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

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 $\Sigma$ and ensures that the total number is between $l$ and $u$, i.e.,

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

Consider two constraints $c_1 = \text{AMONG}([x_1, x_2, x_3], l_1 = 1, u_1 = 2, \Sigma_1 = \{1\})$ and $c_2 = \text{AMONG}([x_1, x_2, x_3], l_2 = 1, u_2 = 2, \Sigma_2 = \{2\})$ where each variable has domain $\{0, 1, 2\}$. An MDD for these constraints is a layered directed acyclic graph with four layers ($L_0, \ldots, L_3$), a source $s_\perp$, and a sink $s_\top$. Arcs flow from a node in layer $L_{i-1}$ to a node in layer $L_i$ and are labeled with a domain value $v$, stating the assignment $x_i = v$. Every $s_\perp \prec \perp s_\top$ path denotes a candidate solution. Each node carries a state $s = (s_1, s_2)$ with $s_1 = (L_1^1, U_1^1, L_1^2, U_1^2)$ and $s_2 = (L_2^1, U_2^1, L_2^2, U_2^2)$ with the properties of $c_1$ and $c_2$. Intuitively, $L_i^1$ and $U_i^1$ denote the lower and upper bound, respectively, on the number of occurrences of values from $\Sigma_i$ on any $s_\perp \prec \perp s$ paths in the MDD. $L_i^1$ and $U_i^1$ are similarly defined on $s \prec \perp s_\top$ paths.
A labeled transition system is a triplet \((S, \rightarrow, \Lambda)\) where \(S\) is a set of states,
Heuristics for MDD Propagation in Haddock

We initialize the state for the source $\ell$-transition.

**Transition Functions** The transition between a node $a \in L_{i-1}$ and $b \in L_i$ is an arc $(a, b)$ labeled by a value $\ell \in D(x_i)$. We use transition functions $T^\perp(a, b, i, \ell)$ 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 $\texttt{AMONG}$, we apply $f(s, p, \ell) = p(s) + (\ell \in \Sigma)$ for each property $p$ in $\langle L^\perp, U^\perp, L^\uparrow, U^\uparrow \rangle$. For example, we define $L^\perp(b) = f(a, L^\perp, \ell)$.
i.e., $L^i(a) + (\ell \in \Sigma)$. We likewise define $L^i(a) = f(b, L^i, \ell)$, $U^i(b) = f(a, U^i, \ell)$ and $U^i(a) = f(b, U^i, \ell)$. The state-level transition functions $T^i$ and $T^\dagger$ compute all the down or up properties of the next state as follows:

$$T^i(a, b, i, \ell) = \langle f(a, L^i, \ell), f(a, U^i, \ell), -,- \rangle$$

$$T^\dagger(b, a, i, \ell) = \langle -, -, f(b, L^i, \ell), f(b, U^\dagger, \ell) \rangle.$$ 

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:

$$T^i(a, b, i, \ell) = \langle f(a, L^i, \ell), f(a, U^i, \ell), L^\dagger(b), U^\dagger(b) \rangle$$

$$T^\dagger(b, a, i, \ell) = \langle L^i(a), U^i(a), f(b, L^i, \ell), f(b, U^\dagger, \ell) \rangle.$$ 

**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.:

$$U^i(a) + (\ell \in S) + U^\dagger(b) \geq l \land L^i(a) + (\ell \in S) + L^\dagger(b) \leq u.$$ 

**Node Relaxation Functions** Two states $a$ and $b$ in the same layer $L_i$ can be relaxed (merged) to produce a new state $s'$ according to a relaxation function $\text{relax}(a, b)$. For AMONG, we can use:

$$\text{relax}(a, b) = \langle \min\{L^i(a), L^i(b)\}, \max\{U^i(a), U^i(b)\}, \min\{L^\dagger(a), L^\dagger(b)\}, \max\{U^\dagger(a), U^\dagger(b)\} \rangle.$$ 

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:

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

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 [16].

