Optimization Bounds from Decision Diagrams in Haddock

Rebecca Gentzel[1], Laurent Michel[1](Synchrony Chair in Cybersecurity)[0000−0001−7230−7130], and Willem-Jan van Hoeve[2][0000−0002−0023−753X]

[1] University of Connecticut, Storrs CT 06269, USA
{rebecca.gentzel,laurent.michel}@uconn.edu
[2] Carnegie Mellon University, Pittsburgh PA 15213, USA
vanhoeve@andrew.cmu.edu

Abstract. We study the automatic generation of primal and dual bounds from decision diagrams in constraint programming. In particular, we expand the functionality of the Haddock system to optimization problems by extending its specification language to include an objective function. We describe how restricted decision diagrams can be compiled in Haddock similar to the existing relaxed decision diagrams. Together, they provide primal and dual bounds on the objective function, which can be seamlessly integrated into the constraint programming search. The entire process is automatic and only requires a high-level user model specification. We evaluate our method on the sequential ordering problem and compare the performance of Haddock to a dedicated decision diagram approach. The results show that Haddock achieves comparable results in similar time, demonstrating the viability of our automated decision diagram procedures for constraint optimization problems.

Keywords: Decision Diagrams · Constraint Programming Systems · Optimization Bounds

1 Introduction

Constraint Programming (CP) traditionally focuses on feasibility solving for Constraint Satisfaction Problems (CSP). Namely, it focuses on finding either one or all feasible solutions to a CSP $\langle X, D, C\rangle$. The ability to tackle *optimization* problems is added by solving a sequence of CSPs, each one with an additional constraint that requires the production of a solution that improves upon the last incumbent solution. Naturally, the last problem in the sequence is infeasible and the entire search tree itself is the optimality certificate. While search strategies to explore this search tree vary, the most common choice is a depth first search on a search tree dynamically-defined with variable and value selection heuristics. Black-box searches [14,13,4,9] provide pre-defined variable and value selection heuristics and non-sequential strategies such as limited discrepancy search offer compositional solutions to consider alternative strategies that remain orthogonal to the objective function.

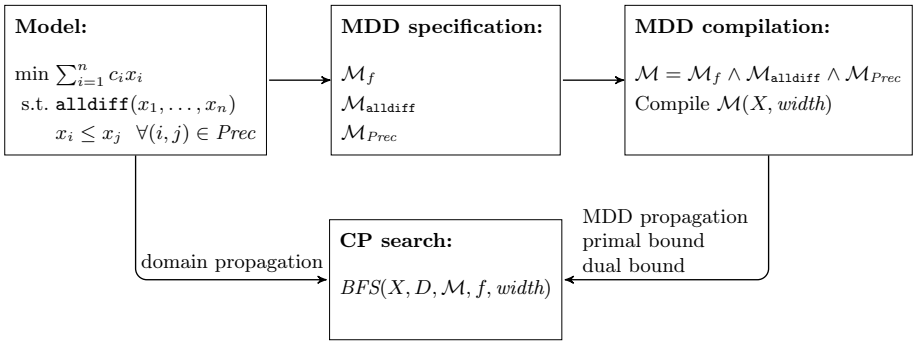

Fig. 1: Overview of automatic MDD-based constraint programming in HADDOCK on an example constraint optimization problem $\langle X, D, C, f \rangle$ with variables X, domains D, constraints C and objective function f. The constraint programming search employs a branch-and-bound best-first strategy (BFS). The MDD specification and compilation are derived automatically from the model declaration.

Mixed Integer Programming (MIP) solvers employ a different strategy. They rely on a *linear relaxation* of the MIP model that removes integrality constraints and leverage a linear programming solver to obtain a dual bound. MIP solvers then use branch-and-bound style techniques to organize and explore the frontier of nodes to be expanded. Like CP solvers, MIP solvers primarily rely on the search process to produce a sequence of improving incumbents (thus, tightening the primal bound), though techniques such as probing or the feasibility pump [5] offer additional mechanisms to tighten the primal bound. The dual bounds produced at each node by the relaxation give a mechanism to prioritize nodes in the frontier and explore the most promising options first.

Multi-valued decision diagrams (MDDs) were recently introduced as an effective tool to derive optimization bounds for discrete optimization problems, and embed these in a branch-and-bound search [2]. This paper explores the use of MDDs as a systematic mechanism to leverage both primal and dual bounds *within a CP solver* to enable MIP-style branch-and-bound search within the confines of a CP solver. HADDOCK was introduced as a generic architecture and language for MDD propagation in a CP framework [6]. To this end, we extend the existing HADDOCK framework to allow its use in optimization problems and derive both classes of bounds for problems that are expressible as an MDD in the HADDOCK language. A schematic overview is depicted in Figure 1 on an example constraint optimization problem (COP), that has a weighted sum as objective function, an `alldiff` constraint, and precedence constraints (defined on a set *Prec*). In addition to automatically deriving an MDD specification for each constraint [6], we now also derive an MDD specification for the objective function. The specification language is compositional, which means that HADDOCK can take the conjunction of all MDD specifications to compile a single MDD. By adding the objective specification, the MDD can now be automatically used to derive primal and dual bounds during the CP search. For the primal bound, we assume that all constraints are represented in the MDD.

Contributions. Our main contributions are 1) a formal MDD specification for objective functions in CP systems, 2) a procedure to compile a restricted decision diagram using the MDD specification, 3) integrating the primal and dual bounds from the MDD into a CP search, and 4) demonstrating the abilities of the framework on the sequential ordering problem as a concrete application. The empirical results show that Haddock offers comparable performance relative to a dedicated implementation of an MDD-based branch-and-bound search method.

Section 2 gives an overview of the formalization used in Haddock. Section 3 provides the necessary additions to allow Haddock to communicate with an objective variable. Section 4 covers the use of the language for building restricted MDDs to obtain primal bounds. Section 5 describes how to do a best-first search using Haddock. Finally, Section 6 reports on the empirical results, and Section 7 concludes the paper.

