Carnegie Mellon University
Research Showcase @ CMU

Tepper School of Business

2010

A Systematic Approach to MDD-Based Constraint Programming

Samid Hoda
Carnegie Mellon University

Willem-Jan van Hoeve
Carnegie Mellon University, vanhoeve@andrew.cmu.edu

John N. Hooker
Carnegie Mellon University, john@hooker.tepper.cmu.edu

Follow this and additional works at: http://repository.cmu.edu/tepper

Published In
D. Cohen (Ed.): CP 2010,, LNCS 6308, 266-280.

This Article is brought to you for free and open access by Research Showcase @ CMU. It has been accepted for inclusion in Tepper School of Business by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

A Systematic Approach to MDD-Based Constraint Programming

Samid Hoda, Willem-Jan van Hoeve, and J. N. Hooker

Tepper School of Business, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, U.S.A.
shoda,vanhoeve@andrew.cmu.edu, john@hooker.tepper.cmu.edu

Abstract. Fixed-width MDDs were introduced recently as a more refined alternative for the domain store to represent partial solutions to CSPs. In this work, we present a systematic approach to MDD-based constraint programming. First, we introduce a generic scheme for constraint propagation in MDDs. We show that all previously known propagation algorithms for MDDs can be expressed using this scheme. Moreover, we use the scheme to produce algorithms for a number of other constraints, including `Among`, `Element`, and unary resource constraints. Finally, we discuss an implementation of our MDD-based CP solver, and provide experimental evidence of the benefits of MDD-based constraint programming.

1 Introduction

The domain store is a fundamental tool for constraint programming (CP), because it propagates the results of individual constraint processing. It allows the reduced domains obtained for one constraint to be passed to the next constraint for further filtering. A weakness of the domain store, however, is that it transmits a limited amount of information. It accounts for no interaction among the variables, because any solution in the Cartesian product of the current domains is consistent with it. This restricts the ability of the domain store to pool the results of processing individual constraints and provide a global view of the problem.

To address this shortcoming, Andersen, Hadzic, Hooker, and Tiedemann [1] proposed replacing the domain store with a richer data structure, namely a multivalued decision diagram (MDD). In their approach, domain filtering algorithms are replaced or augmented by algorithms that refine and update the MDD to reflect each constraint. It was found that MDD-based propagation can lead to substantial speedups in the solution of multiple `AllDifferent` constraints. The idea was extended to equality constraints by Hadzic et al. [4]. A unified node-splitting scheme for refining the MDD was proposed by Hadzic et al. [3] and applied to certain configuration problems. For this reason, we will mainly focus on filtering algorithms in this work.

MDDs have been applied before in CP. For example, in [6] and [2] MDDs are applied to perform inferences (domain filtering) based on individual constraints.

In [5], this is taken one step further, by passing structural information from one constraint to the next. The key difference is that in these approaches, an MDD is built and maintained for each individual constraint, whereas in MDD-based constraint programming, the MDD *is* the information that is passed from one constraint to the next. In other words, multiple constraints process the same MDD, instead of each constraint processing its individual MDD.

The contributions of this work are threefold. First, we introduce a systematic scheme for constraint propagation in MDDs, and we show that all previously proposed filtering algorithms for MDD-based CP can be viewed as instantiations of our scheme. Second, we apply our scheme to introduce new filtering algorithms for other constraints; we present such algorithms for the `Among`, `Element`, and unary resource constraints as an illustration of the versatility of the approach. Third, we present computational results for the first pure MDD-based CP solver, showing that *i)* this approach can scale up to realistic problem sizes, and *ii)* enormous savings (in terms of time as well as search tree size) can be realized when compared to solvers relying on the traditional domain store.

The remainder of the paper is organized as follows. In Section 2 we provide the necessary background on MDDs and MDD-based constraint programming. In Section 3 we present and discuss a systematic scheme for constraint propagation in MDDs. We apply our scheme to a variety of constraints in Section 4. In Section 5 we provide a brief description of our MDD-based constraint programming system. Experimental results using this system are reported in Section 6. Finally, we conclude in Section 7.

