Heuristics for MDD Propagation in Haddock

Rebecca Gentzel
University of Connecticut, Storrs CT 06269, USA

Laurent Michel
Synchrony Chair in Cybersecurity, University of Connecticut, Storrs CT 06269, USA

Willem-Jan van Hoeve
Carnegie Mellon University, Pittsburgh PA 15213, USA

Abstract

HADDOCK, introduced in [11], is a declarative language and architecture for the specification and the implementation of multi-valued decision diagrams. It relies on a labeled transition system to specify and compose individual constraints into a propagator with filtering capabilities that automatically deliver the expected level of filtering. Yet, the operational potency of the filtering algorithms strongly correlate with heuristics for carrying out refinements of the diagrams. This paper considers how to empower HADDOCK users with the ability to unobtrusively specify various such heuristics and derive the computational benefits of exerting fine-grained control over the refinement process.

2012 ACM Subject Classification Mathematics of computing → Decision diagrams; Theory of computation → Constraint and logic programming

Keywords and phrases Decision Diagrams

Digital Object Identifier 10.4230/LIPIcs.CP.202.27

1 Introduction

Heuristics are a key ingredient in Constraint Programming. They have been at the core of search procedures for decades. The first-fail heuristic [15] is probably the most well-known representative of how one can affect the performance of a constraint solver with a mere influence on the search strategy that guides the branching process towards the most promising variables. Modern constraint programming solvers typically offer a full complement of such heuristics including weighted degree [8], impact-based search [23], activity-based search [21], conflict-driven search [25], or counting-based search [13] to name just a few. This practice is equally common in mathematical programming with strong branching [3, 1] or pseudo-cost branching [10] or even machine learning based heuristics [5]. This is also true in Boolean satisfiability, with LRB (Learning Rate Branching) [20] and VSIDS (Variable State Independent Decaying Sum) [22] being two of the most regarded such heuristics.

Yet, all these heuristics operate on the level of the entire model and exploit "global behaviors" of the solvers. In constraint programming, for instance, the propagators of most constraints use a prescribed level of consistency when they execute, which dictates the fixpoint they reach. This often leaves little to no room for heuristics to play a role *within* the propagators themselves; however, this is not always true. Cost-based filtering propagators [9, 24] can make use of relaxations to derive bounds on the objective function of a model and use that signal to filter variable domains. Recently, [7] showed how to seek specific Lagrangian multipliers that improve filtering. It is notable that the adoption of relaxations within propagators creates opportunities for heuristics.

Decision diagrams present similar opportunities. When applied to optimization problems, multi-valued decision diagrams (MDDs) typically adopt a bounded width (the maximum number of nodes in a layer) and therefore employ some form of relaxation to merge nodes of the diagram [2, 14, 6]. Such merging decisions induce the presence of paths in the MDD

that no longer correspond to solutions, necessitating a search process to seek solutions. During the search, internal nodes belonging to layers of the MDD propagator get filtered out (possibly leading to the filtering of variable domains) which reduces the layer size and prompts refinement phases. Indeed, a depleted layer has room to accommodate more nodes that only currently exist in a latent form as part of another, merged node within the layer. Merging and refining nodes are core operations that raise key questions about the impact of choices made on the quality of the obtained relaxation. *The purpose of this paper is to explore the impact of such choices and provide the solver user with a way to dictate the policies that govern relaxation-inducing choices.* Our findings can potentially be applied to any solver that uses relaxed decision diagrams [6, 11, 12].

Haddock [11] provides a specification language and implementation architecture for automatic decision diagram compilation. Haddock provides the rules for refining (splitting) and filtering (propagating) MDD abstractions. The filtering rules are determined by the properties and functions detailed in the specification language, but the refinement process is more abstract. While the filtering rules give valuable tools to remove arcs and states from the MDD, how the MDD is split determines whether filtering rules are able to find infeasible arcs and states and to ultimately filter domains [14].

Contributions. This paper presents an approach to MDD refinement containing configurable heuristics that integrate into Haddock such that all existing Haddock solutions still fit the framework. These heuristics allow the tailoring of refinement rules to specific constraints or models. The rules for refinement play a large role in MDD propagation, and we present insights into why certain refinement rules outperform others.

Paper Structure. The remainder of the paper is organized as follows. Section 2 introduces a motivating example using `among` constraints. Section 3 reviews the relevant preliminaries, including the formalization used in Haddock. Section 4 discusses the heuristics that parameterize the refinement strategy. Section 5 treats the aggressiveness of the refinement process across layers through the reboot hyper-parameter, while Section 6 reports on the empirical results, and Section 7 concludes the paper.

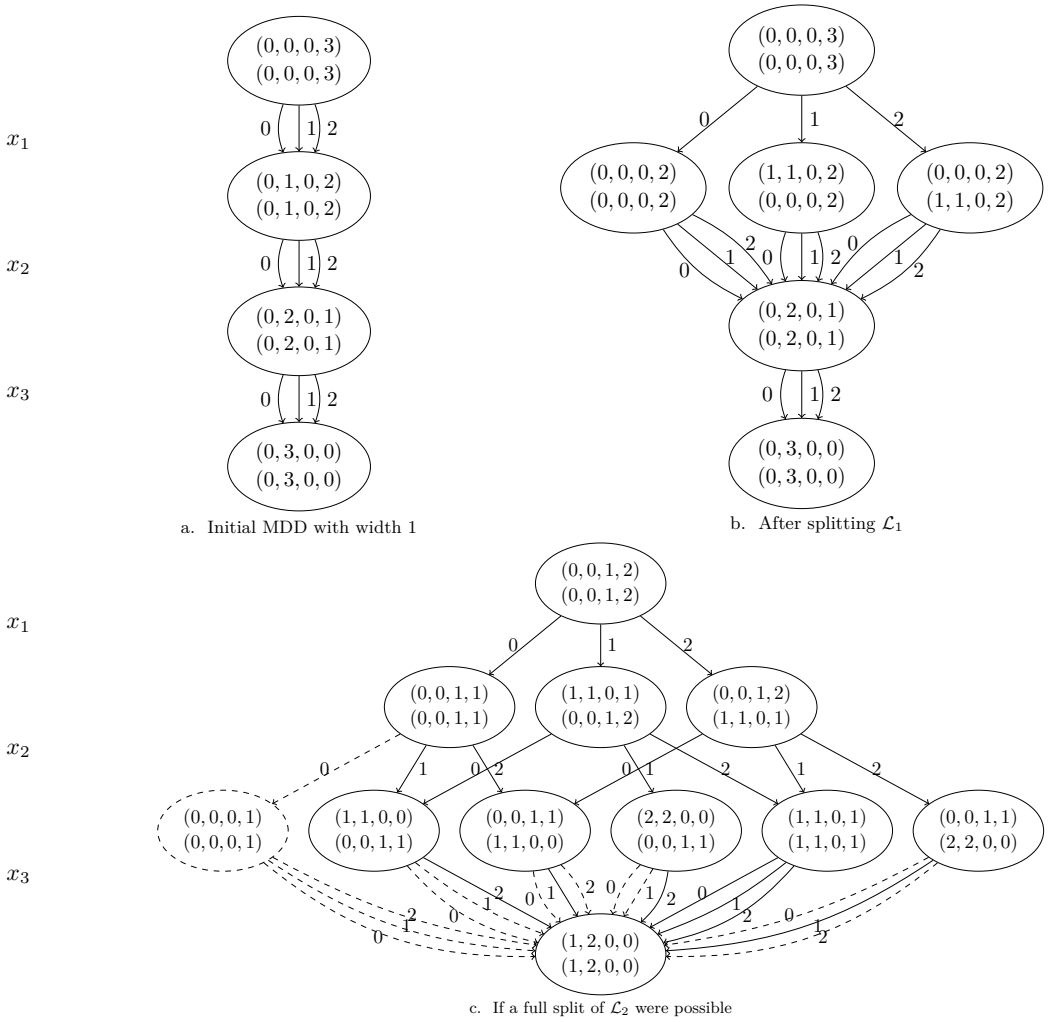
2 Motivating Example

The following example explores the impact that state selection can have on the accuracy of the relaxation produced by an MDD propagator.