2 Background

2.1 MDD as Layered Transition System

Following [6], we formally define an MDD as a labeled transition system [11]:

Definition 1. *A* labeled transition system *is a triplet* $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ *where* $\mathcal{S}$ *is a set of states,* $\rightarrow$ *is a relation of labeled transitions between states from* $\mathcal{S}$*, and* Λ *is a set of labels used to tag transitions.*

Definition 2. *Given an ordered set of variables* $X = \{x_1, \ldots, x_n\}$ *with domains* $D(x_1)$ *through* $D(x_n)$*, a* multi-valued decision diagram (MDD) *on* X *is a layered transition system* $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ *in which:*

- *the state set* $\mathcal{S}$ *is stratified in* $n+1$ *layers* $\mathcal{L}_0$ *through* $\mathcal{L}_n$ *with transitions from* $\rightarrow$ *connecting states between layers* i *and* $i+1$ *exclusively;*
- *the transition label set* Λ *is defined as* $\bigcup_{i \in 1..n} D(x_i)$*;*
- *a transition between two states* $a \in \mathcal{L}_{i-1}$ *and* $b \in \mathcal{L}_i$ *carries a label* $v \in D(x_i)$ *(*$i \in 1..n$*);*
- *the layer* $\mathcal{L}_0$ *consists of a single* source *state* $s_\perp$*;*
- *the layer* $\mathcal{L}_n$ *consists of a single* sink *state* $s_\top$*.*

An MDD M can represent a constraint set with specific state definitions and transition functions. If each solution in the constraint set is represented by an $s_\perp$-$s_\top$ path in M, and vice-versa, M is *exact*. If M represents a superset of the solutions of the constraint set, it is *relaxed*. If M represents a subset of the solutions of the constraint set, it is *restricted*. In Haddock, states consist of integer-valued sets of *properties* to represent the constraints. We next describe how these are used to automatically compile the LTS, using the `among` constraint as an illustration. For a complete description, we refer to [6].

2.2 State Properties

Recall the definition of the `among` global constraint on an ordered set X of n variables [1]. It counts the number of occurrences of values taken from a given set Σ and ensures that the total number is between l and u, i.e.,

$$\texttt{among}(X, l, u, \Sigma) := l \leq \sum_{i=1}^{n} (x_i \in \Sigma) \leq u.$$

A state for $\texttt{among}(X, l, u, \Sigma)$ carries four properties, i.e., $\langle L^{\downarrow}, U^{\downarrow}, L^{\uparrow}, U^{\uparrow} \rangle$, for each node v in the MDD:

- $L^{\downarrow} \in \mathbb{Z}$: minimum number of times a value in Σ is taken from $s_{\bot}$ to v.
- $U^{\downarrow} \in \mathbb{Z}$: maximum number of times a value in Σ is taken from $s_{\bot}$ to v.
- $L^{\uparrow} \in \mathbb{Z}$: minimum number of times a value in Σ is taken from v to $s_{\top}$.
- $U^{\uparrow} \in \mathbb{Z}$: maximum number of times a value in Σ is taken from v to $s_{\top}$.

We initialize the state for the source $s_{\bot}$ as $\langle 0, 0, -, - \rangle$ and the sink $s_{\top}$ as $\langle -, -, 0, 0 \rangle$.

2.3 Transition Functions

The transition between a node $a \in \mathcal{L}_{i-1}$ and $b \in \mathcal{L}_i$ is an arc (a, b) labeled by a value $\ell \in D(x_i)$. We use *transition functions* $T^{\downarrow}(a, b, i, \ell)$ and $T^{\uparrow}(b, a, i, \ell)$ to derive the property values (the states) for b and a, respectively. For each individual property p, we use the function $f(s, p, \ell)$ for a given state s. For `among`, we apply $f(s, p, \ell) = p(s) + (\ell \in \Sigma)$ for each property p in $\langle L^{\downarrow}, U^{\downarrow}, L^{\uparrow}, U^{\uparrow} \rangle$. For example, we define $L^{\downarrow}(b) = f(a, L^{\downarrow}, \ell)$, i.e., $L^{\downarrow}(a) + (\ell \in \Sigma)$. We likewise define $L^{\uparrow}(a) = f(b, L^{\uparrow}, \ell)$, $U^{\downarrow}(b) = f(a, U^{\downarrow}, \ell)$ and $U^{\uparrow}(a) = f(b, U^{\uparrow}, \ell)$. The state-level transition functions $T^{\downarrow}$ and $T^{\uparrow}$ compute all the down or up properties of the next state as follows:

$$T^{\downarrow}(a, b, i, \ell) = \langle f(a, L^{\downarrow}, \ell), f(a, U^{\downarrow}, \ell), -, - \rangle$$
$$T^{\uparrow}(b, a, i, \ell) = \langle -, -, f(b, L^{\uparrow}, \ell), f(b, U^{\uparrow}, \ell) \rangle.$$

Note that slight variants of both functions that preserve the properties of states b and a, respectively, in the opposite directions are equally helpful. Those are:

$$T^{\downarrow}(a, b, i, \ell) = \langle f(a, L^{\downarrow}, \ell), f(a, U^{\downarrow}, \ell), L^{\uparrow}(b), U^{\uparrow}(b) \rangle$$
$$T^{\uparrow}(b, a, i, \ell) = \langle L^{\downarrow}(a), U^{\downarrow}(a), f(b, L^{\uparrow}, \ell), f(b, U^{\uparrow}, \ell) \rangle.$$

2.4 Transition Existence Function

The *transition existence function* $E_t(a, b, i, \ell)$ specifies whether an arc (a, b) with label $\ell \in D(x_i)$ exists in the LTS. For `among`, this function should ensure that the lower bound l is met and the upper bound u is not exceeded, i.e.:

$$U^{\downarrow}(a) + (\ell \in S) + U^{\uparrow}(b) \geq l \wedge L^{\downarrow}(a) + (\ell \in S) + L^{\uparrow}(b) \leq u.$$

2.5 Node Relaxation Functions

Two states a and b in the same layer $\mathcal{L}_i$ can be relaxed (merged) to produce a new state s' according to a *relaxation function* $R(a, b)$. For `among`, we can use:

$$R(a,b) = \langle\ \min\{L^{\downarrow}(a), L^{\downarrow}(b)\}, \max\{U^{\downarrow}(a), U^{\downarrow}(b)\}, \\ \min\{L^{\uparrow}(a), L^{\uparrow}(b)\}, \max\{U^{\uparrow}(a), U^{\uparrow}(b)\}\ \ \rangle.$$

State relaxation generalizes to an ordered set of states $\{s_0, s_1, \ldots, s_{k-1}\}$ as follows:

$$R(s_0, R(s_1, R(\ldots, R(s_{k-2}, s_{k-1}) \ldots))).$$

For `among`, we maintain MDD-bounds consistency on this expression, i.e., we only maintain a lower and upper bound on the count to ensure feasibility and rely on the above relaxation function to merge nodes and bound the width of the MDD to at most w states. The usage of a relaxation is precisely why we maintain bounds (L and U) in both up and down directions. Note that full MDD consistency for `among` can be established in polynomial time by maintaining a set of exact counts [10].