2 MDDs and MDD-Based Constraint Solving

In this work, an *ordered Multivalued Decision Diagram (MDD)* is a directed acyclic graph whose nodes are partitioned into n (possibly empty) subsets or *layers* $L_1, \ldots, L_{n+1}$, where the layers $L_1, \ldots, L_n$ correspond respectively to variables $x_1, \ldots, x_n$. L_1 contains a single *top* node $\mathbf{T}$, and L_{n+1} contains two *bottom* nodes $\mathbf{0}$ and $\mathbf{1}$. The *width* of the MDD is the maximum number of nodes in a layer, or $\max_{i=1}^{n}\{|L_i|\}$. In MDD-based CP, the MDDs typically have a given fixed maximum width.

All edges of the MDD are directed from an upper to a lower layer; that is, from a node in some L_i to a node in some L_j with $i < j$. For our purposes it is convenient to assume (without loss of generality) that each edge connects two adjacent layers. Let $L(s)$ denote the layer of the node s, and $\mathrm{var}(s)$ the variable associated with $L(s)$. Each edge out of layer i is labeled with an element of the domain $D(x_i)$ of x_i. The set $E(s,t)$ of edges from node s to node t may contain multiple edges, and we denote each with its label. Let $E^{in}(s)$ denote the set of edges coming into s, and $E^{out}(s)$ the set of edges leaving s. For an edge e, $\mathrm{tail}(e)$ is the tail of e and $\mathrm{head}(e)$ the head.

An edge with label v leaving a node in layer i represents an assignment $x_i = v$. Each path in the MDD from $\mathbf{T}$ to $\mathbf{0}$ or $\mathbf{1}$ can be denoted by the edge labels $v_1, \ldots, v_n$ on the path and is identified with the assignment $(x_1, \ldots, x_n) =$

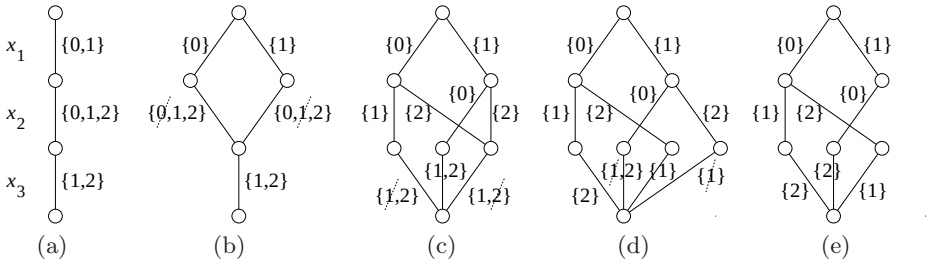

Fig. 1. Refining and filtering an MDD of width one (a) for $x_1 \neq x_2$ (b), $x_2 \neq x_3$ (c), and $x_1 \neq x_3$ (d), yielding the MDD in (e). Dashed lines mark filtered values.

$(v_1, \ldots, v_n)$. For our purposes, it is convenient to generate only the portion of an MDD that contains paths from $\mathbf{T}$ to $\mathbf{1}$. A path $v_1, \ldots, v_n$ is *feasible* for a given constraint C if setting $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$ satisfies C. Constraint C is feasible on an MDD if the MDD contains a feasible path for C.

A constraint C is called *MDD consistent* on a given MDD if every edge of the MDD lies on some feasible path. Thus MDD consistency is achieved when all redundant edges (i.e., edges on no feasible path) have been removed. Domain consistency for C is equivalent to MDD consistency on an MDD of width one that represents the variable domains. That is, it is equivalent to MDD consistency on an MDD in which each layer L_i contains a single node s_i, and $E(s_i, s_{i+1}) = D(x_i)$ for $i = 1, \ldots, n$.

Typically, MDD-based constraint programming starts with simple MDD (of width one) that permits all solutions represented by the Cartesian product of the domains. This MDD is then *refined* each time a constraint is processed. Refinement is accomplished by adding some nodes and edges to the MDD so as to exclude solutions that violate the constraint. Example 1 below gives an illustration of this (see also Figure 1).