**Notation** For any state $s \in L_i$ with $1 \leq i \leq n$, let $\delta^-(s)$ denote the set of inbound arcs from layer $L_{i-1}$. Likewise let $\delta^+(s)$ denote the set of outbound arcs into $L_{i+1}$. We sometimes overload notation and use $\delta^-(s)$ and $\delta^+(s)$ to also refer to the set of states in $L_{i-1}$ and $L_{i+1}$, 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 \texttt{refineLayer}(\mathcal{L}_i, [\mathcal{L}_0, \ldots, \mathcal{L}_{i-1}], w, (Y, Q, W))

\begin{algorithmic}[1]
\Require $|\mathcal{L}_i| \leq w$
\Ensure $|\mathcal{L}_i| = w \lor \text{appx}(\mathcal{L}_i) = \emptyset$
\State let $s^* = \arg \max_{s \in \text{appx}(\mathcal{L}_i)} Y(s)$
\State let $cs = \text{partition}((\text{refine}(s^*), Q))$
\If{$|cs| \leq w - |\mathcal{L}_i| + 1$}
\State $\mathcal{L}_i = \mathcal{L}_i \setminus \{s^*\} \cup \mathcal{L}_{\mathcal{L}_i + 1} \text{relax}(cs_j)$
\Else
\State let $\pi = \text{permutation}(cs)$ \hspace{1em} $\forall j, k \in 1..|cs| : j \leq k \Rightarrow W(s_{\pi_j}) \leq W(s_{\pi_k})$
\State $\mathcal{L}_i = \mathcal{L}_i \setminus \{s^*\} \cup \bigcup_{j=1}^{\left|cs\right|} \text{relax}(cs_{\pi_j}) \cup \text{relax}(\bigcup_{j=w-|\mathcal{L}_i|+1}^{\left|cs\right|} cs_{\pi_j})$
\EndIf
\end{algorithmic}

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 ($\text{appx}$, $\text{refine}$, $\text{partition}$, and $\text{permutation}$) 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 $\text{refineLayer}$ 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 $\text{appx}(\mathcal{L}_i)$ 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:

\begin{definition}
A state priority function $Y : S \rightarrow \mathbb{Z}$ takes as input state $s = (P_0, \ldots, P_{k-1})$ and returns an integer value representing its priority where the larger is the more preferable.
\end{definition}

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:

\begin{example}[Rank heuristics]
Let $Y(s) = -\text{rank}(s)$ be the heuristic to first select the oldest states inserted in the layer. Likewise, one can define $Y(s) = \text{rank}(s)$ to first select the nodes that were most recently inserted in the layer.
\end{example}

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 \( \text{AMONG}(X, l, u, \Sigma) \) 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 \( \Sigma \) on any path \( s^\downarrow \) to \( s^\downarrow \). 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 \( \Sigma \) 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:

\[
A(s) = \{ p_j \xrightarrow{\ell_j} s \mid p_j \in L_{i-1} \land \ell_j \in D(x_i) \}
\]

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

\[
\delta^+(s') = \{ s' \xrightarrow{\ell_j} c_j \mid s \xrightarrow{\ell_j} c_j \in \delta^+(s) \land E_l(s', c_j, i, \ell_j) \}
\]

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 \( T^\downarrow \):

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

Note how \( \text{relax}(K(s^*)) = s^* \) since \( K(s^*) \) is none other than the multiset of states that
would yield \( s^* \) if merged. The \( \text{refine}(s^*) \) function in Algorithm 1 (line 3) is responsible
for producing the multiset \( K(s^*) \). With unbounded width, one could retain the \( \text{unique} 
\) states in \( K(s^*) \) and add all of them into \( L_i \setminus \{s^*\} \) to upgrade \( s^* \). Otherwise, we need to
group together states in \( K(s^*) \) to be merged. The generic \( \text{partition} \) 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,
\( \text{partition}(K(s^*), Q) \) 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 \( \text{relax} \) 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)
\( w - |L_i| \) collapsed states for inclusion and merges the remaining ones into a single state.

To formalize the description above, let us adopt the following definitions:
Heuristics for MDD Propagation in Haddock

Definition 8 (Equivalence class). A state equivalence function takes the form \( Q : S \times S \rightarrow B \).

It takes as input states \( a = (A_0, \ldots, A_{k-1}) \) and \( b = (B_0, \ldots, B_{k-1}) \) 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 \( a = (A_0, \ldots, A_{k-1}) \) and \( b = (B_1, \ldots, B_{k-1}) \) 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 \( \text{AMONG}(x, l, u, \Sigma) \) 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 \( \Sigma \) are compared to \( l \) and \( u \). Given two states \( a, b \in K(s) \),

\[
\begin{align*}
& a = T^\downarrow(p_a, s, i, \ell_a) \quad \text{and} \quad b = T^\downarrow(p_b, s, i, \ell_b).
\end{align*}
\]

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

\[
Q(a, b) = ((l - (L^\downarrow(a) + L^\uparrow(a)) > t) \land (u - (U^\downarrow(b) + U^\uparrow(b)) > t))
\]

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 \( W : S \rightarrow \mathbb{Z} \).