2.6 MDD Language

All of the above are used to define an *MDD language* used to generate an MDD for propagation:

Definition 3 (MDD Language). *Given a constraint $c(x_1, \ldots, x_n)$ over an ordered set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D(x_1), \ldots, D(x_n)$ the* MDD language *for c is a tuple $\mathcal{M}_c = \langle X, \mathcal{P}, s_\perp, s_\top, T^{\downarrow}, T^{\uparrow}, U, E_t, E_s, R, H\rangle$ where $\mathcal{P}$ is the set of properties used to model states, $s_\perp$ is the source state, $s_\top$ is the sink state, $T^{\downarrow}$ is the forward state transition function, $T^{\uparrow}$ is the reverse state transition function, U is the state update function [6], E_t is the transition existence function, E_s is the state existence function [6], R is the state relaxation function, and H is the trio of heuristics controlling the refinement process [7].*

3 MDDs for Optimization

Consider a COP $\langle X, D, C, f\rangle$ to be solved within the Haddock MDD framework. Without loss of generality, assume for now that for all constraints in C, Haddock has an MDD language for that constraint. Compiling the COP to solve it within Haddock requires one to *compose* the MDD languages for each constraint $c \in C$ as well as an MDD language for the objective function f. To carry out this compilation, one must rewrite the objective function $\{\min, \max\} f$ into an additional constraint of the form $z = f$ where z is an auxiliary variable, replace the objective with $\{\min, \max\}\ z$, and obtain an MDD language for this objective to be composed with the rest.

Some restrictions on f are needed. Since it is meant to model some form of transition costs over prefixes of the variable list, it is required to be separable

(e.g., additive). Any inductive definition for f would meet this requirement. A simple example is $\sum_{i=1}^{n}(x_i \in \Sigma)$ which counts the number of variables taking their value from a prescribed set Σ. Likewise, a weighted sum $\sum_{i=1}^{n} w_i x_i$ that captures transition costs is acceptable.

Definition 4 (MDD Language for Objective Function). *Given an objective function* $\{\min, \max\} f(x_1, \ldots, x_n)$ *over an ordered set of variables* $X = \{x_1, \ldots, x_n\}$ *with domains* $D(x_1), \ldots, D(x_n)$ *let the auxiliary* z *be defined as* $z = f(x_1, \ldots, x_n)$ *and the concrete objective be* $\{\min, \max\}\ z$*. Then, the* MDD language *for the objective* $\{\min, \max\}\ f$ *is*

$$\mathcal{M}_f \;=\; \langle X, \mathcal{P}, s_\bot, s_\top, T^\downarrow, T^\uparrow, -, E_t, -, R, -, \{\min, \max\}\ z \rangle$$

where $\mathcal{P}$ *is the set of properties used to model states (for* $z = f(x_1, \ldots, x_n)$*),* $s_\bot$ *is the source state,* $s_\top$ *is the sink state,* $T^\downarrow$ *is the forward state transition function,* $T^\uparrow$ *is the reverse state transition function,* E_t *is the transition existence function, and* R *is the state relaxation function. Dashes denotes the absence of state update, state existence, and heuristic bundles.*

A few observations are in order. First, the auxiliary z is not a model variable and therefore does not occupy a layer in the MDD. Second, the auxiliary z is typically used within E_t to filter arcs that cannot produce solutions of the desired quality. Third, the source and sink states, respectively $s_\bot$ and $s_\top$, hold properties related to f (and therefore z) that pertain to all source-sink paths in the MDD and will be used to read both primal and dual bounds. Fourth, internal states of the MDD hold properties for f that are related to the source-sink paths going through that specific node.

Example 1 (Minimize a Weighted Sum Objective). For the objective function $\min \sum_{i=1}^{n} w_i \cdot x_i$, the auxiliary z is defined as $z = \sum_{i=1}^{n} w_i \cdot x_i$ and is associated to properties L and U giving the lower and upper bounds on f in both the up and down directions in the diagram. As a result, the values $L^\downarrow(s_\top)$ and $L^\uparrow(s_\bot)$ represent a lower bound[3] for z in a relaxed MDD while, for any internal state s, $L^\downarrow(s) + L^\uparrow(s)$ denotes z's bound for any internal state for all paths going through s. The transition functions are simply

$$T^\downarrow(a, b, i, \ell) = \langle L^\downarrow(a) + (w_i \cdot \ell), U^\downarrow(a) + (w_i \cdot \ell), L^\uparrow(b), U^\uparrow(b) \rangle$$
$$T^\uparrow(b, a, i, \ell) = \langle L^\downarrow(a), U^\downarrow(a), L^\uparrow(b) + (w_i \cdot \ell), U^\uparrow(b) + (w_i \cdot \ell) \rangle$$

while the relaxation of two states a and b is:

$$R(a, b) = \langle\ \min\{L^\downarrow(a), L^\downarrow(b)\}, \max\{U^\downarrow(a), U^\downarrow(b)\},\ \min\{L^\uparrow(a), L^\uparrow(b)\}, \max\{U^\uparrow(a), U^\uparrow(b)\}\ \rangle$$

The arc existence function meant to test the viability of a value $\ell \in D(x_i)$ to connect states a and b is

$$E_t(a, b, i, \ell) = U^\downarrow(a) + w_i \cdot \ell + U^\uparrow(b) \geq \min(z) \wedge L^\downarrow(a) + w_i \cdot \ell + L^\uparrow(b) \leq \max(z)$$

[3] For a maximization, $U^\downarrow(s_\top)$ and $U^\uparrow(s_\bot)$ give the upper bound

To derive the Haddock MDD language from a COP $\langle X, D, C, f\rangle$, it suffices to compile

$$\mathcal{M} = \bigwedge_{c \in C} \mathcal{M}_c \wedge \mathcal{M}_f$$

in which $\wedge$ is the MDD composition operator. To search for a global optimum over D using Haddock, one must instantiate propagators for $\mathcal{M}$. While it is often not tractable to maintain an *exact* MDD, it is natural to rely instead on *relaxed* and *restricted* MDD propagators to derive dual and primal bounds and carry out a branch-and-bound search. Given a maximum width w, we can obtain:

Relaxed MDD Let $\underline{M}$ be the relaxed MDD (to width w) where nodes are merged within each layer to never exceed width w;
Exact MDD Let M^* be the exact MDD;
Restricted MDD Let $\overline{M}$ be the restricted MDD (to width w) in which overflow nodes are discarded.