▶ **Example 1.** Recall the definition of the `among` global constraint on an ordered set X of n variables [4]. 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.,

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

Consider two constraints $c_1 = \textsc{Among}(\{x_1, x_2, x_3\}, l_1 = 1, u_1 = 2, \Sigma_1 = \{1\})$ and $c_2 = \textsc{Among}(\{x_1, x_2, x_3\}, l_2 = 1, u_2 = 2, \Sigma_2 = \{2\})$ where each variable has domain $\{0, 1, 2\}$. An MDD for these constraints is a layered directed acyclic graph with four layers $(\mathcal{L}_0, \ldots, \mathcal{L}_3)$, a source $s_\bot$, and a sink $s_\top$. Arcs flow from a node in layer $\mathcal{L}_{i-1}$ to a node in layer $\mathcal{L}_i$ and are labeled with a domain value v, stating the assignment $x_i = v$. Every $s_\bot$-$s_\top$ path denotes a candidate solution. Each node carries a state $s = \langle s_1, s_2 \rangle$ with $s_1 = \langle L_1^\downarrow, U_1^\downarrow, L_1^\uparrow, U_1^\uparrow \rangle$ and $s_2 = \langle L_2^\downarrow, U_2^\downarrow, L_2^\uparrow, U_2^\uparrow \rangle$ with the properties of c_1 and c_2. Intuitively, $L_i^\downarrow$ and $U_i^\downarrow$ denote the lower and upper bound, respectively, on the number of occurrences of values from Σ_i on any $s_\bot$-s paths in the MDD. $L_i^\uparrow$ and $U_i^\uparrow$ are similarly defined on s-$s_\top$ paths.



a. Initial MDD with width 1

b. After splitting $\mathcal{L}_1$

c. If a full split of $\mathcal{L}_2$ were possible

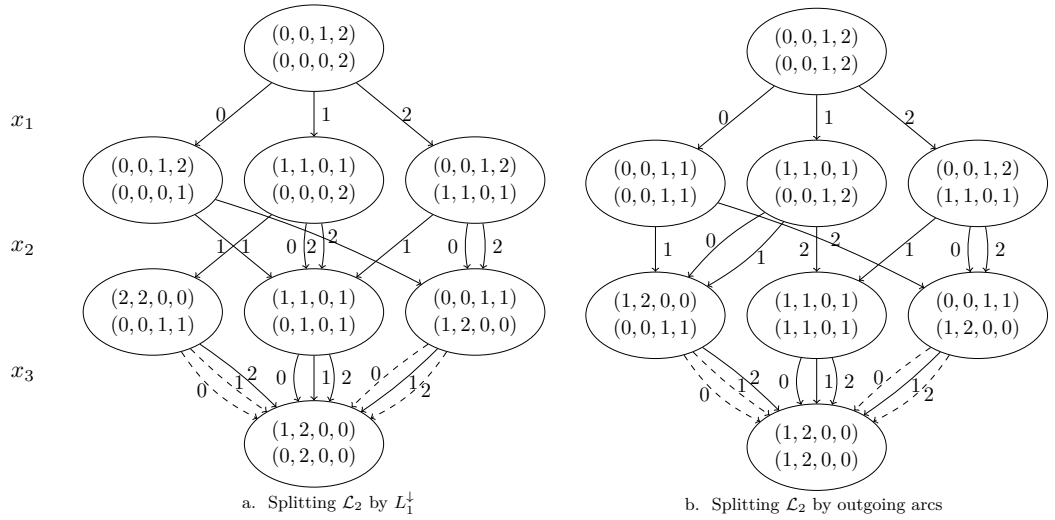
Figure 1 Exact refinement process. Dashed nodes and arcs can be filtered.

Figure 1(a) depicts the MDD at width 1. Assume one imposes a maximum width of 3. Refinement begins by splitting $\mathcal{L}_1$. As shown in Figure 1(b), $\mathcal{L}_1$ can be fully split into three states. Next, refinement is performed on $\mathcal{L}_2$. A full split is shown for this layer in Figure 1(c). While the state on the far left is infeasible and can be deleted, five states remain with a maximum width of 3. A splitting of this layer partitions the five states into three groups. One partitioning strategy is to solely rely on $L_1^{\downarrow}$. Since there are exactly three values for $L_1^{\downarrow}$ in these five states $(0, 1, 2)$, the five states group neatly. The result is shown in Figure 2(a). An alternative is depicted in Figure 2(b) the grouping is based on the labels of outgoing arcs to $s_\top$ ($\{1\}$, $\{2\}$, and $\{0, 1, 2\}$ after filtering infeasible arcs). While the first partition strategy still has $s_\bot$-$s_\top$ paths representing infeasible assignments, e.g. $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, the second partition provides an exact MDD despite $\mathcal{L}_2$ still harboring merged states. It is clear that choices made during refinement impact the accuracy of the MDD and its ability to filter.

3 Background

Following [11], we formally define an MDD as a labeled transition system [17]:

▶ **Definition 2.** *A* labeled transition system *is a triplet* $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ *where* $\mathcal{S}$ *is a set of states,*



a. Splitting $\mathcal{L}_2$ by $L_1^{\downarrow}$ b. Splitting $\mathcal{L}_2$ by outgoing arcs

Figure 2 Options for partitioning $\mathcal{L}_2$. Dashed arcs can be filtered.

$\rightarrow$ *is a relation of labeled transitions between states from* $\mathcal{S}$*, and* Λ *is a set of labels used to tag transitions.*

▶ **Definition 3.** *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 an LTS* $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ *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*. 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 [11].

State Properties As mentioned in Example 1, a state for $\textsc{Among}(X, l, u, \Sigma)$ 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 $s_\perp$ to v.
- $U^{\downarrow} \in \mathbb{Z}$: maximum number of times a value in Σ is taken from $s_\perp$ 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 $s_\perp$ as $\langle 0, 0, -, - \rangle$ and the sink $s_\top$ as $\langle -, -, 0, 0 \rangle$.

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:

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

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.:

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

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* $\texttt{relax}(a, b)$. For Among, we can use:

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

We also call such relaxed states *approximate* states.
State relaxation generalizes to an ordered set of states $\{s_0, s_1, \ldots, s_{k-1}\}$ as follows:

$$\texttt{relax}(s_0, \texttt{relax}(s_1, \texttt{relax}(..., \texttt{relax}(s_{k-2}, s_{k-1})...))).$$

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 [16].

Notation For any state $s \in \mathcal{L}_i$ with $1 \leq i \leq n$, let $\delta^-(s)$ denote the set of inbound arcs from layer $\mathcal{L}_{i-1}$. Likewise let $\delta^+(s)$ denote the set of outbound arcs into $\mathcal{L}_{i+1}$. We sometimes overload notation and use $\delta^-(s)$ and $\delta^+(s)$ to also refer to the set of states in $\mathcal{L}_{i-1}$ and $\mathcal{L}_{i+1}$, respectively, one can reach from s via those arcs.

