman et al. [13]). Nodes in C must all be exact, i.e., they are not the result of the merge operator. With each node $u \in C$ we record the value of the longest r-u path as a constant to be added to its objective function.

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

Example 5. Continuing our running example, the last exact layer in the relaxed decision diagram depicted in Fig. 3.c is the layer associated with decision x_b having states $\{b, c, d\}$ (with value 0) and $\{b, d\}$ (with value 5). We can identify a subproblem with both states, and continue the process recursively. Doing so, we will find the optimal solution $(x_a, x_b, x_c, x_d) = (1, 0, 0, 1)$ with weight 8.

A recent alternative method, called 'peel-and-bound', integrates ideas from compilation by separation into the branch-and-bound process (Rudich et al. [67, 68]). Instead of using an exact cut-set to define the branch-and-bound search, it branches on single exact nodes while the search queue *stores diagrams*. When a new diagram is compiled, it can re-use the sub-diagram that was already compiled in its parent diagram, and separate those nodes in a similar fashion as compilation by separation. A key advantage of this approach is that it reduces the repeated compilation of the same sub-diagrams that is common when using top-down compilation with the cut-set branch-and-bound.

Computational Performance Bergman et al. [13] evaluated the decision diagram-based branch-and-bound method on three classical combinatorial optimization problems, the maximum independent set problem, the maximum cut problem, and the MAX-2SAT problem, obtaining competitive results. In several cases their approach outperformed a state-of-the-art integer programming solver, and for the maximum cut problem improved bounds were found for open benchmark instances. Gillard et al. [38, 37] developed the generic decision diagram-based optimization solver Ddo, which has been applied to a range of additional applications including Single-Row Facility Layout Problems (Coppé et al. [31]). The same approach has been implemented as the core methodology of the industrial solver maintained by the company NextMv,[1] that is particularly focused on scheduling and vehicle routing applications (O'Neil and Hoffman [62]). Lastly, the decision-diagram based branch-and-bound search is particularly suitable for parallelization (Bergman et al. [11]).

4. Constraint Programming with Decision Diagrams

Constraint programming (CP) solvers combine a systematic search with constraint propagation to find feasible or optimal solutions to combinatorial optimization problems (Rossi et al.

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

[66]). Decision diagrams can be naturally integrated into this solving process, by designing dedicated constraint propagation algorithms that are based on decision diagrams. One line of research in this area uses decision diagrams as a data structure for improved constraint propagation of individual constraints (Hawkins et al. [43], Cheng and Yap [26, 27], Perez and Régin [63], Verhaeghe et al. [80], Jung and Régin [47]). Another line of research extends the constraint propagation process by communicating (relaxed) decision diagrams between constraints, unlocking the potential of exponential reductions in the search tree size (Andersen et al. [3]). The latter approach will be the primary topic of this section.

Constraint Programming Methodology A *constraint optimization problem*, or COP, is defined by a set of decision variables $X = \{x_1, x_2, \ldots, x_n\}$, a set of constraints C defined over (subsets of) X, and an objective function $f : X \to \mathbb{R}$. Each variable x_i has an associated *domain* D_i of possible values, for $i = 1, \ldots, n$. An optimal solution to the COP is a variable assignment that satisfies all constraints and optimizes the objective function. If no objective function is given, the formulation is called a *constraint satisfaction problem*, or CSP.

What distinguishes CP from similar methodologies such as mixed integer programming (MIP), is that the variables can range over a variety of domains (e.g., continuous, integers, discrete sets, intervals, graphs) and that the constraints and the objective function can be arbitrary relations or expressions. CP libraries also contain so-called *global constraints* that typically represent a (combinatorial) structure over a set of variables. An example of a global constraint is $\texttt{alldifferent}(x_1, \ldots, x_n)$, which specifies that variables $x_1, \ldots, x_n$ take distinct values. It is semantically equivalent to the set of pairwise not-equal constraints but represents a structure that allows for stronger constraint reasoning (Régin [64]). Global constraints are often key to the fast performance of constraint programming solvers (van Hoeve and Katriel [79], Régin [65]).

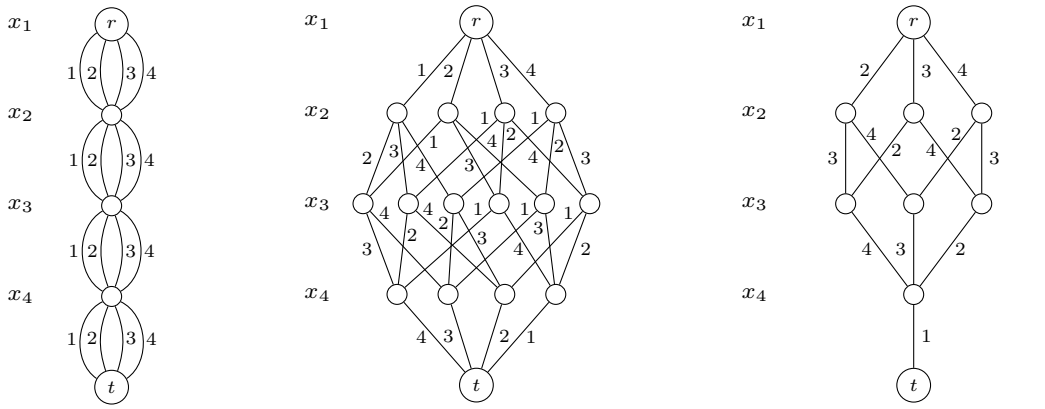
Because CP models need not be linear or differentiable, the core solving process of CP uses a mechanism that does not rely on such properties: constraint propagation. Each constraint has an associated algorithm that performs two functions: 1) feasibility checking when all variables are instantiated, and 2) eliminating domain values that are proven to be infeasible to that constraint. The latter is also referred to as *domain filtering*. When all domain values are part of a solution to a given constraint (and thus no domain values can be filtered), the constraint is said to be *domain consistent*. The propagation algorithm associated with a constraint ideally achieves domain consistency in polynomial time. For some constraints (e.g., certain scheduling constraints) this is not possible, and we need to be satisfied with propagation algorithms that may not identify all infeasible domain values but are computationally efficient. A CP solver applies domain filtering to each constraint and then *propagates* any updated variable domains to the other constraints that have that variable in their scope. This process of *constraint propagation* continues until a fixed point is reached. Under certain assumptions on the domain filtering algorithms, it can be shown that the fixed point is unique, regardless of the order in which the constraints are considered (Apt [4]).

Example 6. Consider the following CSP:

$$x_1 > x_2 \tag{2}$$
$$x_1 + x_2 = x_3 \tag{3}$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4) \tag{4}$$
$$x_1 \in \{1, 2\}, x_2 \in \{0, 1, 2, 3\}, x_3 \in \{2, 3\}, x_4 \in \{0, 1\}. \tag{5}$$

We apply domain filtering to the constraints in the order listed: Constraint (2) eliminates values $\{2, 3\}$ from D_2, resulting in $x_2 \in \{0, 1\}$. Constraint (3) has no impact. For constraint (4), observe that x_2 and x_4 both range over $\{0, 1\}$ and therefore x_1 cannot take value 1, i.e., $x_1 \in \{2\}$. It subsequently leads to eliminating value 2 from D_3, i.e., $x_3 \in \{3\}$. This

FIGURE 4. MDD-based constraint propagation for the CP model in Example 7.



a. Initial MDD b. Propagating `alldifferent` c. Propagating $x_1 + x_2 + x_3 \geq 9$

completes one round of propagation. Because the domains were updated, we repeat the process, ultimately resulting in the following updated domains:

$$x_1 \in \{2\}, x_2 \in \{1\}, x_3 \in \{3\}, x_4 \in \{0\}.$$

In the example above, constraint propagation resulted in finding the solution to the CSP. In most cases, however, constraint propagation alone is not sufficient to solve the problem. The propagation process is therefore embedded within a systematic search procedure, which is typically a depth-first search over the variables. After each search decision (or branch) the constraint propagation is again applied so as to prune the resulting search tree as much as possible. For many applications this approach can be very effective especially in the context of complex industrial applications such as machine scheduling with resource limitations and side constraints.

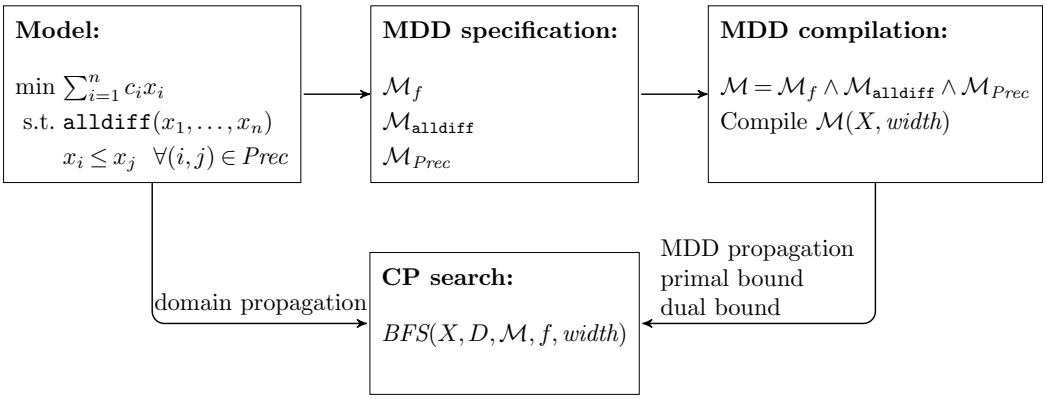
MDD-based Constraint Propagation As illustrated in Example 6, classical constraint propagation communicates domains from one constraint to the next, using a central *domain store*. A drawback of this approach is that more refined structural relations between variables captured by a (global) constraint are not visible to other constraints, while they could be useful. The idea behind MDD-based constraint propagation is to additionally utilize an *MDD store* to propagate higher-order information between constraints (Andersen et al. [3]). This is illustrated in the following example.