Note that the maximum width w together with the MDD language and a given COP instance will yield a unique relaxed or restricted diagram. (Exact diagrams are always unique for a given variable ordering.) This is because the MDD language also controls any heuristic compilation choices. Therefore, so long as the heuristics do not introduce any randomness, the relaxed or restricted MDD will be unique.

For a decision diagram M, let $\Psi(M)$ be the set of solutions ($s_\bot$-$s_\top$ paths) encoded by M. By construction, we can obtain a bound on z by reading $M.L^{\downarrow}(s_\top)$ from the sink state of M. We have the following results [2]:

Proposition 1. $\Psi(\overline{M}) \subseteq \Psi(M^*) \subseteq \Psi(\underline{M})$.

Proposition 2. $\underline{M}.L^{\downarrow}(s_\top) \leq M^*.L^{\downarrow}(s_\top) \leq \overline{M}.L^{\downarrow}(s_\top)$.

That is, the relaxed MDD $\underline{M}$ delivers a dual bound while the restricted MDD $\overline{M}$ delivers a primal bound.

Example 2 (COP). Consider the COP defined over $X = \{x_1, \ldots, x_4\}$, $z \in \{0, \ldots, 4\}$, and $D(x_i) \in \{0, 1\}$ for $i = 1, \ldots, 4$:

$$COP = \langle X, D, \{\texttt{among}(X, 1, 3, \{1\})\}, \min \sum_{i=1}^{4} (x_i \in \{1\})\rangle$$

With the auxiliary $z = \sum_{i=1}^{4} (x_i \in \{1\})$, the MDD language from $\mathcal{M} = \mathcal{M}_{\texttt{among}} \wedge \mathcal{M}_{\texttt{sum}} \wedge \mathcal{M}_{\min z}$ models the COP. It can be used to compile a relaxed, exact, and restricted diagram as shown in Figure 2, where we impose a maximum width 2 on the relaxed and restricted MDDs. Each state is labeled with properties $(L^{\downarrow}, U^{\downarrow}, L^{\uparrow}, U^{\uparrow})$.[4]

[4] With a slight abuse of notation as we do not repeat the bounds on z and `among` since those properties are identical.

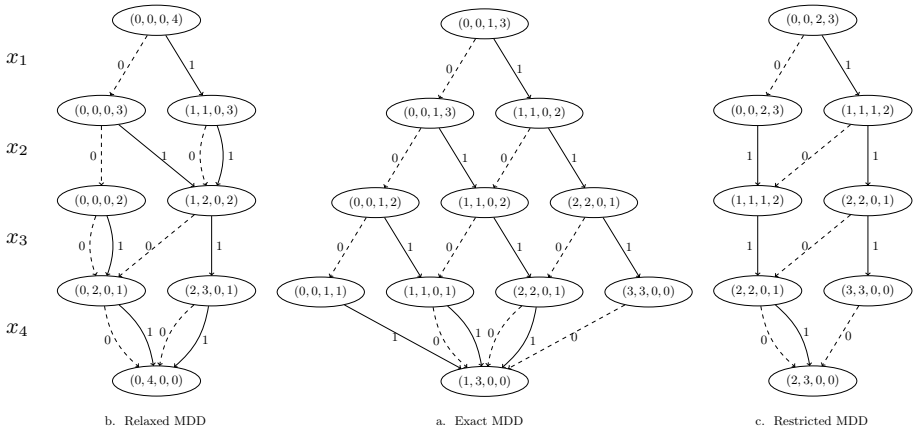

Fig. 2: MDDs for the COP $\langle X, D, \{\texttt{among}(X, 1, 3, \{1\})\}, \min \sum_{i=1}^{4} (x_i \in \{1\})\rangle$ of Example 2.

The CP solver maintains the relaxed and restricted variants within propagators and uses the bounds to drive the search, i.e., z is tightened using both $\underline{M}.L^{\downarrow}(s_\top)$ and $\overline{M}.L^{\downarrow}(s_\top)$. While the Exact MDD only contains paths with sums between 1 and 3, the Relaxed MDD includes paths of value 0 and 4, and the Restricted only contains paths of values 2 and 3. Observe that $\overline{M}.L^{\downarrow}(s_\top)$ yields a primal bound of value 2 while $\underline{M}.L^{\downarrow}(s_\top)$ delivers a dual bound of value 0.

4 Restricted Decision Diagrams

Reference [6] offers a way to compile a propagator for the relaxed diagram $\underline{M}$. This section adapts the mechanism to produce a propagator for the restricted diagram $\overline{M}$, meant to run at a higher priority, to compute primal bounds. When the propagator runs, if the restricted MDD is feasible, the best path through the restricted diagram from source to sink spells out a *witness solution* and its objective value which can be submitted to the solver as a new incumbent (and therefore trigger the usual addition of a global optimality cut based on this primal value). The restricted MDD construction is shown in Algorithm 1. The main loop (lines 2-9) constructs the layers sequentially. Each iteration starts with an empty layer and considers every node and outgoing arc from the prior layer (line 3). If the arc exists, then the transition $T^{\downarrow}$ produces a new state that is added to layer $\overline{\mathcal{L}}_i$. The loop on lines 8-9 trims layer i until it reaches the desired width, discarding the arcs chosen by the `selectState` heuristic introduced in [7]. Lines 10-13 conclude by connecting nodes of the penultimate layer to the sink and making use of the relaxation function R. Note that R is not used anywhere else, preferring instead to discard overflowing states. When creating a restricted MDD with top-down compilation, there are no bottom-up properties, hence the

Algorithm 1 buildRestrictedMDD($\mathcal{M}, [x_1, \ldots, x_n], width$)