The basic operation of refinement is *node-splitting*, in which the edges entering a given node are partitioned into equivalence classes, and ideally the node is split into one copy for each equivalence class. The set of outgoing edges for each copy is the same as the set of outgoing edges of the original node. We note that determining the equivalence classes may be costly to compute in practice, in which case an approximation of equivalence is used. We take care that the width of the MDD (maximum number of nodes in a layer) remains within a fixed bound. When splitting a node we merge equivalence classes when necessary in order to respect this restriction. The resulting MDD is a relaxation in the sense that it may fail to exclude all assignments that violate the constraint, but it is a much stronger relaxation than a domain store. A principled approach to node refinement in MDDs is introduced by Hadzic et al. [3].

We also update the MDD by deleting infeasible edges, an operation that generalizes conventional domain filtering. We will refer to this operation as *MDD filtering*. This can lead to further reduction of the MDD, if after the removal of the edge some other edges no longer have a path to $\mathbf{1}$ or can no longer be

reached by a path from the root. An MDD-based constraint solver is based on *propagation* and *search* just as traditional CSP solvers, but the domain filtering process at each node of the search tree is replaced (or supplemented) by an MDD refinement and filtering process.

Example 1. Consider a CSP with variables $x_1 \in \{0, 1\}$, $x_2 \in \{0, 1, 2\}$, and $x_3 \in \{1, 2\}$, and constraints $x_1 \neq x_2$, $x_2 \neq x_3$, and $x_1 \neq x_3$. All domain values are domain consistent (even if we were to apply the `AllDifferent` propagator on the conjunction of the constraints), and the domain store defines the relaxation $\{0, 1\} \times \{0, 1, 2\} \times \{1, 2\}$, which includes infeasible solutions such as $(1, 1, 1)$.

The MDD-based approach starts with the MDD of width one in Figure 1(a), in which multiple arcs are represented by a set of corresponding domain values for clarity. We refine and filter each constraint separately. Starting with $x_1 \neq x_2$, we refine the MDD by splitting the node at layer 2, resulting in Figure 1(b). This allows us to filter two domain values, based on $x_1 \neq x_2$, as indicated in the figure. In Figure 1(c) and (d) we refine and filter the MDD for the constraints $x_2 \neq x_3$ and $x_1 \neq x_3$ respectively, until we reach the MDD in Figure 1(e). This MDD represents all three solutions to the problem, and provides a much tighter relaxation than the domain store.

3 A Systematic Scheme for MDD Propagation

In the literature, MDD propagation algorithms (in the sense of Section 2) have been proposed for the following three constraint types: (one-sided) inequality constraints [1], `AllDifferent` [1], and equality constraints [3]. The reasoning used for designing propagation algorithms for each of these constraints seemed to be ad-hoc. In this section we will present and analyze a systematic scheme for designing MDD propagation algorithms. In the following section we use this procedure to express the existing MDD propagation algorithms and introduce new algorithms for the `Among`, `Element`, and unary resource constraints.

3.1 The General Scheme

Our scheme is based on the idea of 'local information' $I(s)$ stored for each constraint at each node s. We will show that all MDD propagation schemes so far introduced, and others as well, can be viewed as based on local information. The precise nature of the local information depends on the propagation scheme, which is characterized in part by what kind of local information is required to apply it.

More precisely, the decision as to whether to delete an edge in $E(s, t)$ is based solely on $I(s)$ and $I(t)$—that is, on local information stored at either end of the edge. Furthermore, the local information can be accumulated by a single top-down pass and a single bottom-up pass through the MDD.

It is convenient to regard $I(s)$ as a pair $(I^{\downarrow}(s), I^{\uparrow}(s))$ consisting of the information $I^{\downarrow}(s)$ accumulated during the top-down pass, and the information $I^{\uparrow}(s)$

accumulated during the bottom-up pass. $I^{\downarrow}(s)$ and $I^{\uparrow}(s)$ can take several forms, such as a set of domain values or parameters related to the constraint, or a tuple of such sets. What is common to all the schemes is that $I^{\downarrow}(s)$ is computed solely on the basis of local information at the opposite end of edges coming into s, and $I^{\uparrow}(s)$ on the basis of local information at the opposite end of edges leaving s.