Example 7. Consider the following CSP:

$$\texttt{alldifferent}(x_1, x_2, x_3, x_4) \tag{6}$$
$$x_1 + x_2 + x_3 \geq 9 \tag{7}$$
$$x_i \in \{1, 2, 3, 4\}, \text{ for } i = 1, \ldots, 4. \tag{8}$$

One can inspect that individually the constraints (6) and (7) do not permit to identify any infeasible domain value; i.e., the constraints are domain consistent. However, if x_4 would take any value other than 1, the constraints cannot be simultaneously satisfied. We can deduce this information using MDD-based constraint propagation. We define a multivalued decision diagram which initially represents the Cartesian product of the variable domains (see Fig. 4.a). This MDD functions as our MDD store and we let constraint (6) refine it to represent all its solutions (see Fig. 4.b). We then propagate the updated MDD to constraint (7) which further refines it, resulting in the MDD in Fig. 4.c. As expected, this MDD restricts x_4 to take only the value 1. We can communicate the MDD information to the domain store

FIGURE 5. Overview of automatic MDD-based constraint programming in HADDOCK on an example constraint optimization problem $\langle X, D, C, f \rangle$ with variables X, domains D, constraints C and objective function f. The constraint programming search employs a branch-and-bound best-first strategy (BFS). The MDD specification and compilation are derived automatically from the model declaration. The illustration is taken from (Gentzel et al. [35]).



by projecting the domain values along each layer onto the variable domains, resulting in $D_1 = D_2 = D_3 = \{2, 3, 4\}$ and $D_4 = \{1\}$. Observe that the domain store (similar to the MDD of width 1 in Fig 4.a) implicitly represents $4^4 = 256$ solutions. After propagating `alldifferent` only 24 solutions are left, and subsequently propagating constraint (7) results in an MDD representing just 6 solutions.

MDD-based constraint propagation was first introduced by Andersen et al. [3]. They recognized that sharing this structural information between constraints has enormous potential to improve constraint propagation, thereby reducing the search tree size. The caveat is that the MDDs can grow exponentially large, even when representing an individual constraint such as the `alldifferent` constraint. They therefore introduced the concept of a *relaxed* MDD of polynomial size by restricting its width (as in Definition 2). This opened a new research direction for studying and developing MDD propagation algorithm for a variety of constraint types (Hoda et al. [44], Bergman et al. [13]).

A central concept in this research is that of *MDD consistency*, which is the MDD analog of domain consistency. Consider a constraint $C(X)$ over a set of variables X and a decision diagram $\mathcal{D}$ over the variables X. We say that C is *MDD-consistent* with respect to $\mathcal{D}$ if every arc in $\mathcal{D}$ belongs to a path that is feasible for C. Similar to domain propagation algorithms, for some constraints it is not possible to establish MDD consistency in polynomial time; one example is the `alldifferent` constraint. In such cases, efficient MDD propagation algorithms may be designed that can still be effective. MDD propagation algorithms can also be applied to strengthen decision diagram representations in contexts other than CP. For example, Tjandraatmadja and van Hoeve [74] apply MDD propagation to decision diagrams for use in an integer programming solver.

All known MDD propagation algorithms can be expressed in terms of a generic state-based framework that associates a separate DP-style transition system with each constraint (Hoda et al. [44]). The propagation algorithm applies the state-transition functions to systematically update states and eliminate infeasible arcs. This framework forms the basis of the HADDOCK system that integrates MDD propagation into a CP solver (Gentzel et al. [36]). HADDOCK can automatically compile MDDs from a given CP model description by converting each constraint into a state-based representation. Moreover, HADDOCK allows to generically define MDD filtering algorithms for each constraint, and combine them into an

MDD propagation process. The MDD propagation is part of the overall constraint propagation algorithm as well as the systematic search. A schematic illustration of Haddock is shown in Fig 5.

Because MDDs can be used to evaluate an objective function, as we have seen in Section 3, MDD-based constraint programming offers a new perspective of handling optimization problems. Indeed, the Haddock system can automatically integrate an objective function into the MDD, again using a state-based representation, and use it to compute dual and primal bounds for the CP solver (Gentzel et al. [35]). We refer to the survey paper (Castro et al. [25]) and the textbook (Bergman et al. [13]) for more information on MDD-based constraint programming.

5. Decision Diagrams for Scheduling and Routing

One of the earliest successful applications of decision diagram-based optimization was in the context of sequencing problems such as single machine scheduling and the traveling salesman problem with time windows (Ciré and van Hoeve [29]). That work combines ideas from MDD-based constraint programming and decision-diagram based optimization. This section presents the basic representation, problem variants, and methodological extensions for this problem domain.

Sequencing Problems We consider sequencing problems as *constraint-based scheduling* problems over a discrete and finite time horizon H with the following elements (Baptiste et al. [5]):

- A set of *activities* $\mathtt{A}$ that need to be scheduled. We associate a non-negative processing time p_i, release time $r_i \in H$, and deadline $d_i \in H$ with each activity $i \in \mathtt{A}$.
- A non-preemptive resource that can process at most one activity at a time.
- Decision variables s_i representing the start time for activity $i \in \mathtt{A}$, with domain $D(s_i) \subseteq \{r_i, \ldots, d_i\}$.

We also introduce an auxiliary variable e_i representing the end time of activity i, together with the invariant relationship $s_i + d_i = e_i$, for all $i \in \mathtt{A}$. In addition to these basic elements, we consider:

- A set of *precedence relations* $Prec$ of the form $i \ll j$ representing that activity i must be executed before j, where $i, j \in \mathtt{A}$.
- Sequence-dependent setup times t_{ij} for $i, j \in \mathtt{A}$. If i has j as its immediate successor, then there must be at least t_{ij} time units between the completion of i and the start of j.
- Various objective functions, including makespan (the end time of the last activity), sum of setup times, (weighted) sum of completion times, (weighted) tardiness, and number of late jobs (by interpreting the deadline as a due date) can be considered.

Example 8. Let V be a set of locations, and let d_{ij} be the distance between each ordered pair $(i, j) \in V \times V$, $i \neq j$. Thus, the distances are not necessarily symmetric. The asymmetric traveling salesman problem (ATSP) asks to find a closed tour of minimum total distance that visits each location exactly once. We consider two variants, both of which assume that we designate one location as the 'depot' that functions as start and end of the tour:

- The *asymmetric traveling salesman problem with time windows* (ATSPTW) is an ATSP for which we are additionally given a time window $[l_i, u_i]$ such that location i must be visited between time l_i and u_i. We assume here that the distances d_{ij} reflect time units.
- The *sequential ordering problem* (SOP) is an ATSP for which we are additionally given a set of ordered precedence relations $P \subseteq V \times V$. If $(i, j) \in P$ then location i must be visited before location j is visited.

We will represent these variants as scheduling problems using the sequencing formalism above. We assume that the location set is $V = \{1, 2, \ldots, n\}$, designating location 1 as the depot. We define an activity i for each location $i \in \{2, \ldots, n\}$. For the depot, we introduce two activities 1^{start} and 1^{end} to represent the start and the end of the tour. The duration for each activity is 0 time units. The distances are represented as sequence-dependent setup times, i.e., $t_{ij} = d_{ij}$ for all $i, j \in V$. As objective function we can either minimize the sum of the setup times, or minimize the makespan represented by the end time of 1^{end}. This completes the sequencing model for the basic ATSP.

For the ATSPTW, we define $r_i = l_i$ and $d_i = u_i$ as release date and deadline for each activity $i \in \{2, \ldots, n\}$, while activities 1^{start} and 1^{end} have $\min\{H\}$ as release date and $\max\{H\}$ as deadline. For the SOP, we define a precedence constraint $i \ll j$ for each precedence relation $(i, j) \in P$. Because the time window constraints in the ATSPTW may imply wait times at the locations, the objective functions are no longer equivalent: minimizing makespan includes wait times while the sum of the setup times only consider the actual travel time.

Natural extensions to these variants are a combination of time windows and precedence relations, immediate predecessor/successor constraints, processing times, etcetera. All of these can be expressed in the underlying constraint-based scheduling framework.