1: $\overline{\mathcal{L}_0} = \{s_\bot\}$, $\overline{\mathcal{L}_n} = \{s_\top\}$, $\overline{\mathcal{L}_i} = \emptyset \quad \forall i \in 1..n-1$, $\overline{\mathcal{A}} = \emptyset$
2: **for** $i \in 1..(n-1)$ **do**
3: **for** $s \in \overline{\mathcal{L}_{i-1}}$ and $\ell \in D(x_i)$ **do**
4: **if** $E_t(s, -, i, \ell)$ **then**
5: $s' = T^{\downarrow}(s, -, i, \ell)$
6: $\overline{\mathcal{L}_i} = \overline{\mathcal{L}_i} \cup s'$
7: $\overline{\mathcal{A}} = \overline{\mathcal{A}} \cup s \xrightarrow{\ell} s'$
8: **while** $|\overline{\mathcal{L}_i}| > width$ **do**
9: $\overline{\mathcal{L}_i} = \overline{\mathcal{L}_i} \setminus$ `selectState`$(\overline{\mathcal{L}_i})$
10: **for** $s \in \overline{\mathcal{L}_{n-1}}$ and $\ell \in D(x_n)$ **do**
11: **if** $E_t(s, -, n, \ell)$ **then**
12: $s_\top = R(s_\top, T^{\downarrow}(s, -, i, \ell))$
13: $\overline{\mathcal{A}} = \overline{\mathcal{A}} \cup s \xrightarrow{\ell} s_\top$
14: `return` $\langle[\overline{\mathcal{L}_0}, \cdots, \overline{\mathcal{L}_n}], \overline{\mathcal{A}}\rangle$

Algorithm 2 Filter $\mathcal{F}_{\overline{M}}$ over variables X for MDD language $\mathcal{M}$ and width w

1: $\overline{M}$ = buildRestrictedMDD($\mathcal{M}, X, w$)
2: **for** $f \in$ *solver*.`onSolCallbacks` **do**
3: f(`bestPath`($\overline{M}.s_\bot, \overline{M}.s_\top$),$\overline{M}.L^{\downarrow}(s_\top)$)
4: **if** $\overline{M}$ is exact **then**
5: `failNow`()

transition existence function must be updated to include this possibility. In place of bottom-up properties, one can use the *rough relaxed bounds* introduced in [8]. Line 14 returns the produced restricted diagram.

4.1 Restricted MDDs in Haddock Propagation

Algorithm 2 gives the pseudocode of the restricted propagator. Line 1 builds the restricted diagram (these are not reused across invocations) for the MDD language $\mathcal{M}$ defined over variables in X. The loop on lines 2-3 iterates over the list of callbacks passing down the witness solution for the best path in $\overline{M}$ together with the primal bound for it, i.e., $\overline{M}.L^{\downarrow}(s_\top)$. As long as the callback tightens z's upper bound, the COP will be required to improve the incumbent for the remainder of the execution. Finally, line 4 determines whether the diagram is exact or not. If it is exact, then it contains the optimal solution and the search can stop.

4.2 Relaxed MDDs in Haddock Propagation

The propagator for the relaxed MDD $\underline{M}$ is unchanged from [6]. The only difference is that the propagator is accessible by the search procedure as an oracle capable of producing a dual bound on request (produced at its last fixpoint).

4.3 Restricted MDDs and Constraints external to the MDD

One of the strengths of HADDOCK is the ability to support additional constraints with their own propagators within a single solver. The algorithms presented here assume that all constraints are embedded in one MDD. This assumption means every $s_\bot$-$s_\top$ path in a restricted MDD corresponds to a feasible solution. If some constraints are external to the MDD, a solution sent to the solver on Line 3 of Algorithm 2 may violate one or more of those external constraints. Thankfully in this case, the callback f can invoke a sub-solver for all external constraints based on the binding imposed by the witness solution it receives to verify their feasibility. Note that this can entail a nested search [17]. For brevity's sake, this paper only considers models where the MDD contains all constraints.

5 Best-First Search

CP often uses a depth-first search and relies on optimality cuts to discard subtrees that cannot improve upon the incumbent. With both a primal and a dual bound, a best-first search strategy becomes feasible. Consider Algorithm 3 modeled after the DFS in miniCP [12]. Lines 1 and 2 specify the propagators for the restricted and the relaxed MDDs, respectively, of width w associated to $\mathcal{M}$. Line 3 creates a priority queue and populates it with an initial problem where the constraints and objective function f are embedded in M. A trivial upper bound for the primal is set to $+\infty$ (without loss of generality, we assume a minimization). Line 5 adds an anonymous function to be called each time an incumbent is produced. The purpose of this lambda is to tighten the primal bound. Lines 6-14 offer the main loop. Each iteration starts in line 7 with pulling the most promising node from the queue. Line 8 propagates this node fully with the filtering associated to all constraints to obtain the refined domains D'. Line 9 picks a variable to branch on (i.e., x_i). The loop spanning lines 10-14 considers each value v in turn. Line 11 queries layer i of the relaxed MDD to retrieve the state reachable via value v, and line 12 recovers the dual bound for that node. If the node appears viable (line 13), a problem is added to the queue with the revised domain, the binding of x_i to v, and the tightening of the objective.

A few observations are worth making:

- The propagator for $\mathcal{F}_{\overline{M}}$ should be scheduled at a higher priority than the propagator for $\mathcal{F}_{\underline{M}}$ since it is cheaper to compute and has the potential to end the search early. This is easy to achieve with any solver that has at least 2 priority lists.
- The BFS implementation outline above adopts a *lazy* technique by only enqueueing the specification of the new search node in *queue* on line 14 and propagating the effect of the branching constraint only where the node is de-queued on line 7. This lazy strategy dominates the eager when a node is propagated before being added to the queue. The rationale is that propagation is relatively expensive in the context of MDD solvers where a substantial computational effort is expanded when refining the MDD. BFS nodes that

Algorithm 3 $BFS(X = [x_1, \ldots, x_n], D, \mathcal{M}, f, w)$

1: $\mathcal{F}_{\overline{M}}$ = filtering function for the restricted MDD $\overline{M}$ of width w
2: $\mathcal{F}_{\underline{M}}$ = filtering function for the relaxed MDD $\underline{M}$ of width w
3: $queue = \{(\langle X, D, \{\overline{M}, \underline{M}\}\rangle, -\infty)\}$
4: $primal = +\infty$
5: $solver$.`onSolution`$(\lambda X.\lambda z \rightarrow primal = \min(primal, z))$
6: **while** $queue \neq \emptyset$ **do**
7: $(\langle X, D, C\rangle, \_) = queue$.`extractBest`$()$
8: $D' = \mathcal{F}_C(D)$
9: $i = \min_{1\ldots n}\{i \mid x_i \text{ is not bound}\}$
10: **for** $v \in D(x_i)$ **do**
11: $s = \underline{M}.\mathcal{L}_i$.`stateWithIncomingArc`$(v)$
12: $dual = L^{\downarrow}(s) + L^{\uparrow}(s)$
13: **if** $dual < primal$ **then**
14: $queue = queue \cup (\langle X, D', C \cup \{x_i = v, f \geq dual\}\rangle, dual)$

are ultimately fathomed do not have to carry this burden in the lazy approach.