Formally, we introduce an operation $\otimes$ that processes information when traversing an edge during a top-down or bottom-up pass. When traversing an edge $e \in E(s,t)$ during the top-down pass, the information $I^{\downarrow}(s)$ at node s is combined with edge e to obtain updated information $I^{\downarrow}(s) \otimes e$. We view this updated information as an object I having the same form as $I^{\downarrow}(s)$. When several edges enter node t, we use an operation $\oplus$ to combine the information obtained by traversing the incoming edges. The top-down information at t is therefore

$$I^{\downarrow}(t) = \bigoplus_{e \in E^{in}(t)} I^{\downarrow}(\text{tail}(e)) \otimes e$$

Similarly, the bottom-up information at s is

$$I^{\uparrow}(s) = \bigoplus_{e \in E^{out}(s)} I^{\uparrow}(\text{head}(e)) \otimes e$$

Several examples of this scheme appear in the following sections.

Since a top-down (bottom-up) pass of the MDD visits each edge exactly once, the passes themselves involve an amount of work that is linear in the size of the MDD (modulo the work required to compute $\oplus$ and $\otimes$ at each node).

The operators $\otimes$ and $\oplus$ can be implemented as high-level macros that are instantiated differently for each constraint type. Our MDD-based CP solver follows this idea very closely.

3.2 MDD Consistency

The scheme above can sometimes achieve MDD consistency in polynomial time. In particular, if it can determine in polytime whether any particular assignment $x_j = v$ is consistent with the MDD, then it achieves MDD consistency in polytime due to the following theorem.

Theorem 1. *Suppose that the feasibility of $x_j = v$ for a given constraint C on a given MDD M can be determined in $\mathcal{O}(f(M))$ time and space for any variable x_j in C and any $v \in D(x_j)$. Then we can achieve MDD consistency for C in time and space at most $\mathcal{O}(\text{poly}(M)f(M))$.*

The proof is a straightforward shaving argument. For each edge e of M we consider the MDD M_e that consists of all the **T**–to–**1** paths in M containing e. Then e can be removed from M if and only if $x_j = e$ is inconsistent with C and M_e, where $j = L(\text{tail}(e))$. This can be determined in time and space at most $\mathcal{O}(f(M_e)) \leq \mathcal{O}(f(M))$. By repeating this operation $\text{poly}(M)$ times (i.e., on each edge of M) we obtain the theorem.

To establish MDD consistency, the goal is to efficiently compute information that is strong enough to apply Theorem 1. In the sequel, we will see that this can be done for inequality constraints and `Among` constraints in polynomial time, and in pseudo-polynomial time for two-sided inequality constraints. Furthermore, we have the following result.

Corollary 1. *For binary constraints that are given in extension, our scheme can be applied to achieve MDD consistency in polynomial time (in the size of the constraint and the MDD).*

To see how this is accomplished, suppose C is a binary constraint containing variables x_i, x_j for $i < j$ (the argument for $j < i$ is similar). We let $I^{\downarrow}(s)$ contain all the values assigned to x_i in some path from $\mathbf{T}$ to s, and $I^{\uparrow}(t)$ contain all the values assigned to x_j in some path from s to $\mathbf{1}$. Then we can delete edge $e \in E(s,t)$ from M if and only if (a) $L(s) = j$ and $(x_i, x_j) = (v, e)$ satisfies C for no $v \in I^{\downarrow}(s)$, or (b) $L(s) = i$ and $(x_i, x_j) = (e, v)$ satisfies C for no $v \in I^{\uparrow}(t)$. Because $|I^{\downarrow}(s)|$ and $|I^{\uparrow}(t)|$ are polynomial in the size of M, we can perform this check in time that is polynomial in the size of M and C. Also, we can compute the local information by defining the $\oplus$ operator

$$I^{\downarrow}(s) \otimes e = \begin{cases} \{e\} & \text{if } L(s) = i \\ I^{\downarrow}(s) & \text{otherwise} \end{cases}$$

$$I^{\uparrow}(t) \otimes e = \begin{cases} \{e\} & \text{if } L(s) = j \\ I^{\uparrow}(t) & \text{otherwise} \end{cases}$$

with $I^{\downarrow}(\mathbf{T}) = I^{\uparrow}(\mathbf{1}) = \emptyset$, and the $\oplus$ operator

$$I \oplus I' = I \cup I'$$

The top-down and bottom-up passes clearly require time that is polynomial in the size of M. The corollary then follows from Theorem 1.