Decision Diagram Representation As shown in (Ciré and van Hoeve [29]), sequencing problems can be naturally represented in a decision diagram via the permutation model. That is, we view a solution as a permutation of the set of activities $\mathtt{A}$. Given a permutation $\pi : \mathtt{A} \to \mathtt{A}$, the implied schedule can be inferred from the relation $s_{\pi_i} \geq s_{\pi_{i-1}} + t_{\pi_{i-1},\pi_i} + p_{\pi_{i-1}}$. The aim is to represent all feasible permutations compactly in an MDD such that the decisions along each r-t path correspond to a feasible ordered sequence of activities. We let $\ell(a) \in \mathtt{A}$ represent the label (decision) associated with arc a in the MDD.

Because these MDDs can grow exponentially large in the worst case, we again utilize relaxed MDDs of limited width that provide a polynomial size over-approximation of the set of solutions. We can build the MDD using a compilation method that combines constraint propagation and node splitting, similar to compilation by separation. It takes the sequencing model as input and associates the following information with each node v in the MDD:

- The set $\mathcal{A}_v$ represents the set of activities that are taken on *all* paths from r to v.
- The set $\mathcal{S}_v$ represents the set of activities that are taken on *some* paths from r to v.
- The integer E_v denotes the earliest completion time of all paths from r to v.
- The integer L_v denotes the latest completion time of all paths from r to v.

Given an MDD, each of these can be computed recursively in a top-down manner starting from the root node r. That is, for each node $v \neq r$ with incoming arc set $\delta^-(v)$ we compute

$$\mathcal{A}_v = \bigcap_{(u,v)\in\delta^-(v)} \mathcal{A}_u \cup \{\ell(u,v)\},$$
$$\mathcal{S}_v = \bigcup_{(u,v)\in\delta^-(v)} \mathcal{S}_u \cup \{\ell(u,v)\}.$$

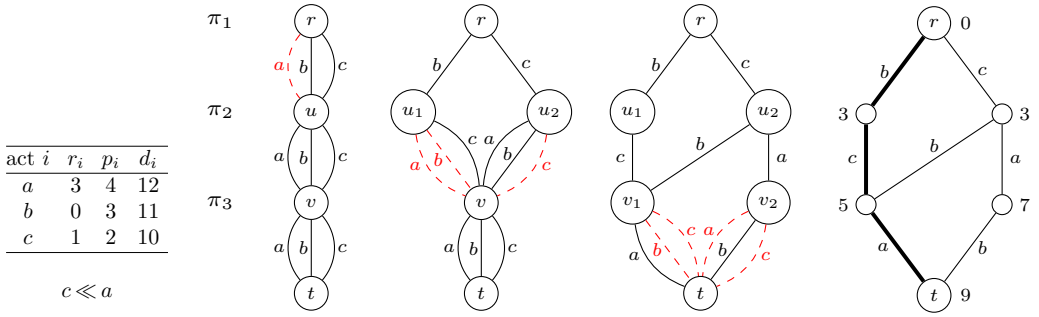
For computing E_v and L_v, recall that the completion time of an activity depends on the completion time of the previous activity, the setup time, and the problem parameters: $e_{\pi_i} = \max\{r_{\pi_i}, e_{\pi_{i-1}} + t_{\pi_{i-1},\pi_i}\} + p_{\pi_i}$. We thus have

$$E_v = \min_{(u,v)\in\delta^-(v)} \left\{\max\{r_{\ell(u,v)}, \min_{(u',u)\in\delta^-(u)} E_u + t_{\ell(u',u),\ell(u,v)}\} + p_{\ell(u,v)}\right\},$$
$$L_v = \max_{(u,v)\in\delta^-(v)} \left\{\max\{r_{\ell(u,v)}, \max_{(u',u)\in\delta^-(u)} E_u + t_{\ell(u',u),\ell(u,v)}\} + p_{\ell(u,v)}\right\}.$$

The constraint propagation process considers an arc (u, v) in the MDD and determines whether assigning $\ell(u, v)$ violates any constraint and causes arc (u, v) to be infeasible. For example:

- For the permutation (or `alldifferent`) constraint: if $\ell(u, v) \in \mathcal{A}_u$ then (u, v) is infeasible.

FIGURE 6. The sequencing model and associated MDD compilation for Example 9. Dashed arcs indicate infeasible decisions and can be removed.



a. Model b. Layer 1 c. Layer 2 d. Layer 3 e. Final MDD

- For precedence constraints: if $(i,j) \in Prec$ and $\ell(u,v) = j$ but $i \notin \mathcal{S}_u$ then (u,v) is infeasible.
- For time window constraints: if $r_{\ell(u,v)} > L_u$ or if $E_u + p_{\ell(u,v)} > d_{\ell(u,v)}$ then (u,v) is infeasible.

Additional constraint propagation rules are described in (Ciré and van Hoeve [29], Bergman et al. [13]).

The MDD compilation proceeds in stages, considering layers $i = 1, 2, \ldots, n$ in turn:

- *Propagation:* Remove infeasible arcs from layer i based on the MDD propagation rules.
- *Node splitting:* If the number of nodes in layer i is less than the maximum width, split any node u that has multiple incoming arcs into separate nodes, one for each arc. Create a copy of the arcs going out of u for each newly created node copy.

This process has several heuristic choices; the selection of which nodes to split is often the most important. These are discussed in in (Ciré and van Hoeve [29], Bergman et al. [13]).

Example 9. Consider the sequencing problem with activity set $\mathtt{A} = \{a, b, c\}$, time horizon $H = \{0, 1, \ldots, 12\}$, and model parameters in Fig. 6.a. In addition to the release date r_i, processing time p_i, and deadline d_i, for $i \in \mathtt{A}$, the model contains one precedence constraint $c \ll a$. We assume that the objective is to minimize the makespan.

The MDD compilation process starts with the width-1 MDD in Fig. 6.b. Starting at the root node in layer 1, it identifies and removes the infeasible outgoing arc with label a, based on the precedence constraint. It then splits node u in the next layer into two nodes u_1 and u_2, copying the outgoing arcs of u for each. Constraint propagation in layer 2 determines that the arc (u_1, v) with label a is infeasible (again based on $c \ll a$) and the arc (u_1, v) with label b is infeasible because of the `alldifferent` constraint ($b \in \mathcal{A}_{u_1}$). Simularly, the arc (u_2, v) with label c is infeasible because $c \in \mathcal{A}_{u_2}$. We proceed by splitting node v into two nodes v_1, v_2. Observe that $\mathcal{A}_{v_1} = \{b, c\}$ which means that the arcs (v_1, t) with labels b and c are infeasible. Likewise for arcs (v_2, t) with labels a and c.

The resulting MDD in Fig. 6.e is exact with respect to the permutation and time window constraints. To evaluate the objective function, we consider the earliest completion time at each node (indicated in the figure) and determine that the shortest path, corresponding to sequence $[b, c, a]$ is the optimal solution with makespan 9.

Computational Performance Compiling the MDD using the permutation representation has two main advantages. First, the variable ordering is implicit and fixed: layer i corresponds to the i-th activity to be sequenced. Second, it allows for a natural computation of earliest and latest completion times. An additional advantage is inherent in the design of

MDD-based optimization: We can compute both top-down and bottom-up information for each node in the MDD, which can strengthen the propagation rules for the arcs.

Cire and van Hoeve [29] demonstrated the computational benefits of their approach on a variety of sequencing problems, including the SOP, TSPTW, and objective functions such as weighted tardiness. They provided very competitive results, often outperforming the state-of-the-art constraint-based scheduling solver in ILOG CP Optimizer. Specifically, they closed three open SOP instances from the TSPLIB benchmark set for the first time. Extensions of their work include strengthening the MDD bounds with a Lagrangian relaxation (Bergman et al. [12], Hooker [45]), time-dependent travel times (Kinable et al. [51]), Pickup-and-Delivery TSP (O'Neil and Hoffman [62], Castro et al. [23]), and multi-machine scheduling (van den Bogaerdt and de Weerdt [75, 76]).

6. Decision Diagrams for Integer Programming

This section describes how decision diagrams can be used as a computational tool within integer programming solvers. One such application is using decision diagrams to generate cutting planes for integer linear or nonlinear programming problems. Other applications are the embedding of (relaxed) decision diagrams into integer programming models, or the use of BDD-based bounds in branch-and-bound solvers. A foundational element of these approaches is the polyhedral connection between decision diagrams and integer programming models to represent a set of solutions. This was first proposed in (Becker et al. [7], Behle [8]) who use decision diagrams for generating cutting planes in a branch-and-cut framework to solve binary linear programming problems.

A Polyhedral View of Decision Diagrams Consider the general integer programming problem

$$\max \left\{ c^T x : Mx \leq b, x \in \mathbb{Z}^n \right\} \tag{9}$$

where $c \in \mathbb{R}^n$, $M \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$, and x is a vector of integer decision variables. We define $P_I = \text{conv}(\{x \in \mathbb{Z}^n : Mx \leq b\})$ as the convex hull of all feasible integer points of (9). Given an arbitrary ordering of the variables x, the set of integer points in P_I can be represented by a decision diagram. We assume that such diagram has been compiled, as $\mathcal{D} = (N, A)$ with node set N, arc set A, root $r \in N$ and single terminal $t \in N$. Denote by $L(u)$ the index i of the layer of $u \in N$, corresponding to the associated decision variable x_i. Let $\ell(a)$ denote the label (an integer decision value) associated with arc $a \in A$.