- When branching on x_i, every variable sequentially before x_i is bound. This ensures that every earlier layer in the MDD consists of a single state with one outgoing arc. As a result, when obtaining the state reachable via value v (line 11), there can be only one state because the previous layer consists of a single node. If using a branching technique that does not ensure every previous variable is bound, then there may be multiple states in $\mathcal{L}_i$ reachable via v. In this case, the dual bound on layer 12 would instead be the minimum across all such states in $\mathcal{L}_i$.
- This search bears similarities to the MDD-based branch-and-bound proposed in [3]. In [3], the branching is done on MDD nodes from cutsets consisting exclusively of *exact* nodes, i.e. nodes whose states did not require merging, and required that every $s_\perp$-$s_\top$ path takes at least one node in the cutset. This paper intentionally applied a traditional CP style branching on variables. Yet, it is possible to adopt the same cutset branching provided that the necessary API is provided on an MDD propagator, a task reserved for future work.

6 Empirical Evaluation

HADDOCK is part of `MiniC++`, a C++ implementation of the MiniCP specification [12]. All benchmarks were executed on a Intel Xeon CPU E5-2640 v4 at 2.40GHz with 32GB.

Comparison to MDD-based Branch and Bound We compare constraint optimization in HADDOCK to the *dedicated* MDD-based branch and bound solver from [16]. We downloaded the provided source code for the dedicated

Table 1: Evaluating SOP instances. The maximum MDD width is $w = 64$.

		Haddock				Haddock w/ Priority			B&B		
Instance	n	# Nodes	Time (s)	Dual	Primal	Time (s)	Dual	Primal	Time (s)	Dual	Primal
esc07	9	2	**0.001**	2125	2125	**0.001**	2125	2125	**0.001**	2125	2125
esc11	13	7	**0.003**	2075	2075	**0.003**	2075	2075	0.26	2075	2075
esc12	14	109	0.102	1675	1675	**0.080**	1675	1675	1.89	1675	1675
esc25	27	1456	17.129	1681	1681	**16.786**	1681	1681	1002.01	1681	1681
esc47	49	77278	-	171	1441	-	326	**1427**	-	**335**	1542
esc63	65	46163	-	**21**	**62**	-	**21**	**62**	-	8	**62**
esc78	80	8602	-	2050	**19575**	-	2025	**19575**	-	**2230**	19800
br17.10	18	7995	39.387	55	55	**34.052**	55	55	270.92	55	55
br17.12	18	5765	25.456	55	55	**19.982**	55	55	146.50	55	55
ft53.1	54	18969	-	**2996**	**8198**	-	1625	**8198**	-	1785	8478
ft53.2	54	26150	-	**2322**	8840	-	1729	**8458**	-	1945	8927
ft53.3	54	36033	-	2018	11519	-	2138	**11707**	-	**2546**	12179
ft53.4	54	71566	-	3549	**14758**	-	3681	14776	-	**3773**	14811
ft70.1	71	7293	-	24428	41751	-	24556	**41647**	-	**25444**	41926
ft70.2	71	9546	-	24560	42294	-	24664	**41932**	-	**25237**	42805
ft70.3	71	15824	-	25263	**46497**	-	25220	47232	-	**25809**	48073
ft70.4	71	21663	-	28775	**56477**	-	**28928**	**56477**	-	28583	56644
kro124p.1	101	2800	-	**14667**	45025	-	9556	**44699**	-	10773	46158
kro124p.2	101	3801	-	**13901**	46802	-	10003	**46608**	-	11061	46930
kro124p.3	101	6407	-	10606	**55137**	-	10882	**55137**	-	**12110**	55991
kro124p.4	101	9854	-	**16524**	**84492**	-	15297	84685	-	13829	85533
p43.1	44	37428	-	375	**28785**	-	350	29090	-	**630**	29450
p43.2	44	68066	-	405	**28770**	-	370	29010	-	**440**	29000
p43.3	44	76591	-	505	**29530**	-	510	**29530**	-	**595**	**29530**
p43.4	44	121550	-	960	83800	-	1015	**83760**	-	**1370**	83900
prob.42	42	93554	-	90	271	-	**106**	**263**	-	99	289
prob.100	100	4232	-	166	**1673**	-	163	**1673**	-	**170**	1841
rbg048a	50	51446	-	55	**369**	-	60	**369**	-	**76**	379
rbg050c	52	59245	-	**70**	**500**	-	56	**500**	-	63	566
rbg109a	111	35077	-	**313**	**1127**	-	307	**1127**	-	91	1196
rbg150a	152	16265	-	**354**	**1863**	-	201	**1863**	-	63	1874
rbg174a	176	9447	-	**453**	**2156**	-	335	**2156**	-	118	2157
rbg253a	255	4408	-	**538**	**3178**	-	390	**3178**	-	112	3181
rbg323a	325	3492	-	**678**	3380	-	416	**3370**	-	89	3519
rbg341a	343	3321	-	**319**	**2968**	-	246	2970	-	68	3038
rbg358a	360	2074	-	**181**	**3202**	-	175	**3202**	-	69	3359
rbg378a	554	1789	-	**196**	**3402**	-	67	**3402**	-	52	3429
ry48p.1	49	33024	-	**6414**	16892	-	4668	**16763**	-	5198	17555
ry48p.2	49	46264	-	**6284**	17439	-	4908	**17410**	-	5290	18046
ry48p.3	49	46053	-	5772	**20890**	-	5793	20962	-	**6208**	21161
ry48p.4	49	41435	-	12443	33391	-	**14576**	**33261**	-	13598	34517

solver[5], compiled, and ran it on the same machine as Haddock. We evaluate the implementations on the Sequential Ordering Problem (SOP) from [16]. This problem can be represented as an asymmetric traveling salesman problem with precedence constraints. Given n elements labeled $v_1, \ldots, v_n$ with asymmetric arcs connecting them, the objective is to find a minimum path from v_1 to v_n visiting each element once and respecting precedence constraints. The precedence constraints are defined as a precedence ordering of v_i before v_j, the index of v_i in the path must be before the index of v_j. The solvers were tested on the 41 SOP problems in TSPLIB [15].