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 \( W(s) = -\max_{p \in \delta^-(s)} \text{rank}(p) \) 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 \((Y, Q, W)\) 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:

\textbf{Definition 14 (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 \(M_c = \langle X, P, s_\bot, s_\top, T^\downarrow, T^\uparrow, U, E_t, E_s, R, H = \langle Y, Q, W \rangle \rangle\) where \(P\) is the set of
properties used to model states, \(s_\bot\) 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.

\subsection{Direct Composition}

Consider two MDD languages \(M_1\) and \(M_2\) 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 \(M_1 \land M_2\) denote the
composition of \(M_1\) and \(M_2\) and associate to it a heuristic bundle \(H_{M_1 \land M_2}\) defined as:

\textbf{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 \(H_{M_1 \land M_2} = \langle Y_{c_1} + Y_{c_2}, Q_{c_1} \land Q_{c_2}, W_{c_1} + W_{c_2} \rangle\) denote the heuristic bundle of the composition.

\subsection{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:

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

To understand how the portfolio is leveraged, consider the fixpoint algorithm used within
an MDD propagator for the conjunction of \(m\) constraints \(\land_{i=1}^m c_i\) 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 \(L_{n-1}\) to \(L_0\) 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 \(L_1\) to \(L_n\). 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($\mathcal{M}_{c_1 \land \ldots \land c_m}, [x_1, \ldots, x_n], \text{width}, \text{reboot}, \text{maxRef}$)

1: let $HP = [\langle Y_1, Q_1, W_1 \rangle, \ldots, \langle Y_k, Q_k, W_k \rangle]$ 
2: let $\text{iter} = 0$
3: repeat
4:  $\text{changed} = \text{computeUp}(\mathcal{M}_{c_1 \land \ldots \land c_m})$
5:  $\text{changed} = \text{computeDown}(\mathcal{M}_{c_1 \land \ldots \land c_m}, \text{width}, HP, \text{iter}, \text{reboot}, \text{maxRef}) \lor \text{changed}$
6:  for $i \in 1..n$ do
7:    trimVariable($x_i$)
8:  $\text{iter} = \text{iter} + 1$
9: until $\neg \text{changed}$

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

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

4.3.3 Refinement Portfolio Options

Different choices for $Q$ are possible. One could use (for a given constraint $c$) either an approximation $\tilde{Q}$ or pure state equality $Q$. Alternatively, both can be used in a portfolio $\langle Y, \tilde{Q}, W \rangle, \langle Y, Q, W \rangle$ 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 $\pi$. 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 $\tilde{Q}_{\pi_i}$ 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 $\tilde{Q}$ before $\overline{Q}$.

5 Layer Processing

5.1 Reboot Distance

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

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

HADDOK 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 HADDOK propagator. Specifically, we consider the following experiments:

Experiment 1 Investigate the impact of the \( Y \) and \( W \) heuristics.

Experiment 2 Explore the merits of \( Q \), \( \overline{Q} \), and a portfolio using first \( \overline{Q} \), then \( Q \).
Table 1: CPU time (seconds) to obtain all solutions for Nurse Rostering using \( HP = ([Y,Q,W], [Y,Q,W]) \) for different \( Y \) (columns) and \( W \) (rows) heuristics.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Width 16</th>
<th>Width 32</th>
<th>Width 64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HH</td>
<td>LR</td>
<td>HD</td>
</tr>
<tr>
<td>C-I</td>
<td>MA</td>
<td>1.9</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td>LA</td>
<td>5.4</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>MaxPI↓</td>
<td>2.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>MaxPI↑</td>
<td>2.2</td>
<td>4.3</td>
</tr>
<tr>
<td>C-II</td>
<td>MA</td>
<td>1.7</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>LA</td>
<td>19.1</td>
<td>14.2</td>
</tr>
<tr>
<td></td>
<td>MaxPI↓</td>
<td>8.2</td>
<td>9.9</td>
</tr>
<tr>
<td></td>
<td>MaxPI↑</td>
<td>8.7</td>
<td>15.7</td>
</tr>
<tr>
<td>C-III</td>
<td>MA</td>
<td>21.7</td>
<td>26.8</td>
</tr>
<tr>
<td></td>
<td>LA</td>
<td>17.7</td>
<td>27.7</td>
</tr>
<tr>
<td></td>
<td>MaxPI↓</td>
<td>15.5</td>
<td>21.5</td>
</tr>
</tbody>
</table>