We can reformulate the integer program (9) as a *network flow problem* over $\mathcal{D}$. For each arc $a \in A$ we introduce a 'flow' variable y_a. We also define a 'weight' $w_a = \ell(a)c_i$, where $i = L(u)$ for $a = (u, v)$. The network flow model is defined as

$$\max \sum_{a \in A} w_a y_a \tag{10}$$

$$\text{s.t.} \sum_{a \in \delta^-(u)} y_a - \sum_{a \in \delta^+(u)} y_a = 0, \qquad \forall u \in N \setminus \{r, t\}, \tag{11}$$

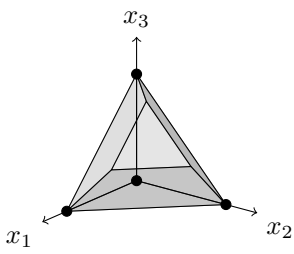
$$\sum_{a \in \delta^+(r)} y_a = 1, \tag{12}$$

$$\sum_{a \in \delta^-(t)} y_a = 1, \tag{13}$$

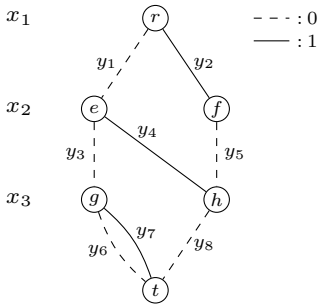
$$0 \leq y_a \leq 1, \qquad \forall a \in A, \tag{14}$$

where $\delta^-(u)$ and $\delta^+(u)$ are the sets of incoming and outgoing arcs for $u \in N$, respectively. In this model, we require that exactly one unit of flow leaves the root node r (constraint (12)) and enters the terminal node t (constraint (13)). For the other nodes $u \in N \setminus \{r, t\}$ we require

FIGURE 7. The polytope defined by the inequalities in Example 10 (a), the associated exact binary decision diagram (b), and the flow model representing the solutions in the BDD, together with the mapping to the original variables (c). Each of the four r-t paths in the BDD corresponds to an integer point in the polytope.



a. Polytope



b. BDD representation

$$
\begin{aligned}
&y_1 = y_3 + y_4\\
&y_2 = y_5\\
&y_3 = y_6 + y_7\\
&y_4 + y_5 = y_8\\
&y_1 + y_2 = 1\\
&y_6 + y_7 + y_8 = 1\\
&0 \le y_i \le 1,\ i \in \{1,\dots,8\}\\
&x_1 = 0y_1 + 1y_2\\
&x_2 = 0y_3 + 1y_4 + 0y_5\\
&x_3 = 0y_6 + 1y_7 + 0y_8
\end{aligned}
$$

c. Flow model and mapping

flow conservation, i.e., the total flow into u equals the total flow out of u (constraints (11)). Because the network flow model solutions are integral (Ahuja et al. [1]), each integer solution of (9) has a one-to-one correspondence with a r-t path of flow value 1 in $\mathcal{D}$. Given a solution y to (11)-(14), we can obtain the associated solution x for problem (9) by defining

$$x_i = \sum_{(u,v)\in A: L(u)=i} \ell(u,v) y_{(u,v)} \quad \forall i \in \{1,\dots,n\}. \tag{15}$$

Example 10. Consider the set of points described by

$$
\begin{aligned}
\{x \in \{0,1\}^3 : \quad & x_1 + x_2 - x_3 \le 1,\\
& x_1 - x_2 + x_3 \le 1,\\
& -x_1 + x_2 + x_3 \le 1,\\
& x_1 + x_2 + x_3 \le 2\}.
\end{aligned} \tag{16}
$$

Fig. 7.a shows the polytope defined by the inequalities in (16), containing the integer points $\{(0,0,0),(0,0,1),(0,1,0),(1,0,0)\}$. The same set of integer points is represented in the decision diagram in Fig. 7.b. The flow model associated with the diagram is presented in Fig. 7.c, together with the mapping to the original x-variables.

The one-to-one correspondence between solutions of the original integer program and the network flow model can be formalized in the following manner. First, define the polytope $P_{\text{flow}}(\mathcal{D}) = \{y \in \mathbb{R}^{|A|} : (11)\text{-}(14)\}$ corresponding to the feasible set of the network flow model. We add the equations (15) to $P_{\text{flow}}(\mathcal{D})$ and obtain a polytope $P(y,x) \subseteq \mathbb{R}^{|A|+n}$. The *projection* of $P(y,x)$ onto x is defined as

$$\text{Proj}_x(P(y,x)) := \{x \in \mathbb{R}^n : \exists y \in \mathbb{R}^{|A|} \text{ such that } (y,x) \in P(y,x)\}.$$

Behle [8] showed that the projection of $P(y,x)$ onto the x-space is precisely the convex hull of the original binary linear programming problem, i.e.,

$$P_I = \text{Proj}_x(P(y,x)).$$

This result is useful in that it shows that the decision diagram representation provides the tightest possible description of P_I, although in the worst case it needs an exponentially large exact diagram to represent the integer program.

Decision Diagrams for Cut Generation The most successful approach to solving integer programming problems is based on the branch-and-bound method (Land and Doig [56]). It relies heavily the strength of the continuous linear programming relaxation whose solution yields a dual bound. A fundamental approach to strengthening the linear programming relaxation is to (iteratively) add valid inequalities that eliminate a given fractional solution but do not remove any feasible integer solution (Conforti et al. [30]). We next describe how decision diagrams can be used for this purpose.

To start, suppose we are given a point x^* as a solution to the linear programming relaxation of the integer program (9). The *separation problem* is to decide whether x^* is inside the convex hull of integer feasible points, i.e., $x^* \in P_I$. This can be determined by finding a hyperplane $\alpha x \leq \beta$ that is satisfied by all points $x \in P_I$, while $\alpha x^* > \beta$. If so, we add the valid inequality $\alpha x \leq \beta$ to the linear program to cut off x^* and resolve the linear program. Here, we will assume that the fractional solution x^* is obtained from solving the linear program in the original space $x \in \mathbb{R}^n$, whereas the cuts are derived from a pre-compiled decision diagram in the space $y \in \mathbb{R}^{|A|}$, projected back onto the x-space.

One generic way to compute valid inequalities (in fact, a 'deepest' cut) is to use a cut-generating linear program, or CGLP (Conforti et al. [30]). It computes (α, β) such that $\alpha x^* - \beta$ is maximized while ensuring that no feasible integer solution is removed. In our case, we derive these cuts from the decision diagram $\mathcal{D}$ whose linear description, projected onto the original x-variable space, provides P_I. In fact, we can utilize the special network flow structure of the decision diagram, and use the dual formulation of the model (11)-(15). By using dual variables θ_u for $u \in N$ and γ_i for $i \in \{1, \ldots, n\}$, the CGLP can be defined as (Behle [8], Davarnia and van Hoeve [33]):

$$\max \sum_{i=1}^{n} x_i^* \gamma_i - \theta_t \tag{17}$$

$$\text{s.t. } \theta_u - \theta_v + \ell(u,v)\gamma_i \leq 0 \qquad i \in \{1, \ldots, n\}, \forall (u,v) \in A : L(u) = i, \tag{18}$$

$$\theta_r = 0, \tag{19}$$

$$\sum_{i=1}^{n} \gamma_i + \sum_{u \in N} \theta_u \leq 1, \tag{20}$$

where constraint (20) is a normalization constraint to ensure the solution is bounded. Let $(\bar{\theta}, \bar{\gamma})$ be a solution to model (17)-(20) with objective value β. Then $\bar{\gamma} x \leq \beta$ is a valid inequality for the integer program (9).

Example 11. We continue Example 10 and consider the integer program $\max\{x_1 + x_2 + x_3 : (16)\}$. We solve the linear programming relaxation and obtain the fractional solution $x^* = (x_1, x_2, x_3) = (1, \frac{1}{2}, \frac{1}{2})$ with objective value 2. To cut off x^*, we solve the CGLP and obtain the solution $(\gamma_1, \gamma_2, \gamma_3) = (\frac{1}{6}, \frac{1}{6}, \frac{1}{6})$, $(\theta_r, \theta_e, \theta_f, \theta_g, \theta_h, \theta_t) = (0, 0, \frac{1}{6}, 0, \frac{1}{6}, \frac{1}{6})$, with objective value $\beta = \frac{1}{6}$. This results in the valid inequality $\frac{1}{6}x_1 + \frac{1}{6}x_2 + \frac{1}{6}x_3 \leq \frac{1}{6}$, or $x_1 + x_2 + x_3 \leq 1$. Indeed this cut, together with the nonnegativity constraints, defines the convex hull of (16).