4 Decision Diagram Refinement

Haddock [11] provides an abstract definition for refining an MDD. For refining one layer, it takes a single state, orders all of that state's incoming arcs, groups these arcs based on equivalence classes, and creates new states for each of these equivalence classes [14]. This process introduces space for multiple heuristics. Which relaxed state is selected for splitting? How should the results of the splitting be ordered and partitioned? This section turns these choices into definable heuristic functions building off of the framework of Haddock.

Algorithm 1 `refineLayer`$(\mathcal{L}_i, [\mathcal{L}_0, \ldots, \mathcal{L}_{i-1}], w, \langle Y, Q, W\rangle)$

Require: $|\mathcal{L}_i| \leq w$
Ensure: $|\mathcal{L}_i| = w \vee \texttt{appx}(\mathcal{L}_i) = \emptyset$
1: **while** $|\mathcal{L}_i| < w \wedge \texttt{appx}(\mathcal{L}_i) \neq \emptyset$ **do**
2: **let** $s^* = \arg\max_{s \in \texttt{appx}(\mathcal{L}_i)} Y(s)$
3: **let** $cs = \texttt{partition}(\texttt{refine}(s^*), Q)$
4: **if** $|cs| \leq w - |\mathcal{L}_i| + 1$ **then**
5: $\mathcal{L}_i = \mathcal{L}_i \setminus \{s^*\} \cup \bigcup_{j=1}^{|cs|} \texttt{relax}(cs_j)$
6: **else**
7: **let** $\pi = \texttt{permutation}(cs) \mid \forall j, k \in 1..|cs| : j \leq k \Rightarrow W(s_{\pi_j}) \leq W(s_{\pi_k})$
8: $\mathcal{L}_i = \mathcal{L}_i \setminus \{s^*\} \cup \bigcup_{j=1}^{w-|\mathcal{L}_i|} \texttt{relax}(cs_{\pi_j}) \cup \texttt{relax}(\bigcup_{j=w-|\mathcal{L}_i|+1}^{|cs|} cs_{\pi_j})$

Algorithm 1 gives the pseudo-code of the layer refinement. It takes as input layer $\mathcal{L}_i$, a target width w and three functions Y, W, and Q (shown in red) that are the embodiment of the user-definable heuristics. The algorithm makes use of several sub-routines (`appx`, `refine`, `partition`, and `permutation`) that will be explained below. Algorithm 1 refines a layer by repeatedly pulling out states that can be refined (if any) and replacing them in the layer by more precise versions given the availability of space in the targeted layer. The Y function drives the selection of the approximate state to replace, while Q and W govern the mechanisms to synthesize the replacement. The section closes with an in-depth discussion of `refineLayer` once all its components are laid out.

4.1 State Selection with Y

The first step is to select which state in the layer $\mathcal{L}_i$ should be refined (line 2 in Alg. 1). When the MDD is first constructed, each layer only has one state, so this is trivial. We therefore assume that $1 < |\mathcal{L}_i| < w$. $\mathcal{L}_i$ may contain both exact and approximate states as a result of prior merging. The function call $\texttt{appx}(\mathcal{L}_i)$ returns the subset of states that are the results of prior approximations (merges). Ideally, one would wish to refine the layer and replace all approximate nodes with exact ones until $|\mathcal{L}_i| = w$. The order in which we select an approximate state s^* for refinement is driven by *state priority functions*:

▶ **Definition 4.** *A* state priority function $Y : \mathcal{S} \to \mathbb{Z}$ *takes as input state* $s = \langle P_0, \ldots, P_{k-1}\rangle$ *and returns an integer value representing its priority where the larger is the more preferable.*

The refinement will retract the selected state s^* from the layer and replace it with an expansion that consists of one or more new states. The size of this expansion drives the remainder of the algorithm. Focusing on Y, several natural choices come to mind. Some are based on the local topology of the MDD around the selected state s^*, while others are *semantics* driven and leverage the properties held within s^*. Recall that the layer is an ordered set (states are ordered within the layer and have a rank between 0 and the cardinality of the set) and that states have topological properties such as the sets of incoming ($\delta^-(s)$) and outgoing ($\delta^+(s)$) arcs. While purely syntactic, these properties may be attractive. As the newest states are the ones most recently refined, the age of states may be a useful metric:

▶ **Example 5** (Rank heuristics). Let $Y(s) = -\texttt{rank}(s)$ be the heuristic to first select the oldest states inserted in the layer. Likewise, one can define $Y(s) = \texttt{rank}(s)$ to first select the nodes that were most recently inserted in the layer.

Another natural option is to consider the in-degree of the state in the MDD to get:

▶ **Example 6** (Degree heuristics). Let $Y(s) = -\delta^-(s)$ be the heuristic to first select low in-degree states, i.e., states that have few parents in the prior layer.

▶ **Example 7** (Semantics-based heuristic). Consider the constraint $\textsc{Among}(X, l, u, \Sigma)$ using state $s = (L^\downarrow, U^\downarrow, L^\uparrow, U^\uparrow)$ with $L^\downarrow$ and $U^\downarrow$ as specified earlier. Define the state selection heuristic $Y(s) = L^\downarrow(s) + L^\uparrow(s)$ to preferentially select a state with the largest lower bound on the number of occurrences of values from Σ on any path $s_\top$ to $s_\bot$. Likewise, the heuristic $Y(s) = -(U^\downarrow(s) + U^\uparrow(s))$ would select the state with the smallest upper bound on the number of occurrences of values from Σ along those paths.

4.2 Candidate Selection with Q and W

Once line 2 of Algorithm 1 has executed, state s^* needs to be refined. To evaluate its incoming arcs, we define the function $A(s)$ that collects the set of arcs leading to state s from the prior layer:

$$A(s) = \{p_j \xrightarrow{\ell_j} s \mid p_j \in \mathcal{L}_{i-1} \wedge \ell_j \in D(x_i)\}$$

Equipped with $A(s^*)$ one can compute what the true endpoint of each arc should have been without relaxation. The outgoing arcs of these endpoints are a subset of $\delta^+(s^*)$ built by removing infeasible arcs from $\delta^+(s^*)$. Namely for a true descendent s' computed from an endpoint in $A(s)$, we have

$$\delta^+(s') = \{s' \xrightarrow{\ell_j} c_j \mid s \xrightarrow{\ell_j} c_j \in \delta^+(s) \wedge E_t(s', c_j, i, \ell_j)\}$$

If $\delta^+(s') = \emptyset$, then the corresponding arc in $A(s^*)$ can be removed from the MDD. With this, we can compute $K(s^*)$, the multiset of true descendants according to the remaining arcs in $A(s^*)$ thanks to the forward state transition rule $T^\downarrow$:

$$K(s) = \{s' = T^\downarrow(p_j, s, i, \ell_j) \mid p_j \xrightarrow{\ell_j} s \in A(s) \wedge \delta^+(s') \neq \emptyset\}.$$

Note how $\texttt{relax}(K(s^*)) = s^*$ since $K(s^*)$ is none other than the multiset of states that would yield s^* if merged. The $\texttt{refine}(s^*)$ function in Algorithm 1 (line 3) is responsible for producing the multiset $K(s^*)$. With unbounded width, one could retain the *unique* states in $K(s^*)$ and add all of them into $\mathcal{L}_i \setminus \{s^*\}$ to upgrade s^*. Otherwise, we need to group together states in $K(s^*)$ to be merged. The generic `partition` function (line 3 in Alg. 1) returns a partition of $K(s^*)$ into multisets $S_1, \ldots, S_p$, each of which representing an approximately equivalent multiset of states. That is, $S_i \subseteq K(s^*)$ for $1 \leq i \leq p$, $S_i \cap S_j = \emptyset$ for $1 \leq i < j \leq p$, and $\bigcup_{i=1}^{p} S_i = K(s^*)$. The heuristic function Q determines which states should be grouped together. For example, if Q is a binary relation that encodes equality, $\texttt{partition}(K(s^*), Q)$ must ensure that $Q(a, b)$ holds for all $a, b \in S_i$ $(1 \leq i \leq p)$ and $Q(a, b)$ does not hold for all $a \in S_i$, $b \in S_j$ $(1 \leq i < j \leq p)$.