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 \([Y,Q,W], [Y,Q,W]\). 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 \( Y(s) = \text{rank}(s) \) to select the most recent state first.
- **LR** Define \( Y(s) = -\text{rank}(s) \) 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↓** \( W(s) = -\min_{p \in \delta(s)} \text{rank}(p) \) ranks nodes with decreasing age of oldest parent.
- **MinPI↑** \( W(s) = \min_{p \in \delta(s)} \text{rank}(p) \) ranks nodes with increasing age of oldest parent.
- **MaxPI↓** \( W(s) = -\max_{p \in \delta(s)} \text{rank}(p) \) ranks nodes with decreasing age of youngest parent.
- **MaxPI↑** \( W(s) = \max_{p \in \delta(s)} \text{rank}(p) \) 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 \( Q \) vs. \( \tilde{Q} \). Consider the role of the two equivalence heuristics. Figure 4 graphs the shortest time and least number of backtracks when \( Q \) is used alone, \( \tilde{Q} \) is used alone, or as a portfolio \([Q,\tilde{Q}]\). At higher widths, the heuristic bundle with \( Q \) stagnates...
since the approximate equivalence prevents it from making full use of the width. The bundle using $Q$ 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 $Q$ 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: $\text{minW}$, upper bounding the number of work days: $\text{maxW}$ and restricting the number of work days to 4 or 5 in any given week: $\text{resW}$) it is tempting to rely on 3 constraint groups and use a portfolio based on the three bundles of heuristics $\{H(\text{minW}), H(\text{maxW}), H(\text{resW})\}$. To simplify, we test three portfolios: $\text{minW}$ First ($[H(\text{minW}), H(\text{maxW} \land \text{resW})]$), $\text{maxW}$ First ($[H(\text{maxW}), H(\text{minW} \land \text{resW})]$), and $\text{resW}$ First ($[H(\text{resW}), H(\text{minW} \land \text{maxW})]$). Figure 5 shows the results while using $\langle LR, Q, \text{MinPI} \rangle$ for each bundle; the results are quite spread out. The best performance, on all of C-I, C-II, and C-III, occurs whenever $\text{resW}$ is the first entry in the portfolio, giving it the first opportunity to drive refinements.

The characteristics of constraints in $\text{resW}$ 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 $\text{resW}$ constraint groups are always the smallest. Last, $\text{resW}$ 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
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 \( \geq 2 \)) of variables sampled from \( \{x_{k-1}+1, \ldots, x_{k-1}+s_i\} \) where each variable has a probability \( p_i \) of being included. Three instances (available online at \url{http://hidden.url.domain}) were created from \((50, 7, [(3, 1, 1), (6, 6, 1), (10, 1, 3), (8, 5, 6), (20, 7, 2)]\).

Performance is measured with time and backtracks to prove infeasibility. Figure 6 shows the performance using a heuristic bundle of \( \langle HR, QR, MinPI \rangle \) 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 \texttt{computeDown} during the search, that is, reboots that were not cut short. The gains occur when around 80 – 90%. By the time \( \text{reboot} = 7 \), 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 \( L_i \) is as follows. As usual, let \( \text{vars}(c) \) denote the set of variables appearing in \( c \) and \( \text{csf}(x) \) be the set of constraints mentioning variable \( x \). Let \( L(x) \) be the layer of variable \( x \). Then,

\[
\text{related}(x_i) = \bigcup_{c \in \text{csf}(x_i) \mid |\text{vars}(c)| \leq \frac{1}{n}} \text{vars}(c)
\]

in \( \text{reboot}(i) = \min_{y \in \text{related}(x_i)} \text{index}(L(y)) \) 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...
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↓ and MaxPI↓ 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 cornerstone 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.
Heuristics for MDD Propagation in Haddock

References