Different approaches have been proposed in the literature to solve the CGLP over $\mathcal{D}$. All of them exploit the network flow structure to obtain the dual values γ_i for the projection constraints (15). For example, in the context of binary programs, Becker et al. [7] use a Langrangian reformulation of (11)-(15) by relaxing the constraints (15) with Langrangian multipliers γ_i. The Lagrangian relaxation assigns a weight γ_i to each arc in layer i with label 1, and the resulting problem, for fixed γ, can be solved efficiently as a shortest path problem in $\mathcal{D}$. The optimal Lagrangian bound is then found using a subgradient search over γ. Tjandraatmadja and van Hoeve [73] take a different approach and formulate the valid inequalities as *target cuts* for which they develop a CGLP. The associated procedure generates cuts that are facet defining with respect to the convex hull of the integer points

represented by the decision diagram. Lastly, Davarnia and van Hoeve [33] reformulate model (17)-(20) as a bilevel program and develop an efficient projected subgradient method that only contains γ-variables at the higher level.

In practice, one would never use an *exact* decision diagram representing all solutions to the integer programming problem to derive cutting planes. Namely, if such a diagram can be compiled and fits in computer memory, the integer optimal solution can readily be found in polynomial time (in the size of the diagram). Otherwise, the diagram would be too large to be computationally practical. Becker et al. [7] therefore choose to a subset of constraints $M'x \leq b'$ for which an exact BDD $\mathcal{D}'$ of reasonable size can be compiled, to apply the separation procedure. Determining the best set of constraints to form $\mathcal{D}'$ is problem dependent. As an alternative, Tjandraatmadja and van Hoeve [73] propose the use of relaxed decision diagrams of polynomial size. Also, multiple different diagrams can be compiled and evaluated. Note that when the diagram $\mathcal{D}$ is not exact, a point x^* may be outside P_I and *inside* $P_{\text{flow}}(\mathcal{D})$. In such cases, the decision diagram will not be able to cut off that point. Determining under what conditions a relaxed decision diagram is guaranteed to separate a given point $x^* \notin P_I$ is an open question.

Nonlinear Programming Even though optimizing over decision diagrams corresponds to solving a linear objective function over the arcs in the diagram, the explicit representation of the discrete solutions as paths often allows to encode nonlinear relations effectively. One successful example is the work by Bergman and Cire [10] who consider binary optimization problems with nonlinear objectives and linear constraints. They introduce a decomposition approach that partitions the objective function into separate low-dimensional dynamic programming models, each of which can be equivalently represented as a shortest-path problem in a decision diagram. By reformulating the decision diagrams as linear network flow problems, the resulting model can be solved using standard mixed-integer programming (MIP) solvers. To make their method scalable, they apply relaxed decision diagrams of polynomial size, ensuring that the resulting MIP solver provides both lower and upper bounds. The computational benefits are strong, outperforming state-of-the-art approaches often by orders of magnitude on problems in revenue management, portfolio optimization, and healthcare.

A related work by Bergman and Lozano [17] applies a similar approach to binary optimization problems with quadratic constraints. They show how a quadratic matrix can be decomposed into multiple decision diagrams, each of provably limited size. The BDDs are linked through channeling constraints to ensure that the solution represented is consistent across all diagrams. This approach can be readily embedded in general-purpose integer programming solvers, and provides significant computational improvements relative to state-of-the-art solvers. As a related work, González et al. [39] study the integration of BDDs into MIP solving for the quadratic stable set problem.

Another example is the extension of BDD-based cutting plane procedures for integer linear programming problems to integer *nonlinear* problems. For example, Davarnia and Van Hoeve [33] developed a cut generation method for optimization problems with integer variables, a linear objective, and nonlinear constraints. Their approach is a generalization of the well-known outer approximation framework, as the decision diagram-based cutting planes can also be derived for nonconvex integer programs. Another recent work is by Castro et al. [24] who develop a general cut-and-lift procedure based on decision diagrams, with a specific application to second-order conic inequalities.

Lastly, Davarnia [32] extends the application of decision diagrams to nonlinear optimization problems with *continuous* variables. Because decision diagrams rely on the decisions to be discrete and finite, continuous decisions pose a challenge. In (Davarnia [32]) this is tackled by first defining a Riemann approximation of the nonlinear functions, after which only the breakpoints are used to define the discrete 'decisions', yielding a so-called 'arc-reduced' decision diagram relaxation. This relaxation is then used to derive cutting planes that form a linear outer approximation for the original solution set. This procedure is also

FIGURE 8. (a) General integer program with $S \subset \mathbb{N}^n$, $M \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, (b) the column reformulation with $c_k \in \mathbb{R}$, $a_k \in \mathbb{R}^{m \times 1}$, $b \in \mathbb{R}^m$, and (c) a column reformulation form if S can be decomposed into n identical subsets.

a. General form	b. Column reformulation	c. Identical subsets form
$\min\ c(x)$	$\min\ \sum_k c_k \lambda_k$	$\min\ \sum_k c_k \lambda_k$
s.t. $Mx \geq b$	s.t. $\sum_k a_k \lambda_k \geq b$	s.t. $\sum_k a_k \lambda_k \geq b$
$x \in S$	$\sum_k \lambda_k = 1$	$\sum_k \lambda_k = n$
x integer	$\lambda_k \in \{0,1\}$	$\lambda_k \geq 0$ and integer

used in (Salemi and Davarnia [69]) to handle continuous variables to solve the unit commitment problem in the electric grid market.

Computational Performance To date, the most successful use of decision diagrams in general-purpose mixed-integer programming (MIP) solvers is the direct embedding of (relaxed) decision diagrams by reformulating the diagrams as a network flow. This is most effective when the diagram can represent a sub-structure of the problem that is otherwise challenging to handle with a linear programming formulation. Indeed, the work by Bergman and Cire [10] who embed nonlinear objective functions as BDD-based network flows into a MIP model showed strong computational benefits.

Tjandraatmadja and van Hoeve [74] propose a generic framework for applying decision diagrams in general-purpose MIP solvers to improve the dual and primal bound calculations. They use the *conflict graph* that is maintained by the MIP solver as the combinatorial structure from which to compile a relaxed BDD. Constraint propagation is used to refine the BDD, and a Lagrangian relaxation is embedded to strengthen the dual bound. The BDD is dynamically applied throughout the branch-and-bound search to help prune suboptimal nodes and find primal solutions. Their computational study shows that substantial speedups can be obtained when conflict graphs of sufficient importance are present.

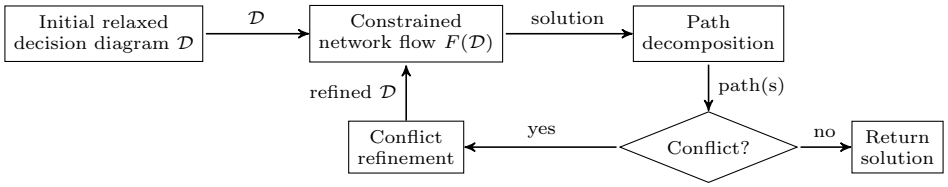
The use of decision diagrams for cut generation can be effective on specific problem domains, or when the problem possesses a combinatorial structure that is amenable to compiling BDDs of small size. A direct application on domain-agnostic MIP models, i.e., by compiling decision diagrams for a subset of the linear constraints, has not yet been shown computationally effective.

7. Column Elimination

In most of the applications so far, our goal was to find a *single discrete solution*, or r-t path, in the decision diagram, e.g., to find a dual or primal bound. This section discusses how we can leverage *multiple paths* simultaneously in a single diagram to solve a combinatorial optimization problem. This is done by representing the solutions as a network flow, and adding appropriate side constraints to ensure that the problem requirements are met. As we will see below, this approach is particularly effective when the original decision variables represent a combinatorial structure, e.g., a vehicle route or machine schedule. For these problems, column generation has been very successful. Decision diagrams allow taking an analogous approach to iteratively *remove* columns until an optimal solution is found.

Problem Representation Following the formalism for column generation (Barnhart et al. [6]), we consider integer programs of the general form depicted in Fig. 8.a. In this form, x is a vector of variables, and the objective function $c(x)$ can be linear or nonlinear. We assume that set S is bounded, and as a consequence that the set S^* consisting of all integer points in S is finite. Let $S^* = \{y_1, \ldots, y_p\}$. To obtain the column reformulation, define variable $\lambda_k \in \{0,1\}$ for each integer point $y_k \in S^*$. So, any point $y \in S^*$ can be written as $y = \sum_k y_k \lambda_k$. Requiring that $\sum_k \lambda_k = 1$ and defining $c_k = c(y_k)$ and $a_k = My_k$ gives the

FIGURE 9. Overview of the column elimination framework, taken from [48]. If the network flow problem $F(\mathcal{D})$ is solved as an integer program, the framework returns the optimal solution. If $F(\mathcal{D})$ is the continuous linear program, a dual bound is returned.



explicit linear reformulation of the general model in Fig. 8.b. Note that neither $c(x)$ nor the description of S needs to be linear.

Often S can be decomposed into different subsets for different components of x, i.e., $S = \cup_{j=1}^{n} S_j$ where $S_j^* = \{x_j \in S_j : x_j \text{ integer}\}$, yielding column reformulations of different forms. In particular, if the subsets in the decomposition are identical, i.e., $S_j^* = S^* = \{y_1, \ldots, y_p\}$ for $j = 1, \ldots, n$, we obtain the reformulation depicted in Fig. 8.c. In this case, a solution represents selecting n points from S^*, including multiplicity. It is also possible for the model to use a subset of columns of arbitrary size, as illustrated in the following example.