Whenever $|S_i| > 1$, we can apply the `relax` function to collapse S_i into a single state. The resulting states can all be added to the layer if it would not exceed maximum width (lines 4-5 in Alg. 1). Otherwise, we need to determine which states to add and which to merge. To do this, we use heuristic function W to compute a sorted permutation of the partition $S_1, \ldots, S_p$. The permutation induced by W identifies the first (and most promising) $w - |\mathcal{L}_i|$ collapsed states for inclusion and merges the remaining ones into a single state.

To formalize the description above, let us adopt the following definitions:

▶ **Definition 8** (Equivalence class). *A* state equivalence function *takes the form* $Q : \mathcal{S} \times \mathcal{S} \to \mathbb{B}$. *It takes as input states* $a = \langle A_0, \ldots, A_{k-1} \rangle$ *and* $b = \langle B_0, \ldots, B_{k-1} \rangle$ *and returns whether the two states are considered similar enough.*

So long as Q is an equivalence relation (reflexive, symmetric, and transitive), Q can generate a partition of $K(s^*)$. Naturally, the most direct example is pure *equality*.

▶ **Example 9** (Equality). Let $\overline{Q}(a, b)$ be a binary state equivalence function that holds over states $a = \langle A_0, \ldots, A_{k-1} \rangle$ and $b = \langle B_0, \ldots, B_{k-1} \rangle$ when all properties are point-wise equal, i.e., $\overline{Q}(a, b)$ holds if and only if $\bigwedge_{i=0}^{k-1} A_i = B_i$.

While combining equal states is helpful, one may wish to group states that are similar but not identical. We refer to all other types of state equivalence as *approximate equivalence*. Which properties are used for determining equivalence may be problem dependent. Hence the desire to make it programmable. Any states that are deemed *approximately equivalent* are relaxed together by virtue of being members of the same class. The desire to preserve a strong relaxation should bias the design of Q to induce the weakest possible losses as a result of applying the `relax` function. To appreciate this *semantic* use, consider this example:

▶ **Example 10** (Bound Slackness). Consider the constraint $\textsc{Among}(x, l, u, \Sigma)$ using state $s = (L^{\downarrow}, U^{\downarrow}, L^{\uparrow}, U^{\uparrow})$ as before. It is easy to assess how close the current bounds on the number of occurrences of values in Σ are compared to l and u. Given two states $a, b \in K(s)$, $a = T^{\downarrow}(p_a, s, i, \ell_a)$ and $b = T^{\downarrow}(p_b, s, i, \ell_b)$. If $L^{\downarrow}(a) + L^{\uparrow}(a)$ and $L^{\downarrow}(b) + L^{\uparrow}(b)$ are equally close to l, one would incur a weak loss of precision when merging a with b since merging uses min on property $L^{\downarrow}$, and $L^{\uparrow}(a) = L^{\uparrow}(b) = L^{\uparrow}(s)$ because a and b are derived by only calling the *forward* state transition rule. The same argument applies to the $U^{\downarrow}, U^{\uparrow}$ properties and the distance to the upper bound u. Therefore, let $Q_t(a, b)$ be a parametric approximate equivalence class (with parameter t) defined as

$$\begin{aligned} Q(a,b) =& ((l - (L^{\downarrow}(a) + L^{\uparrow}(a)) > t) = (l - (L^{\downarrow}(b) + L^{\uparrow}(b)) > t)) \\ & \wedge ((u - (U^{\downarrow}(a) + U^{\uparrow}(a)) > t) = (u - (U^{\downarrow}(b) + U^{\uparrow}(b)) > t)) \end{aligned}$$

Interestingly, setting $t = 0$ means that states a and b are equivalent as soon as both bounds have any amount of slack while $t = +\infty$ means that the inequalities are never satisfied forcing each state to stand in a separate class (no relaxations as a result of similar slackness).

▶ **Definition 11** (Weight function). *A* candidate weight function *takes the form* $W : \mathcal{S} \to \mathbb{Z}$. *It takes as input a state and returns an integer value representing its desirability (smaller is better).*

The weight function is used to derive a permutation of $K(s^*)$. Consider the following examples that leverage simple structural properties:

▶ **Example 12** (Number of arcs heuristic). Let $W(s) = |\delta^-(s)|$ be the heuristic that favors nodes with fewer antecedents in the layer above.

▶ **Example 13** (Parent rank heuristic). Let $W(s) = -\max_{p \in \delta^-(s)} rank(p)$ be the heuristic that favors nodes with parents that were created in the parent layer the most recently.

4.3 Composing Heuristics

Haddock delivers a framework to automatically deliver MDD-driven propagators for constraints through specifications that use state definitions together with several functions to

capture transition, transition existence, state existence, and relaxations. Perhaps even more interestingly, Haddock provides a composition mechanism to produce MDD specifications from the conjunction of multiple high-level constraints. Such composite specifications then drive the generation of the MDD propagator.

The addition of heuristics (Y, Q, and W) to modulate the behavior of the underlying propagator raises a natural question. When each constraint brings its own *preferred heuristics*, how does one combine them into a single composite heuristic for the propagator? We extend the definition of an MDD language from [11] to incorporate the bundle of 3 heuristics:

▶ **Definition 14** (MDD Language). *Given a constraint $c(x_1, \dots, x_n)$ over an ordered set of variables $X = \{x_1, \dots, x_n\}$ with domains $D(x_1), \dots, D(x_n)$ the* MDD language *for c is a tuple $\mathcal{M}_c = \langle X, \mathcal{P}, s_\perp, s_\top, T^\downarrow, T^\uparrow, U, E_t, E_s, R, H = \langle Y, Q, W \rangle \rangle$ 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 rule, $T^\uparrow$ is the reverse state transition rule, U is the state update function, E_t is the transition existence function, E_s is the state existence function [11], and $H = \langle Y, Q, W \rangle$ is the trio of heuristics controlling the refinement process.*

4.3.1 Direct Composition

Consider two MDD languages $\mathcal{M}_1$ and $\mathcal{M}_2$ for constraints c_1 and c_2 defined over overlapping ordered sets of variables X and Y ($X \cap Y \neq \emptyset$). Let the language $\mathcal{M}_1 \wedge \mathcal{M}_2$ denote the composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ and associate to it a heuristic bundle $H_{\mathcal{M}_1 \wedge \mathcal{M}_2}$ defined as:

▶ **Definition 15.** *Given heuristic bundles $H_{c_1} = \langle Y_{c_1}, Q_{c_1}, W_{c_1} \rangle$ and $H_{c_2} = \langle Y_{c_2}, Q_{c_2}, W_{c_2} \rangle$, let $H_{\mathcal{M}_1 \wedge \mathcal{M}_2} = \langle Y_{c_1} + Y_{c_2}, Q_{c_1} \wedge Q_{c_2}, W_{c_1} + W_{c_2} \rangle$ denote the heuristic bundle of the composition.*

4.3.2 Portfolio Composition

While direct composition can be effective, it may be sometimes too restrictive. An MDD may encapsulate several constraints that *disagree* on the guidance that they offer individually. In such circumstances, it might be preferable instead to base the refinements on the advice of a portfolio in which the heuristic bundles coming from each constraint are prioritized. To allow for this, we define the *refinement portfolio* as:

▶ **Definition 16.** *A* refinement portfolio *is an ordered list $(h_1, \dots, h_k)$ of heuristic bundles with $h_i = \langle Y_i, Q_i, W_i \rangle$ for each $i \in \{1, \dots, k\}$.*

To understand how the portfolio is leveraged, consider the fixpoint algorithm used within an MDD propagator for the conjunction of m constraints $\wedge_{i=1}^{m} c_i$ shown in Algorithms 2 and 3. Blue text can be ignored at first as it relates to the reboot and maximum refinement described in Section 5. Algorithm 2 is the core of the fixpoint in the MDD propagator. It first collects into the list HP all the heuristic bundles to be used. It then proceeds in lines 3-9 to carry out passes over the layers of the MDD. Each iteration starts with a backwards pass going over layers $\mathcal{L}_{n-1}$ to $\mathcal{L}_0$ to update the "up" properties of all states. This can lead to the deletion of arcs and states. It then proceeds (line 5) with a down pass to update the forward properties of the states that changed, but also to replenish layers that are no longer full. Finally, lines 6-7 trim the variable domains to echo the changes done to the MDD representation. Any changes prompt another iteration. Algorithm 3 is the crux of the forward pass over layers $\mathcal{L}_1$ to $\mathcal{L}_n$. The loop in lines 3-8 does the layer refinement while lines 9-10 compute the update and the pruning of each layer. While Algorithm 3 implies that the process iterates over all layers, this is a simplification as the implementation only considers changed states in changed layers. That simplification does not affect the layer refinement.

Algorithm 2 mddFixpoint($\mathcal{M}_{c_1 \wedge \cdots \wedge c_m}, [x_1, \ldots, x_n], width, reboot, maxRef$)

```
1: let HP = [⟨Y_1, Q_1, W_1⟩, ..., ⟨Y_k, Q_k, W_k⟩]
2: let iter = 0
3: repeat
4:     changed = computeUp(M_{c_1∧···∧c_m})
5:     changed = computeDown(M_{c_1∧···∧c_m}, width, HP, iter, reboot, maxRef) ∨ changed
6:     for i ∈ 1..n do
7:         trimVariable(x_i)
8:     iter = iter + 1
9: until ¬changed
```

Algorithm 3 computeDown($\mathcal{M}_{c_1 \wedge \cdots \wedge c_m}, width, HP, iter, reboot, maxRef$)

```
1: let changed = false
2: if iter < maxRef then
3:     for hp ∈ HP do
4:         let i = 1
5:         repeat
6:             l = refineLayer(L_i, [L_0, ..., L_{i-1}], width, hp)
7:             i = (l < i) ? max(l, i − reboot) : (i + 1)
8:         until i = n
9: for i ∈ 1..n do
10:    changed = pruneLayer(L_i) ∨ changed
11: return changed
```

4.3.3 Refinement Portfolio Options

Different choices for Q are possible. One could use (for a given constraint c) either an approximation $\widetilde{Q}$ or pure state equality $\overline{Q}$. Alternatively, *both* can be used in a portfolio $[\langle Y, \widetilde{Q}, W\rangle, \langle Y, \overline{Q}, W\rangle]$ that uses them in a round-robin style. This first conservatively expands with a coarse equivalence, and, if room is still available, uses the finer grain equality.

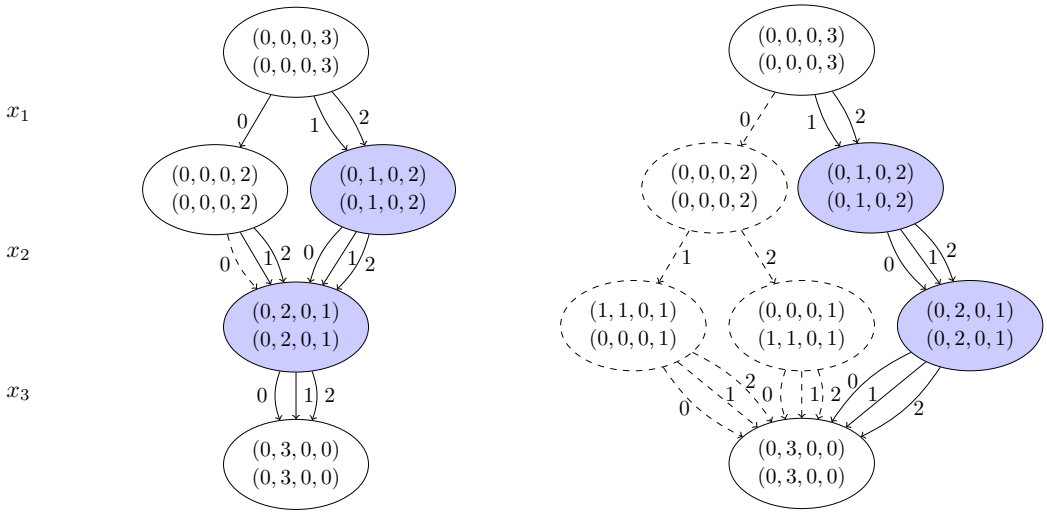
4.3.4 Refinement Portfolio with Constraint Ranking

Another option is to populate the portfolio with heuristic bundles from each constraint embedded in the MDD. Given the constraint set $\{c_1, \ldots, c_m\}$, one can produce a portfolio $HP = [\langle Y_{\pi_0}, Q_{\pi_0}, W_{\pi_0}\rangle, \ldots \langle Y_{\pi_{m-1}}, Q_{\pi_{m-1}}, W_{\pi_{m-1}}\rangle]$ that permutes the bundles according to a user defined ordering π. This can be taken a step further by grouping constraints. Groups have a single heuristic bundle obtained through composition. This grouping of constraints for MDD refinement bears similarities to propagator groups [18]. Both ideas for portfolios compose, expanding HP to include two bundles for each constraint, one that uses $\widetilde{Q}_{\pi_i}$ and one that uses $\overline{Q}_{\pi_i}$. This preserves the ranking goal by prioritizing constraints with a higher rank above constraints of lower rank while always first splitting with $\widetilde{Q}$ before $\overline{Q}$.

5 Layer Processing

5.1 Reboot Distance

The refinement of a layer in Algorithm 1 may terminate with a full layer ($|\mathcal{L}_i| = w$) that still hosts approximate states and has the potential for further refinements. As refinements



a. A refinement of $\mathcal{L}_1$

b. Beginning refinement of $\mathcal{L}_2$

Figure 3 Two Among constraints with maximum width=2. Highlighted nodes are approximate.

proceed through layers, a call to `refineLayer`($\mathcal{L}_i, \ldots$ that causes the deletion of nodes in $\mathcal{L}_i$ and in some preceding layers $\mathcal{L}_l$ can trigger a return to the shallowest layer $\mathcal{L}_l$ to immediately refine it again as opposed to continuing onward from i. When this happens, the referenced loop would 'reboot' to layer $\mathcal{L}_l$. It may be desirable to bound how far one might reboot with a maximum reboot distance between l and i. To reflect this, we add to `computeDown` the changes in blue on lines 6-7 of Algorithm 3. We further assume that `refineLayer` is modified to return the index of the highest layer l changed during the function call.

▶ **Example 17.** Consider an MDD similar to Example 1 with $l_2 = 2$ and maximum width 2. After splitting $\mathcal{L}_1$, we obtain the graph in Figure 3(a). Nodes are highlighted if their state is relaxed. After refining $\mathcal{L}_1$ the right state is still relaxed and cannot be refined due to lack of space in the layer. While splitting $\mathcal{L}_2$, two states in $K(s)$ have no feasible children leading to the deletion of the corresponding arcs in $A(s)$. As a result, a state in $\mathcal{L}_1$ can be removed as shown in Figure 3(b). Without reboot (or $reboot = 0$), $\mathcal{L}_2$ would continue being refined. If $reboot \geq 1$, the refinement will instead elect to further refine $\mathcal{L}_1$ first.