Haddock represents the problem as the composition of an AllDifferent, a sum (for the TSP distances), and a global ordering (that encapsulates all prece-

[5] Source code located at https://github.com/IsaacRudich/PnB_SOP.

dence constraints) MDD languages. The language for the sum is a modified version from Section 3 to use the appropriate weight value in the transition functions. The language for the global ordering constraint is very simple, only requiring one forward property and one reverse property to track which elements have been selected. The solver uses n variables labeled x_1 to x_n with domains $D(x_i) = \{1, \ldots n\}$ where the value of $x_i = v$ means element v is in position i of the sequence. Variables x_1 and x_n are restricted to be 1 and n, respectively. Following [7], the model uses heuristics to refine the MDD. First, the model uses equality for the equivalence function and prioritizes refinement to favor states with a smaller $L^{\downarrow}$. Second, in the initial refinement iteration, we make use of an approximate equivalence function to split nodes based on incoming arc values. We use a `maximum reboot distance` of 100.

All experiments use a 1-hour timeout and record the primal and dual solutions as well as the time taken to terminate. Results appear in Table 1. Bold-faced entries report which solver terminates first (time) or with the best bounds (and thus best incumbent for the primal bound). The "Haddock" columns correspond to the default heuristics while "Haddock w/ Priority" refers to boosting the priority of the ordering constraint. The columns for "B&B" refer to the dedicated MDD-based branch-and-bound method from [16].

Out of the 41 instances, 6 terminate in under an hour. These terminate for Branch and Bound as well but with longer runtimes. This is most likely attributable to the impact of the heuristics used within the relaxed MDD propagator for merging MDD nodes. Without taking advantage of constraint priority, Haddock still obtained better times in the 6 terminating instances. Setting the ordering constraint at top priority, the bounds obtained by Haddock improve for several instances. For example, the dual bound for `esc47` increases from 128 to 336. However, we also observe a couple instances where this heuristic negatively impacts the dual bound. Most notably, `rbg150a` and `rbg341a` both fail to obtain a meaningful dual bound. A limited number of heuristics were tested in Haddock, which leads us to speculate that other heuristics may give tighter bounds within the same time frame. For benchmark instances that timeout after one hour, Haddock obtains competitive bounds compared to Branch and Bound. In most instances, Haddock has a better primal (incumbent) while the dual bound is often marginally weaker. Exceptions where the dual bound is better do exist, e.g., `esc63`, `ft70.4`, `kro124p.4`, `prob.42`, `rgb109a`. From a dual bound standpoint, it leads to the conclusion that neither solver dominates and the difference are most likely attributable to the differences in heuristics with the relaxed MDD propagator with the heuristic used in Haddock being either a better or worse fit depending on the benchmark structure.

Table 2: Time(s) and search nodes to reach target dual bound at different widths.

		$w = 32$		$w = 64$		$w = 128$		$w = 256$	
Instance	Target Dual	Time (s)	# Nodes	Time (s)	#Nodes	Time (s)	#Nodes	Time (s)	# Nodes
esc78	1800	365.457	1707	389.607	916	830.739	723	2210.184	701
ft70.4	28000	59.153	1186	121.803	1083	252.895	1005	530.146	947
prob.42	80	221.997	11870	438.860	10751	1031.687	10909	2124.905	10695
ry48p.2	5500	165.818	4471	73.132	690	210.550	690	685.847	690

Effects of width Table 2 shows how the performance of Haddock scales with the specified width on a subset of benchmarks from the various classes of instances. Since those are larger instances that time out at 1 hour, to have a better comparison, the solvers were asked to stop once they reached a target value for the dual bound (reported in the second column). Note how, as observed before, there is a sweet spot for the width for which runtime is minimized. Also, the number of nodes for the branch-and-bound tree tends to reduce as width increases. Naturally, since the algorithm is not executed to its natural termination (with an optimality proof) the results should be interpreted conservatively.

Table 3: Impact of Restricted MDDs for the primal bound on BFS.

		Depth-First Search Haddock			Best-First Search Haddock		
Instance	n	Time (s)	Dual	Primal	Time (s)	Dual	Primal
esc07	9	0.002	2125	2125	0.001	2125	2125
esc11	13	0.079	2075	2075	0.003	2075	2075
esc12	14	0.737	1675	1675	0.102	1675	1675
esc25	27	605.560	1681	1681	17.129	1681	1681
esc47	49	-	-	7655	-	171	1441
esc63	65	-	-	170	-	21	62
esc78	80	-	-	29340	-	2050	19575
br17.10	18	55.099	-	55	39.387	55	55
br17.12	18	10.298	-	55	24.456	55	55

Comparison to Depth-First Search without Restricted MDDs Table 3 highlights the impact of using Best-First Search with restricted MDDs. DFS finds and proves optimality on the same instances that BFS did. Yet, in all but one of these cases, DFS takes longer. In the exception (br17.12), it appears that DFS gets 'lucky' and finds the optimal solution quickly with the search strategy alone. In other cases, DFS takes over a factor 10 longer, and when the instance takes over an hour, not only does DFS have a weaker incumbent solution, but it has no dual bound (effectively a dual bound of 0).

Comparison to Peel & Bound We ran the Julia implementation of Peel & Bound on our hardware and share in Table 4 a qualitative comparison between Haddock and the results from [16]. First Haddock appears to remain competitive w.r.t. runtime. In addition, Haddock produces primal bounds within the 1-hour timeout that rival (and often exceeds) those produced by peel & bound. Finally, the dual bounds from peel & bound seem quite competitive, overtaking both Haddock and classic Branch & Bound with only a few exceptions.

7 Conclusion

This paper studied the automatic use of primal and dual bounds from Multi-valued Decision Diagrams (MDDs) in the context of branch-and-bound within a CP solver. The paper extended Haddock to support both relaxed and restricted diagrams for any constraints for which a labeled transition system can

Table 4: SOP Instances. Results from [16] at $w = 64$ for Peel & Bound.