Example 12. Given a graph $G = (V, E)$ with vertex set V and edge set E, a *vertex coloring* is an assignment of a 'color' to each vertex such that no two adjacent vertices have the same color. The *graph coloring problem* asks to find a vertex coloring that uses the minimum number of colors (Garey and Johnson [34]).

We can represent this problem as an integer programming model using a column formulation (Mehrotra and Trick [60]). First, observe that for a given vertex coloring all vertices with the same color form an independent set (see Sec. 2). Conversely, each independent set of G can be used as a *color class* by labeling each vertex in the set the same color. We let S be the set of all independent sets of G, and define a binary decision variable x_j for each $j \in S$. We define $a_{ij} = 1$ if vertex $i \in V$ belongs to independent set $j \in S$. The graph coloring problem can then be formulated as:

$$\min \sum_{j \in S} x_j \tag{21}$$

$$\text{s.t.} \sum_{j \in S} a_{ij} x_j = 1 \qquad \forall i \in V, \tag{22}$$

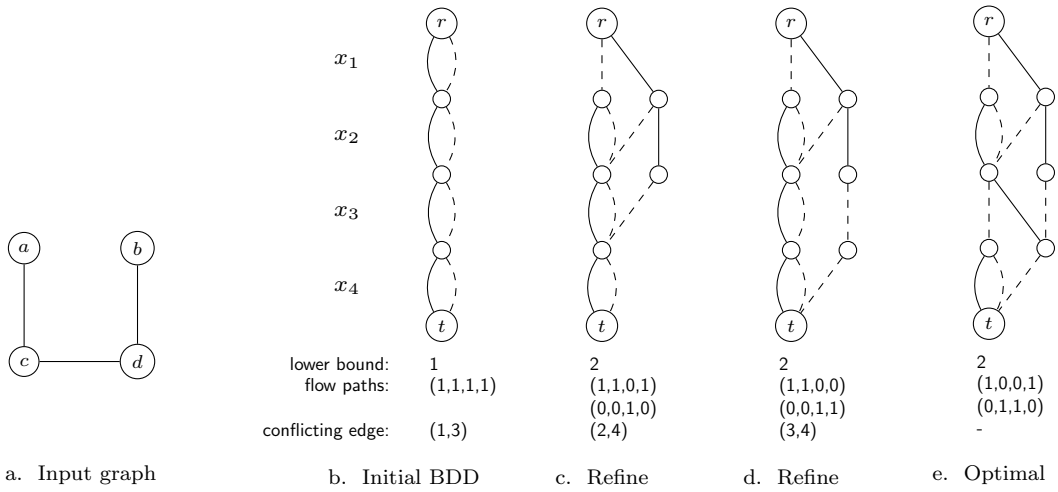
$$x_j \in \{0, 1\} \qquad \forall j \in S. \tag{23}$$

Indeed, this column formulation does not impose a fixed number of points in S to be used.

Column Elimination Algorithm Column formulations have been successfully applied to a variety of industrial applications, including cutting stock production (where a column is a cutting pattern), vehicle routing (where a column is a route), airline crew scheduling (where a column is an employee schedule), and many more. Because the number of columns can generally grow exponentially large in the original problem size, column generation methods solve the linear programming relaxation iteratively, generating and adding columns with negative reduced cost until the linear programming solution is optimal. Throughout this process, column generation maintains a restricted master problem containing a subset of the columns.

In contrast, column *elimination* maintains a *relaxed* master problem that contains all original columns as well columns that are infeasible (van Hoeve [77, 78]). While this may seem counter-intuitive, the relaxation can be represented compactly as a decision diagram which allows solving a polynomial-sized master problem that implicitly represents an exponential number of columns. The column elimination algorithm proceeds as follows (see Fig. 9):

FIGURE 10. Applying column elimination to solve the graph coloring problem on the input graph in (a). The initial relaxed diagram (b) is iteratively refined (c)-(e) until the optimal solution (the flow paths) no longer contains infeasible color classes (e). Images taken from [78].



a. Input graph b. Initial BDD c. Refine d. Refine e. Optimal

(1) An initial relaxed decision diagram $\mathcal{D}$ is compiled such that $\text{Sol}(\mathcal{D}) \supseteq S$.
(2) A master (integer) linear problem is formulated as a minimum-cost network flow over $\mathcal{D}$. Any side constraints are expressed over the arc flow variables in $\mathcal{D}$.
(3) The optimal solution is decomposed into r-t paths, each corresponding to a column. Each path is checked for feasibility; if all paths are feasible, we return the optimal solution.
(4) Otherwise, each infeasible path is eliminated from the diagram by removing the associated partial paths. This usually requires splitting nodes; at most one node needs to be created per layer to remove an infeasible path. We continue the process on the refined diagram until no more paths in the solution are infeasible.

For a given decision diagram $\mathcal{D}$, the network flow formulation is similar to the network flow model in Section 6; in this case we allow the flow value to be larger than 1. That is, the master problem for decision diagram $\mathcal{D} = (N, A)$ with node set N, arc set A, arc weights w_a and arc decision labels $\ell(a)$ for $a \in A$, takes the general form analogous to the column generation models in Fig 8:

$$\min \sum_{a \in A} w_a y_a \tag{24}$$

$$\sum_{a \in \delta^-(v)} y_a - \sum_{a \in \delta^+(v)} y_a = 0 \qquad \forall v \in N \setminus \{r, t\}, \tag{25}$$

$$M'y \geq b, \tag{26}$$

$$y_a \geq 0 \qquad \forall a \in A. \tag{27}$$

Here we assume that constraint set $M'y \geq b$ is a reformulation of the original constraint set $Mx \geq b$ using the mapping from the flow variables y to the original x variables, as in equation (15). Generally, constraints (26) do not retain the integrality property of the network flow solution. We can either solve the model as a linear program to obtain a dual bound as solution, or impose integrality constraints on the flow variables y to obtain the integer optimal solution.

Example 13. We apply the column elimination algorithm to the graph coloring problem in Example 12, using the input graph depicted in Fig. 10.a as illustration. Recall that the set S represents all independent sets of graph G. We have seen in Example 4 how the exact

diagram representing all independent sets can be compiled in a top-down manner. Here, we instead apply compilation by separation, starting with the trivial relaxed diagram $\mathcal{D}$ of width 1 (see Fig. 10.b). The constrained network flow model for this problem is:

$$\min \sum_{(r,v)\in A} y_{(r,v)} \tag{28}$$

$$\text{s.t.} \sum_{a\in\delta^-(v)} y_a - \sum_{a\in\delta^+(v)} y_a = 0 \qquad \forall v \in N \setminus \{r,t\}, \tag{29}$$

$$\sum_{\substack{(u,v)\in A: L(u)=i,\\ \ell(u,v)=1}} y_{(u,v)} = 1 \qquad \forall i \in V, \tag{30}$$

$$y_a \in \{0,1,\ldots,|V|\} \qquad \forall a \in A. \tag{31}$$

Observe that this model minimizes the total flow out of the root, while ensuring that each vertex is part of exactly one path (constraints (30)). Also, this model imposes integrality constraints on y.

After solving the initial network flow model, we apply a path decomposition (see Fig. 10.b) and identify that the all-ones solution has a conflict: edge $(1,3)$ along the partial path $(x_1,x_2,x_3)=(1,1,1)$. Fig. 10.c shows the refined diagram that no longer contains this partial path. We resolve the network flow model and decompose its solution into paths (Fig. 10.c). The dual bound increases from 1 to 2, but we identify a conflict (edge $(2,4)$). Fig. 10.c separates the associated partial path, and we continue the process one more time (refining conflict $(3,4)$ after which the resulting path decomposition contains no more conflicts (Fig. 10.d) and the optimal solution is returned: color classes $\{a,d\}$ and $\{c,b\}$.

Column elimination was first introduced as an alternative to branch-and-price to solve the vertex coloring problem in (van Hoeve [78]). A follow-up work studies the impact of the variable ordering in this context (Karahalios and van Hoeve [49]). Column elimination was subsequently applied to solve a truck-drone routing problem (Tang and van Hoeve [72]), which includes a subgradient descent method for solving a Lagrangian reformulation of the model. Most recently, column elimination was used to solve a capacitated vehicle routing problem (Karahalios and van Hoeve [48]). The latter work includes the addition of cutting planes, reduced cost based variable fixing, and a more general and improved subgradient descent method. A related work, using a similar network flow formulation but on a static decision diagram, was developed for parallel machine scheduling problems (Kowalczyk et al. [54]).

Computational Performance Column elimination has several potential computational advantages with respect to column generation, its close cousin. First, as it works with a relaxed master problem instead of a restriction, it can be terminated at any time and return a dual bound. Second, the flow model can be solved as an integer program, avoiding the need for branch-and-price. Third, because column elimination has no pricing problem it avoids computational issues related to dual degeneracy which is often a challenge in column generation. Fourth, its architecture is relatively simple: the decision diagram is a layered directed graph, the flow models are easily defined and solved, and the refinement process is well understood. In contrast, branch-and-price implementations often involve delicate algorithms for solving the pricing problem efficiently.