5.2 Maximum Refinement Iterations

Algorithm 2 iterates until a fixpoint is reached. It may be wise to bound the number of times refinement can occur within one call to the fixpoint. We denote this the maximum refinement iterations. The refinement in Algorithm 3 is conditional (line 2) and keeps track of the iteration number in Algorithm 2 (lines 2, 8).

6 Empirical Evaluation

Haddock is part of `MiniC++`, a C++ implementation of the MiniCP specification [19]. All benchmarks were executed on a Macbook Pro with a 3.1 GHz Intel Core i7-5557U processor and 16GB. This section explores the effects of several heuristics on the behavior of the Haddock propagator. Specifically, we consider the following experiments:

Experiment 1 Investigate the impact of the Y and W heuristics.

Experiment 2 Explore the merits of $\widetilde{Q}$, $\overline{Q}$, and a portfolio using first $\widetilde{Q}$, then $\overline{Q}$.

Instance		Width 16				Width 32				Width 64			
		HR	LR	HD	LD	HR	LR	HD	LD	HR	LR	HD	LD
C-I	MA	1.9	3.4	3.6	6.2	2.2	1.5	1.0	1.9	2.1	1.1	0.6	1.1
	LA	5.4	1.5	6.0	2.5	2.3	1.0	1.1	0.9	1.7	0.8	0.6	0.9
	MinPI↓	2.1	0.6	1.4	1.0	1.0	0.7	1.1	0.9	0.4	0.9	0.7	0.8
	MinPI↑	2.2	4.3	5.0	7.9	1.0	1.1	1.2	1.2	1.0	0.9	0.8	1.0
	MaxPI↓	1.7	2.7	1.6	2.2	0.9	1.4	0.9	1.3	0.5	0.6	0.6	0.6
	MaxPI↑	3.7	3.2	5.2	6.7	1.0	1.1	0.8	1.1	1.4	1.0	1.0	0.9
C-II	MA	12.4	9.3	10.6	10.5	5.8	4.5	6.7	4.3	3.8	3.0	4.2	3.2
	LA	19.1	14.2	12.3	12.9	5.7	4.9	4.2	4.7	5.0	2.0	3.9	2.7
	MinPI↓	8.1	5.9	5.2	5.2	2.2	6.5	1.4	5.5	2.0	1.0	1.4	0.8
	MinPI↑	8.2	9.9	10.5	10.1	4.0	2.0	4.7	2.1	3.0	1.5	2.8	1.5
	MaxPI↓	6.7	5.7	4.8	4.2	4.7	4.5	1.4	3.2	1.5	1.9	1.3	1.8
	MaxPI↑	7.7	9.0	10.1	9.2	3.8	3.0	3.6	3.4	2.6	2.9	3.2	2.8
C-III	MA	21.2	28.8	27.2	18.4	19.6	20.7	13.7	19.1	15.9	18.6	14.8	19.8
	LA	17.7	27.7	30.0	24.7	18.7	14.5	15.6	14.1	19.9	14.4	15.1	16.0
	MinPI↓	16.5	18.1	20.1	15.4	16.7	11.1	10.8	11.2	16.1	11.4	13.9	11.5
	MinPI↑	19.5	29.1	23.8	23.6	16.7	16.3	12.8	16.9	17.1	15.8	12.8	15.4
	MaxPI↓	15.5	21.5	13.4	19.5	17.1	12.9	11.9	16.8	13.7	14.8	13.8	17.0
	MaxPI↑	22.4	26.0	27.0	23.5	16.5	11.8	11.9	11.4	16.4	12.7	12.7	12.4

Table 1 CPU time (seconds) to obtain all solutions for Nurse Rostering using $HP = [\langle Y, \widetilde{Q}, W\rangle, \langle Y, \overline{Q}, W\rangle]$ for different Y (columns) and W (rows) heuristics.

Experiment 3 Explore portfolios where constraint groups are prioritized.
Experiment 4 Investigate the impact of *reboots*.
Experiment 5 Investigate how results carry over to MDD propagators with other constraints.

Experiment 1: Role of Y and W. First, we evaluate the performance of the Y and W heuristics on three "nurse rostering" problems from [16], which ask to schedule nurse work shifts over a horizon of 40 days, subject to a collection of Among constraints. There are three classes of instances: Class C-I requires at most 6 out of 8 consecutive work days and at least 22 out of 30 consecutive work days. C-II uses 6 out of 9 and 20 out of 30, while C-III uses 7 out of 9 and 22 out of 30. Each instance also requires 4 or 5 work days each week.

The portfolio was set to use $[\langle Y, \widetilde{Q}, W\rangle, \langle Y, \overline{Q}, W\rangle]$. Namely, layer refinement is driven by approximate equivalence first, followed by strict equality when space is still available. Y and W are selected among the following options:

HR Define $Y(s) = \texttt{rank}(s)$ to select the most recent state first.
LR Define $Y(s) = -\texttt{rank}(s)$ to select the oldest state first.
HD Define $Y(s) = |\delta^-(s)|$ to select the state with largest in-degree.
LD Define $Y(s) = -|\delta^-(s)|$ to select the state with lowest in-degree.
MA $W(s) = -|\delta^-(s)|$ ranks nodes according to decreasing arc set cardinality.
LA $W(s) = |\delta^-(s)|$ ranks nodes according to increasing arc set cardinality.
MinPI↓ $W(s) = -\min_{p\in\delta^-(s)} \texttt{rank}(p)$ ranks nodes with decreasing age of oldest parent.
MinPI↑ $W(s) = \min_{p\in\delta^-(s)} \texttt{rank}(p)$ ranks nodes with increasing age of oldest parent.
MaxPI↓ $W(s) = -\max_{p\in\delta^-(s)} \texttt{rank}(p)$ ranks nodes with decreasing age of youngest parent.
MaxPI↑ $W(s) = \max_{p\in\delta^-(s)} \texttt{rank}(p)$ ranks nodes with increasing age of youngest parent.

Table 1 shows the CPU time taken for each combination of Y and W above. The state equivalence function used for approximate equivalence is from example 10 using $t = 3$, maximal reboot distance is 0 and maximum refinement is 10.

These results indicate that both Y and W have a clear impact on the method. While no single pair Y,W dominate, the LR option for Y seems to fare particularly well. Likewise, `MinPI`↓ and `MaxPI`↓ appear to be consistently effective. We also observe that implementing this generic heuristic framework introduces minimal, if not negligible, overhead.

Experiment 2: Role of $\widetilde{Q}$ vs. $\overline{Q}$. Consider the role of the two equivalence heuristics. Figure 4 graphs the shortest time and least number of backtracks when $\widetilde{Q}$ is used alone, $\overline{Q}$ is used alone, or as a portfolio $[\widetilde{Q}, \overline{Q}]$. At higher widths, the heuristic bundle with $\widetilde{Q}$ stagnates

Figure 4 CPU time (left) and backtracks (right) for finding all solutions for `amongNurse` using different equivalence functions.