		Haddock			Branch & Bound			Peel & Bound		
Instance	n	Time (s)	Dual	Primal	Time (s)	Dual	Primal	Time(s)	Dual	Primal
esc07	9	**0.001**	2125	2125	**0.001**	2125	2125	**0.001**	2125	2125
esc11	13	**0.003**	2075	2075	0.26	2075	2075	0.10	2075	2075
esc12	14	**0.102**	1675	1675	1.89	1675	1675	0.67	1675	1675
esc25	27	**17.129**	1681	1681	1002.01	1681	1681	319.47	1681	1681
esc47	49	-	171	**1441**	-	335	1542	-	**364**	1676
esc63	65	-	21	**62**	-	8	**62**	-	**44**	62
esc78	80	-	2050	**19575**	-	2230	19800	-	**4950**	20045
br17.10	18	39.387	55	55	270.92	55	55	**11.36**	55	55
br17.12	18	25.456	55	55	146.50	55	55	**25.26**	55	55
ft53.1	54	-	2996	**8198**	-	1785	8478	-	**3313**	8244
ft53.2	54	-	2232	8840	-	1945	8927	-	**3419**	**8815**
ft53.3	54	-	2018	**11519**	-	2546	12179	-	**4198**	12482
ft53.4	54	-	3549	**14758**	-	3773	14811	-	**6398**	14862
ft70.1	71	-	24428	41751	-	25444	41926	-	**31077**	**41607**
ft70.2	71	-	24560	**42294**	-	25237	42805	-	**31190**	42623
ft70.3	71	-	25263	**46497**	-	25809	48073	-	**31823**	47491
ft70.4	71	-	28775	**56477**	-	28583	56644	-	**35895**	56552
kro124p.1	101	-	14667	**45025**	-	10773	46158	-	**17541**	46158
kro124p.2	101	-	13901	**46802**	-	11061	46930	-	**17608**	46930
kro124p.3	101	-	10606	**55137**	-	12110	55991	-	**18542**	55991
kro124p.4	101	-	16524	**84492**	-	13829	85533	-	**24316**	85316
p43.1	44	-	375	**28785**	-	**630**	29450	-	380	29390
p43.2	44	-	405	**28770**	-	**440**	29000	-	420	29080
p43.3	44	-	505	**29530**	-	**595**	**29530**	-	480	**29530**
p43.4	44	-	960	**83800**	-	**1370**	83900	-	1010	83880
prob.42	42	-	90	**271**	-	**99**	289	-	94	289
prob.100	100	-	166	**1673**	-	170	1841	-	**174**	1841
rbg048a	50	-	55	**369**	-	**76**	379	-	45	380
rbg050c	52	-	70	**500**	-	63	566	-	**154**	512
rbg109a	111	-	313	**1127**	-	91	1196	-	**372**	1196
rbg150a	152	-	354	**1863**	-	63	1874	-	**563**	1865
rbg174a	176	-	453	**2156**	-	118	2157	-	**623**	**2156**
rbg253a	255	-	538	**3178**	-	112	3181	-	**707**	3181
rbg323a	325	-	**678**	**3380**	-	89	3519	-	281	3529
rbg341a	343	-	**319**	**2968**	-	68	3038	-	318	3064
rbg358a	360	-	**181**	**3202**	-	69	3359	-	72	3384
rbg378a	380	-	**196**	**3402**	-	52	3429	-	50	3429
ry48p.1	49	-	**6414**	**16892**	-	5198	17555	-	6140	17454
ry48p.2	49	-	6284	**17439**	-	5290	18046	-	**6442**	17970
ry48p.3	49	-	5772	**20890**	-	6208	21161	-	**6874**	21142
ry48p.4	49	-	12443	**33391**	-	13598	34517	-	**14171**	33804

be specified. The paper described the derivation of the implementation and recognizes the possibility for extending this work to include branching directly on MDD nodes and supporting hybrid CP models that mix MDD propagators with conventional constraints. The empirical evaluation established that the generic implementation one derives is competitive with state of the art *dedicated* MDD branch-and-bound procedures including peel & bound.

Acknowledgements

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

References

1. N Beldiceanu and E Contejean. Introducing Global Constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.
2. D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.
3. D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Discrete Optimization with Decision Diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.
4. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, pages 146–150, NLD, August 2004. IOS Press.
5. Matteo Fischetti, Fred Glover, and Andrea Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, September 2005. doi:10.1007/s10107-004-0570-3.
6. R. Gentzel, L. Michel, and W.-J. van Hoeve. Haddock: A language and architecture for decision diagram compilation. In *Principles and Practice of Constraint Programming. CP 2020*, volume 12333 of *Lecture Notes in Computer Science*, pages 531–547. Springer, Cham, 2020.
7. R. Gentzel, L. Michel, and W.-J. van Hoeve. Heuristics for MDD Propagation in HADDOCK. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
8. X. Gillard, V. Coppé, P. Schaus, and A. A. Cire. Improving the filtering of branch-and-bound mdd solver. In Peter J. Stuckey, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 231–247, Cham, 2021. Springer International Publishing.
9. Gilles Pesant Gilles, Claude Guy Quimper, and Alessandro Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43, 2012. doi:10.1613/jair.3463.
10. S. Hoda, W.-J. van Hoeve, and J. N. Hooker. A Systematic Approach to MDD-Based Constraint Programming. In *Proceedings of CP*, volume 6308 of *LNCS*, pages 266–280. Springer, 2010.
11. R. M. Keller. Formal Verification of Parallel Programs. *Communications of the ACM*, 19(7):371–384, 1976.
12. L. Michel, P. Schaus, and P. Van Hentenryck. Minicp: a lightweight solver for constraint programming. In *Mathematical Programming Computation*, volume 13, page 133–184, 2021.
13. Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012. ISSN: 03029743. doi:10.1007/978-3-642-29828-8_15.
14. Philippe Refalo. Impact-based search strategies for constraint programming. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3258:557–571, 2004.
15. G. Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
16. I. Rudich, Q. Cappart, and L.-M. Rousseau. Peel-and-bound: Generating stronger relaxed bounds with multivalued decision diagrams. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume

235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
17. P Van Hentenryck, L Perron, and J-F. Puget. Search and Strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):1–36, October 2000.