A drawback of column elimination, again relative to column generation, is that the network flow model has many more arc variables than the typical column formulation that has one variable per column. It therefore can take more time to solve the (integer) linear program for each iteration. Furthermore, it may take more refinement iterations for column elimination to arrive at an optimal solution than it takes pricing iterations for column generation. Lastly, the most successful column generation approaches, e.g., for vehicle routing, require only a

few branch-and-price search tree nodes due to addition of strong cuts at the root node. The addition of similarly strong cuts to column elimination has been studied (Karahalios and van Hoeve [48]) but requires more development before it reaches similar performance.

Overall, column elimination is a promising new approach that has been shown to be competitive with the state of the art on graph coloring (van Hoeve [78], Karahalios and van Hoeve [49]), truck-drone routing (Tang and van Hoeve [72]), and capacitated vehicle routing (Karahalios and van Hoeve [48]). Specific opportunities for further development include state-based cut generation and the application to non-linear problem structures.

8. Summary

This tutorial introduced the use of decision diagrams in the context of discrete optimization. We have seen that decision diagrams can compactly represent an exponential set of solutions, and that optimizing over a decision diagram can be done in polynomial time in the size of the diagram. However, decision diagrams can grow exponentially large, which prevents their general application to be computationally practical. This tutorial discussed three ways in which decision diagrams have been successfully applied to optimization problems:

- The first is using exact diagrams, either to represent the entire problem (when it fits in memory), or a sub-structure of the problem. For example, in constraint programming we may choose an exact decision diagram to represent one constraint to improve propagation. In integer programming, we may choose to select a subset of constraints, or the objective function, to be represented using one or more exact decision diagrams. The decision diagrams can then be embedded as linear network flow reformulations into the broader integer programming model.
- The second is using relaxed and/or restricted diagrams of polynomial size. The diagrams can be built using the top-down compilation method or via compilation by separation, up to a maximum size limit. One option is to use the relaxed and restricted diagrams to obtain dual and primal bounds and embed them in a branch-and-bound search, as a stand-alone solver. Another option is to embed relaxed decision diagrams into constraint programming solvers to improve constraint propagation, or into integer programming solvers to generate cuts and improve dual or primal bounds. Various applications of this approach have shown state-of-the art computational performance on a variety of academic benchmarks. They have also been applied in industrial settings in the context of vehicle routing.
- The third is using relaxed decision diagrams in a process called column elimination, in which a solution is computed as a network flow over the diagram, possibly consisting of multiple paths. Rather than compiling the relaxed diagram in a top-down fashion, it is iteratively refined by removing infeasible paths until an optimal feasible solution is obtained. This approach has been applied to graph coloring and vehicle routing applications, showing competitive computational results.

The tutorial has focused on presenting the main underlying concepts of decision diagrams and their use in integer optimization solvers. Many more applications of decision diagrams have recently appeared in the broader optimization literature, including stochastic optimization (Lozano and Smith [58], Salemi and Davarnia [70], MacNeil and Bodur [59]), Benders decomposition (Guo et al. [40]), pricing in column generation (Morrison et al. [61], Kowalczyk and Leus [53]), post-optimality analysis (Hadzic and Hooker [41], Serra and Hooker [71]), domain-specific applications (Cire et al. [28], Hosseininasab and van Hoeve [46]), and many more (Castro et al. [25]). For each of these, the goal is to leverage decision diagrams to capture a (combinatorial) structure that would be challenging to represent using existing methods.

References

[1] Ahuja RK, Magnanti TL, Orlin JB (1993) *Network Flows: Theory, Algorithms, and Applications* (Prentice-Hall).

[2] Akers SB (1978) Binary decision diagrams. *IEEE Transactions on Computers* C-27:509–516.

[3] Andersen HR, Hadzic T, Hooker JN, Tiedemann P (2007) A Constraint Store Based on Multivalued Decision Diagrams. Bessiere C, ed., *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, 118–132 (Springer).

[4] Apt KR (2003) *Principles of Constraint Programming* (Cambridge University Press).

[5] Baptiste P, Pape CL, Nuijten W (2001) *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems* (Dordrecht: Kluwer).

[6] Barnhart C, Johnson EL, Nemhauser GL, Savelsbergh MWP, Vance PH (1998) Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Oper. Res.* 46(3):316–329.

[7] Becker B, Behle M, Eisenbrand F, Wimmer R (2005) BDDs in a branch and cut framework. Nikoletseas S, ed., *Experimental and Efficient Algorithms, Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA 05)*, volume 3503 of *Lecture Notes in Computer Science*, 452–463 (Springer).

[8] Behle M (2007) *Binary Decision Diagrams and Integer Programming*. Ph.D. thesis, Universität des Saarlandes.

[9] Bellman R (1957) *Dynamic Programming* (Princeton University Press).

[10] Bergman D, Ciré AA (2018) Discrete Nonlinear Optimization by State-Space Decompositions. *Manag. Sci.* 64(10):4700–4720.

[11] Bergman D, Ciré AA, Sabharwal A, Samulowitz H, Saraswat VA, van Hoeve WJ (2014) Parallel Combinatorial Optimization with Decision Diagrams. Simonis H, ed., *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, volume 8451 of *Lecture Notes in Computer Science*, 351–367 (Springer).

[12] Bergman D, Ciré AA, van Hoeve W (2015) Lagrangian bounds from decision diagrams. *Constraints An Int. J.* 20(3):346–361.

[13] Bergman D, Cire AA, Van Hoeve WJ, Hooker J (2016) *Decision Diagrams for Optimization* (Springer).

[14] Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2013) Optimization bounds from binary decision diagrams. *INFORMS J. Comp.* 26(2):253–268.

[15] Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2016) Discrete Optimization with Decision Diagrams. *INFORMS J. Comp.* 28(1):47–66.

[16] Bergman D, Ciré AA, van Hoeve WJ, Yunes TH (2014) BDD-based heuristics for binary optimization. *J. Heuristics* 20(2):211–234.

[17] Bergman D, Lozano L (2021) Decision Diagram Decomposition for Quadratically Constrained Binary Optimization. *INFORMS J. Comput.* 33(1):401–418.

[18] Bergman D, van Hoeve WJ, Hooker JN (2011) Manipulating MDD Relaxations for Combinatorial Optimization. Achterberg T, Beck JC, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings*, volume 6697 of *Lecture Notes in Computer Science*, 20–35 (Springer).

[19] Boole G (1854) An Investigation of the Laws of Thought: On which are Founded the Mathematical Theories of Logic and Probabilities.

[20] Bryant RE (1986) Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* C-35:677–691.

[21] Bryant RE (1992) Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys* 24:293– 318.

[22] Bryant RE (2018) Binary Decision Diagrams. Clarke EM, Henzinger TA, Veith H, Bloem R, eds., *Handbook of Model Checking*, 191–217 (Springer International Publishing).

[23] Castro MP, Ciré AA, Beck JC (2020) An MDD-Based Lagrangian Approach to the Multicommodity Pickup-and-Delivery TSP. *INFORMS J. Comput.* 32(2):263–278.

[24] Castro MP, Ciré AA, Beck JC (2022) A combinatorial cut-and-lift procedure with an application to 0-1 second-order conic programming. *Math. Program.* 196(1):115–171.

[25] Castro MP, Ciré AA, Beck JC (2022) Decision Diagrams for Discrete Optimization: A Survey of Recent Advances. *INFORMS J. Comput.* 34(4):2271–2295.

[26] Cheng KCK, Yap RHC (2005) Constrained Decision Diagrams. Veloso MM, Kambhampati S, eds., *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 366–371 (AAAI Press / The MIT Press).

[27] Cheng KCK, Yap RHC (2008) Maintaining Generalized Arc Consistency on Ad Hoc r-Ary Constraints. Stuckey PJ, ed., *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, volume 5202 of *Lecture Notes in Computer Science*, 509–523 (Springer).

[28] Cire AA, Diamant A, T Y, Carrasco A (2019) A Network-Based Formulation for Scheduling Clinical Rotations. *Production and Operations Management* 28(5):1186–1205.

[29] Ciré AA, van Hoeve WJ (2013) Multivalued Decision Diagrams for Sequencing Problems. *Oper. Res.* 61(6):1411–1428.

[30] Conforti M, Cornuéjols G, Zambelli G (2014) *Integer Programming* (Springer).

[31] Coppé V, Gillard X, Schaus P (2022) Solving the Constrained Single-Row Facility Layout Problem with Decision Diagrams. Solnon C, ed., *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, volume 235 of *LIPIcs*, 14:1–14:18 (Schloss Dagstuhl - Leibniz-Zentrum für Informatik).

[32] Davarnia D (2021) Strong relaxations for continuous nonlinear programs based on decision diagrams. *Oper. Res. Lett.* 49(2):239–245.

[33] Davarnia D, van Hoeve W (2021) Outer approximation for integer nonlinear programs via decision diagrams. *Math. Program.* 187(1):111–150.

[34] Garey M, Johnson D (1979) *Computers and Intractability - A Guide to the Theory of NP-Completeness* (Freeman).