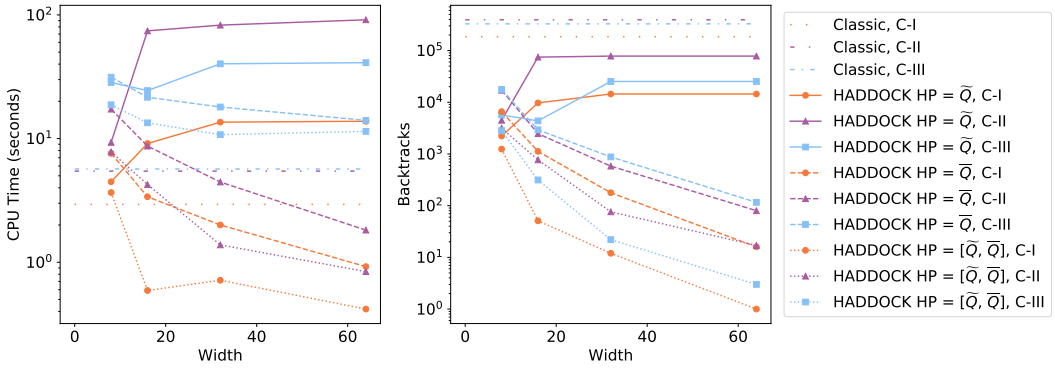
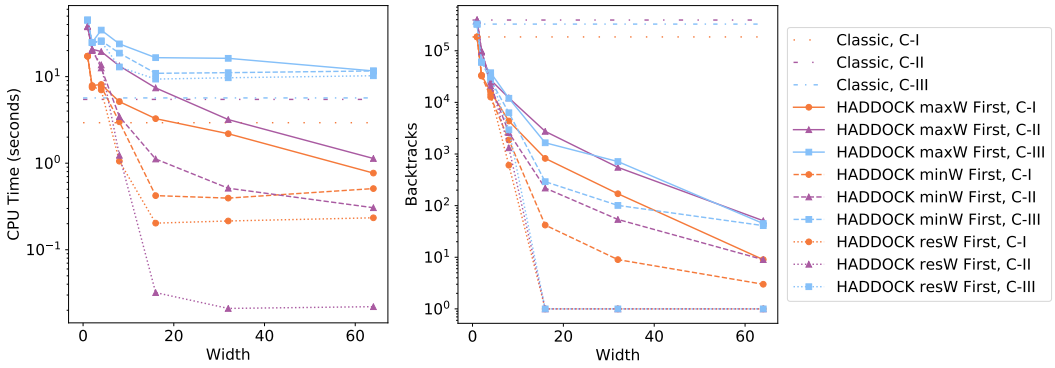


Figure 5 CPU time (left) and backtracks (right) for finding all solutions for `amongNurse` with different constraint group portfolios.



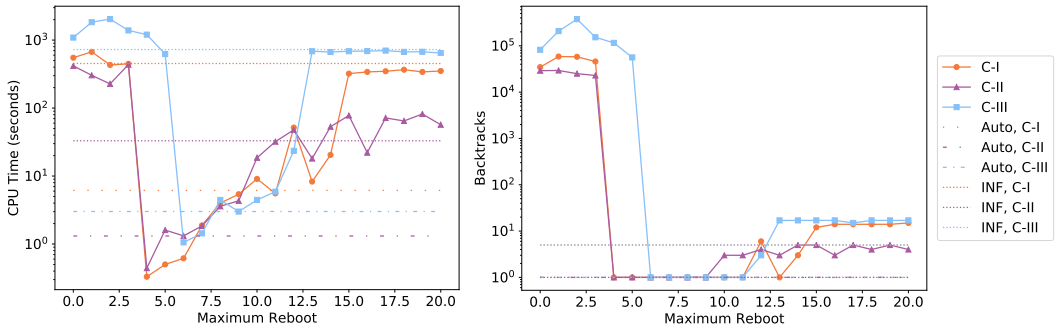
since the approximate equivalence prevents it from making full use of the width. The bundle using $\overline{Q}$ improves as the width increases, which is good. Yet, the best results come from the portfolio which suggest that coarser equivalence is helpful to more judiciously make use of the space in each layer and rely on the stricter $\overline{Q}$ when space is plentiful.

Experiment 3: Portfolio with constraint groups. Given the three classes of constraints that model different aspects (lower bounding the number of work days: $minW$, upper bounding the number of work days: $maxW$ and restricting the number of work days to 4 or 5 in any given week: $resW$) it is tempting to rely on 3 constraint groups and use a portfolio based on the three bundles of heuristics $\{H(minW), H(maxW), H(resW)\}$. To simplify, we test three portfolios: $minW$ First ($[H(minW), H(maxW \wedge resW)]$), $maxW$ First ($[H(maxW), H(minW \wedge resW)]$), and $resW$ First ($[H(resW), H(minW \wedge maxW)]$). Figure 5 shows the results while using $\langle LR, \overline{Q}, \texttt{MinPI}\downarrow\rangle$ for each bundle; the results are quite spread out. The best performance, on all of `C-I`, `C-II`, and `C-III`, occurs whenever $resW$ is the first entry in the portfolio, giving it the first opportunity to drive refinements.

The characteristics of constraints in $resW$ do explain such a behavior. First, these always have the *tightest* bounds ($l = 4$ and $u = 5$). Refining on the tightest constraints may give better opportunities for filtering. Second, the $resW$ constraint groups are always the smallest. Last, $resW$ constraints are stated over disjoint variable sets and since refinements occur on a layer basis (layers are associated to variables) the refinements are more *focused*.

Experiment 4: Reboot for Multiple AllDifferent. The assessment of the `reboot` heuristic is done with randomly generated CSPs that use `allDifferent` constraints, are

Figure 6 CPU time (left) and backtracks (right) for proving infeasibility for `Multiple AllDifferent` across different reboot values using $HP = \langle HR, \overline{Q}, MinPI \downarrow \rangle$.



	reboot	1	2	3	4	5	6	7	8	Auto	INF
A-I	Full	39.2%	66.7%	54.5%	83.8%	94.3%	98.0%	98.1%	98.6%	99.5%	100%
	Time	671.6	430.7	447.0	0.3	0.5	0.6	1.9	4.0	6.2	452.5
A-II	Full	52.2%	66.1%	90.5%	82.5%	94.8%	97.3%	99.3%	99.4%	97.3%	100%
	Time	303.5	226.8	435.4	0.4	1.6	1.3	1.8	3.6	1.3	33.0
A-III	Full	48.8%	46.8%	27.3%	69.6%	66.7%	97.5%	99.1%	99.4%	99.5%	100%
	Time	1834.0	2036.0	1387.2	1202.5	622.6	1.0	1.4	4.4	3.0	725.8

Table 2 Multiple `AllDifferent` for different reboot values using $HP = \langle HR, \overline{Q}, MinPI \downarrow \rangle$ and a width of 16. Each row reports the fraction of full `reboots` and runtime (in seconds).

infeasible and take a non-negligible amount of time to solve with a classic solver. The generator uses the parameters $\langle n, d, [(s_1, f_1, p_1), \ldots, (s_k, f_k, p_k)] \rangle$ where n is the number of variables, d is the domain size, and each (s_i, f_i, p_i) tuple describes a single group of constraints. Group i uses (s_i, f_i, p_i) to produce a set of `AllDifferent` constraints. Each constraint c_k in that set ranges over a random subset (of size ≥ 2) of variables sampled from $\{x_{k \cdot f_i+1}, \ldots, x_{k \cdot f_i+s_i}\}$ where each variable has a probability p_i of being included. Three instances (available online at `http://hidden.url.domain`) were created from $\langle 50, 7, [(3, 1, 1), (6, 6, 1), (10, 1, .3), (8, 5, .6), (20, 7, .2)]$. Performance is measured with time and backtracks to prove infeasibility.