[35] Gentzel R, Michel L, van Hoeve W (2023) Optimization Bounds from Decision Diagrams in Haddock. Ciré AA, ed., *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 20th International Conference, CPAIOR 2023, Nice, France, May 29 - June 1, 2023, Proceedings*, volume 13884 of *Lecture Notes in Computer Science*, 150–166 (Springer).

[36] Gentzel R, Michel L, van Hoeve WJ (2020) HADDOCK: A Language and Architecture for Decision Diagram Compilation. Simonis H, ed., *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, 531–547 (Springer).

[37] Gillard X, Coppé V, Schaus P, Ciré AA (2021) Improving the Filtering of Branch-and-Bound MDD Solver. Stuckey PJ, ed., *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*, 231–247 (Springer).

[38] Gillard X, Schaus P, Coppé V (2020) Ddo, a Generic and Efficient Framework for MDD-Based Optimization. Bessiere C, ed., *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 5243–5245 (ijcai.org).

[39] González JE, Ciré AA, Lodi A, Rousseau L (2022) Bdd-based optimization for the quadratic stable set problem. *Discret. Optim.* 44(Part 2):100610.

[40] Guo C, Bodur M, Aleman DM, Urbach DR (2021) Logic-Based Benders Decomposition and Binary Decision Diagram Based Approaches for Stochastic Distributed Operating Room Scheduling. *INFORMS J. Comput.* 33(4):1551–1569.

[41] Hadzic T, Hooker JN (2007) Cost-Bounded Binary Decision Diagrams for 0-1 Programming. Hentenryck PV, Wolsey LA, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 4th International Conference, CPAIOR 2007, Brussels, Belgium, May 23-26, 2007, Proceedings*, volume 4510 of *Lecture Notes in Computer Science*, 84–98 (Springer).

[42] Hadzic T, Hooker JN, O'Sullivan B, Tiedemann P (2008) Approximate Compilation of Constraints into Multivalued Decision Diagrams. Stuckey PJ, ed., *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, volume 5202 of *Lecture Notes in Computer Science*, 448–462 (Springer).

[43] Hawkins P, Lagoon V, Stuckey PJ (2005) Solving Set Constraint Satisfaction Problems using ROBDDs. *J. Artif. Intell. Res.* 24:109–156.

[44] Hoda S, van Hoeve WJ, Hooker JN (2010) A Systematic Approach to MDD-Based Constraint Programming. Cohen D, ed., *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, 266–280 (Springer).

[45] Hooker JN (2019) Improved Job Sequencing Bounds from Decision Diagrams. Schiex T, de Givry S, eds., *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, 268–283 (Springer).

[46] Hosseininasab A, van Hoeve W (2021) Exact Multiple Sequence Alignment by Synchronized Decision Diagrams. *INFORMS J. Comput.* 33(2):721–738.

[47] Jung V, Régin J (2022) Efficient Operations Between MDDs and Constraints. Schaus P, ed., *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*, volume 13292 of *Lecture Notes in Computer Science*, 173–189 (Springer).

[48] Karahalios A, van Hoeve W (2023) Column Elimination for Capacitated Vehicle Routing Problems. Ciré AA, ed., *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 20th International Conference, CPAIOR 2023, Nice, France, May 29 - June 1, 2023, Proceedings*, volume 13884 of *Lecture Notes in Computer Science*, 35–51 (Springer).

[49] Karahalios A, van Hoeve WJ (2022) Variable ordering for decision diagrams: A portfolio approach. *Constraints* 27(1):116–133.

[50] Keller RM (1976) Formal Verification of Parallel Programs. *Communications of the ACM* 19(7):371–384.

[51] Kinable J, Ciré AA, van Hoeve W (2017) Hybrid optimization methods for time-dependent sequencing problems. *Eur. J. Oper. Res.* 259(3):887–897.

[52] Knuth DE (2009) *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams* (Addison-Wesley Professional).

[53] Kowalczyk D, Leus R (2018) A Branch-and-Price Algorithm for Parallel Machine Scheduling Using ZDDs and Generic Branching. *INFORMS J. Comput.* 30(4):768–782.

[54] Kowalczyk D, Leus R, Hojny C, Røpke S (to appear) A Flow-Based Formulation for Parallel Machine Scheduling Using Decision Diagrams. *INFORMS J. Comp.* .

[55] Lai YT, Pedram M, Vrudhula S (1994) EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 959–975.

[56] Land AH, Doig AG (1960) An Automatic Method of Solving Discrete Programming Problems. *Econometrica* 28(3):497–520.

[57] Lee CY (1959) Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal* 38:985–999.

[58] Lozano L, Smith JC (2022) A binary decision diagram based algorithm for solving a class of binary two-stage stochastic programs. *Math. Program.* 191(1):381–404.

[59] MacNeil M, Bodur M (2024) Leveraging decision diagrams to solve two-stage stochastic programs with binary recourse and logical linking constraints. *Eur. J. Oper. Res.* 315(1):228–241.

[60] Mehrotra A, Trick MA (1996) A Column Generation Approach for Graph Coloring. *INFORMS J. Comp.* 8(4):344–354.

[61] Morrison DR, Sewell EC, Jacobson SH (2016) Solving the Pricing Problem in a Branch-and-Price Algorithm for Graph Coloring Using Zero-Suppressed Binary Decision Diagrams. *INFORMS J. Comput.* 28(1):67–82.

[62] O'Neil RJ, Hoffman K (2019) Decision diagrams for solving traveling salesman problems with pickup and delivery in real time. *Oper. Res. Lett.* 47(3):197–201.

[63] Perez G, Régin J (2015) Efficient Operations On MDDs for Building Constraint Programming Models. Yang Q, Wooldridge MJ, eds., *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 374–380 (AAAI Press).

[64] Régin J (1994) A Filtering Algorithm for Constraints of Difference in CSPs. *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, 362–367 (AAAI Press / The MIT Press).

[65] Régin JC (2011) Global Constraints: A Survey. van Hentenryck P, Milano M, eds., *Hybrid Optimization: The Ten Years of CPAIOR*, 63–134 (Springer New York).

[66] Rossi F, van Beek P, Walsh T, eds. (2006) *Handbook of Constraint Programming* (Elsevier).

[67] Rudich I, Cappart Q, Rousseau L (2022) Peel-And-Bound: Generating Stronger Relaxed Bounds with Multivalued Decision Diagrams. Solnon C, ed., *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, volume 235 of *LIPIcs*, 35:1–35:20 (Schloss Dagstuhl - Leibniz-Zentrum für Informatik).

[68] Rudich I, Cappart Q, Rousseau L (2023) Improved Peel-and-Bound: Methods for Generating Dual Bounds with Multivalued Decision Diagrams. *J. Artif. Intell. Res.* 77:1489–1538.

[69] Salemi H, Davarnia D (2023) On the Structure of Decision Diagram–Representable Mixed-Integer Programs with Application to Unit Commitment. *Oper. Res.* 71(6):1943–1959.

[70] Salemi H, Davarnia D (2023) Solving Unsplittable Network Flow Problems with Decision Diagrams. *Transp. Sci.* 57(4):937–953.

[71] Serra T, Hooker JN (2020) Compact representation of near-optimal integer programming solutions. *Math. Program.* 182(1):199–232.

[72] Tang Z, van Hoeve W (2024) Dual Bounds from Decision Diagram-Based Route Relaxations: An Application to Truck-Drone Routing. *Transp. Sci.* 58(1):257–278.

[73] Tjandraatmadja C, van Hoeve W (2019) Target Cuts from Relaxed Decision Diagrams. *INFORMS J. Comput.* 31(2):285–301.

[74] Tjandraatmadja C, van Hoeve W (2021) Incorporating bounds from decision diagrams into integer programming. *Math. Program. Comput.* 13(2):225–256.

[75] van den Bogaerdt P, de Weerdt M (2018) Multi-machine scheduling lower bounds using decision diagrams. *Oper. Res. Lett.* 46(6):616–621.

[76] van den Bogaerdt P, de Weerdt M (2019) Lower Bounds for Uniform Machine Scheduling Using Decision Diagrams. Rousseau L, Stergiou K, eds., *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*, 565–580 (Springer).

[77] van Hoeve W (2020) Graph Coloring Lower Bounds from Decision Diagrams. Bienstock D, Zambelli G, eds., *Integer Programming and Combinatorial Optimization - 21st International Conference, IPCO 2020, London, UK, June 8-10, 2020, Proceedings*, volume 12125 of *Lecture Notes in Computer Science*, 405–418 (Springer).

[78] van Hoeve W (2022) Graph coloring with decision diagrams. *Math. Program.* 192(1):631–674.

[79] van Hoeve WJ, Katriel I (2006) Global Constraints. Rossi P F van Beek, Walsh T, eds., *Handbook of Constraint Programming*, chapter 6 (Elsevier).

[80] Verhaeghe H, Lecoutre C, Schaus P (2018) Compact-MDD: Efficiently Filtering (s)MDD Constraints with Reversible Sparse Bit-sets. Lang J, ed., *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, 1383–1389 (ijcai.org).

[81] Wegener I (2000) *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM monographs on discrete mathematics and applications (Society for Industrial and Applied Mathematics), ISBN 9780898714586.