Figure 6 shows the performance using a heuristic bundle of $\langle HR, \overline{Q}, MinPI \downarrow \rangle$ for different maximum reboot values with INF representing an unlimited reboot. A dramatic improvement in performance occurs around `reboots` between 4 and 6 that gets erased as the maximum reboot increases. When a reboot occurs, the refinement either moves as far back as possible or is stopped by the maximum reboot distance (Algorithm 3, line 7). To shed light on Figure 6, consider Table 2 that gives the percentage of *full* reboots across all calls to `computeDown` during the search, that is, reboots that were not cut short. The gains occurs when around $80 - 90\%$. By the time $reboot = 7$, 98% of reboots are full meaning any further increase is unlikely to improve refinements but may still add overhead. In the benchmarks, each `AllDifferent` constraint has at most 7, sometimes fewer, variables. Hence, the reboot *may benefit from staying within the scope of the constraint.* A tempting `Auto` strategy for limiting reboots for any variable x_i associated to layer $\mathcal{L}_i$ is as follows. As usual, let $vars(c)$ denote the set of variables appearing in c and $cstr(x)$ be the set of constraints mentioning variable x. Let $\mathcal{L}(x)$ be the layer of variable x. Then,

$$related(x_i) = \bigcup_{c \in cstr(x_i) \mid |vars(c)| \leq \frac{|X|}{2}} vars(c)$$

in $reboot(i) = \min_{y \in related(x_i)} index(\mathcal{L}(y))$ denotes the layer that the propagator should return to when refinement aborts early. The rationale is to consider the shallowest layer of variables directly related to x_i provided that the constraint connecting them does not cover

		HR	LR	HD	LD
A-I	MA	755.98	920.56	899.14	917.74
	LA	746.80	939.54	925.50	933.94
	MinPI↓	0.91	0.91	0.90	.91
	MinPI↑	795.84	949.96	923.89	935.30
	MaxPI↓	0.90	0.92	0.91	0.92
	MaxPI↑	808.84	961.56	923.37	931.59
A-II	MA	224.45	311.54	304.52	302.08
	LA	228.62	318.84	303.10	308.09
	MinPI↓	1.28	1.29	1.33	1.28
	MinPI↑	203.46	267.36	260.42	270.50
	MaxPI↓	1.29	1.29	1.29	1.31
	MaxPI↑	206.10	268.60	259.29	261.77
A-III	MA	2594.93	3240.10	3553.28	3546.93
	LA	2595.43	3138.61	3481.61	3622.81
	MinPI↓	1.00	390.37	0.87	0.89
	MinPI↑	2420.01	2926.05	3256.71	3316.82
	MaxPI↓	0.98	375.55	0.87	0.85
	MaxPI↑	2507.99	2938.20	3275.93	3321.35

Table 3 CPU time (sec.) to prove infeasibility for Multiple `AllDifferent` using $\overline{Q}$ for different Y (columns) and W (rows) heuristics with the MDD $width = 16$.

`reboot`	0	1	2	3	4	5	6	7	8	`Auto`	INF
Total	0%	14%	44%	54%	66%	75%	85%	87%	93%	49%	100%
CPU Time	5.99	7.29	6.24	7.85	6.72	8.79	9.14	8.77	12.38	6.31	16.12
Backtracks	2960	3682	2672	2848	2187	2280	1735	2030	633	2416	13

Table 4 *AIS* ($n = 11$) for different reboot with $HP = \langle HR, \overline{Q}, MinPI\downarrow \rangle$ and $width = 16$.

a majority of the variables in the CSP. Figure 6 and Table 2 give the results. While the `Auto` strategy does not beat the best static reboot value shown, it performs quite well and avoids the risk of setting the maximum reboot too small or too large.

Experiment 5: Similarities across benchmarks. Last, we check how the heuristics behave across benchmarks. Table 3 gives results for different Y and W using the All Different benchmarks with a reboot of 6. While $MinPI\downarrow$ and $MaxPI\downarrow$ are again the clear favorites for W, HR appears to be the best option for Y. This differs from Nurse Rostering and underlines the usefulness of having programmable heuristics.

To assess whether `Auto` performs on other benchmarks, it is tested on the *All-Interval Series* problem (#007 on CSPLIB) measuring the time, number of backtracks, and percentage of full reboots when looking for all solutions. Table 4 shows the results with $n = 11$. `Auto` picks a good compromise somewhere between 2 and 3 which matches the arity of the absolute value constraints. Using an infinite reboot pays off in backtracks, but not in run time.

7 Conclusion

Heuristics can have a significant impact on the filtering ability of an MDD propagator and ultimately on the efficiency of a model. This paper introduces several heuristics that govern such behaviors, formalized their integration into a generic framework, and reported on the impact they have in practice. Interestingly it led to an *automatic* setting for the `reboot` heuristic. The keystone of the paper is the recognition that such heuristics should be user programmable to get the most out of decision diagram technologies.

Acknowledgments

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

References

1. Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33, 2005. doi:10.1016/j.orl.2004.04.002.
2. H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A Constraint Store Based on Multivalued Decision Diagrams. In *Proceedings of CP*, volume 4741 of *LNCS*, pages 118–132. Springer, 2007.
3. David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William Cook. Finding cuts in the tsp. *Annals of Physics*, 54, 1995.
4. N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.
5. Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290:405–421, 4 2021. doi:10.1016/J.EJOR.2020.07.063.
6. D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. *Decision Diagrams for Optimization*. Springer, 2016.
7. Raphaël Boudreault and Claude-Guy Quimper. Improved cp-based lagrangian relaxation approach with an application to the tsp. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI*, volume 21, pages 1374–1380, 2021.
8. Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *First International Workshop: Constraint Propagation and Implementation*, 2004. URL: http://www.cril.univ-artois.fr/~lecoutre/research/publications/2004/CPW2004.ps.
9. T Fahle and M Sellman. Cost based filtering for the constrained knapsack problem. *Annals of Operations Research*, 115:73–93, 2002.
10. J. M. Gauthier and G. Ribière. Experiments in mixed-integer linear programming using pseudo-costs. *Mathematical Programming*, 12, 1977. doi:10.1007/BF01593767.
11. 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.
12. X. Gillard, P. Schaus, and Coppé. Ddo, a Generic and Efficient Framework for MDD-Based Optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.
13. 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.
14. T. Hadžić, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In P. J. Stuckey, editor, *Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *Lecture Notes in Computer Science*, pages 448–462. Springer, 2008.
15. R M Haralick and G L Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
16. 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.
17. R. M. Keller. Formal Verification of Parallel Programs. *Communications of the ACM*, 19(7):371–384, 1976.
18. M. Lagerkvist and C. Schulte. Propagator groups. In Ian Gent, editor, *Fifteenth International Conference on Principles and Practice of Constraint Programming, Lisbon, Portugal*, volume 5732 of *Lecture Notes in Computer Science*, pages 524–538. Springer-Verlag, 2009.
19. Laurent Michel, Pierre Schaus, Pascal Van Hentenryck. MiniCP: A lightweight solver for constraint programming, 2018. Available from https://minicp.bitbucket.io.

20 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. volume 9710, 2016. doi:10.1007/978-3-319-40970-2_9.

21 Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In Nicolas Beldiceanu, Narendra Jussien, and Éric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, volume 7298 of *Lecture Notes in Computer Science*, pages 228–243. Springer Berlin Heidelberg, 2012. URL: http://dx.doi.org/10.1007/978-3-642-29828-8_15, doi:10.1007/978-3-642-29828-8_15.

22 Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. pages 530–535. ACM, 2001. URL: http://doi.acm.org/10.1145/378239.379017, doi:http://doi.acm.org/10.1145/378239.379017.

23 Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.

24 Meinolf Sellmann, Thorsten Gellermann, and Robert Wright. Cost-based filtering for shorter path constraints. *Constraints*, 12, 2007. doi:10.1007/s10601-006-9006-4.

25 Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219, 2015. doi:10.1016/j.artint.2014.11.006.