4 Specialized Propagators

We now present several MDD propagation algorithms that rely on local information obtained as above. The filtering may not be as strong as for a conventional domain store, in the sense that when specialized to an MDD of width one, it may not remove as many values as a conventional filter would. However, a 'weak' filtering algorithm can be very effective when applied to the richer information content of an MDD.

If one prefers not to design a filter specifically for MDDs, there is also the option of using a conventional domain filter by adapting it to MDDs. This can be done in a generic fashion that turns out to be yet another application of the above scheme. Section 4.8 explains how this is done.

4.1 Equality and Not-Equal Constraints

We first illustrate MDD propagation of the constraints $x_i = x_j$ and $x_i \neq x_j$. Because these are binary constraints, by Corollary 1 the scheme presented in Section 3.2 achieves MDD consistency in polytime. If we compute $I^{\downarrow}$ as described in that section, we can achieve MDD consistency for $x_i = x_j$ by deleting an edge $e \in E(s,t)$ whenever (a) $L(s) = j$ and $e \notin I^{\downarrow}(s)$ or (b) $L(s) = i$ and $e \notin I^{\uparrow}(t)$. We can achieve MDD consistency for $x_i \neq x_j$ by deleting e whenever (a) $L(s) = j$ and $I^{\downarrow}(s) = \{e\}$ or (b) $L(s) = i$ and $I^{\uparrow}(t) = \{e\}$.

We note that this scheme generalizes directly to propagating $f_i(x_i) = f_j(x_j)$ and $f_i(x_i) \neq f_j(x_j)$ for functions f_i and f_j. The scheme can also be applied to constraints $x_i < x_j$. However, in this case we only need to maintain bound information instead of sets of domain values, which leads to an even more efficient implementation.

4.2 Propagating Linear Inequalities

We next focus on the filtering algorithm for general inequalities, as proposed in [1]. That is, we want to propagate an inequality over a separable function of the form:

$$\sum_{j \in J} f_j(x_j) \leq b \tag{1}$$

We can propagate such constraint on an MDD by performing shortest-path computations.

Recall that each edge $e \in E(s,t)$ is identified with a value assigned to $\mathrm{var}(s)$. Supposing that $\mathrm{var}(s) = x_j$, we let the length of edge e be $f_j(e)$ when $j \in J$ and zero otherwise. Thus the length of a $\mathbf{T}$–to–$\mathbf{1}$ path is the left-hand side of (1).

We let $I^{\downarrow}(s)$ be the length of a shortest path from $\mathbf{T}$ to s, and $I^{\uparrow}(s)$ the length of a shortest path from s to $\mathbf{1}$. Then we delete an edge $e \in E(s,t)$ when $L(s) \in J$ and every path through e is longer than b; that is,

$$I^{\downarrow}(s) + f_{L(s)}(e) + I^{\uparrow}(t) > b$$

It is easy to compute local information in the form of shortest path lengths, because we can define for $e \in E(s,t)$

$$I^{\downarrow}(s) \otimes e = \begin{cases} I^{\downarrow}(s) + f_{L(s)}(e) & \text{if } L(s) \in J \\ I^{\downarrow}(s) & \text{otherwise} \end{cases}$$

$$I^{\uparrow}(t) \otimes e = \begin{cases} I^{\uparrow}(t) + f_{L(s)}(e) & \text{if } L(s) \in J \\ I^{\uparrow}(t) & \text{otherwise} \end{cases}$$

with $I^{\downarrow}(\mathbf{T}) = I^{\uparrow}(\mathbf{1}) = 0$. We also define

$$I \oplus I' = \min\{I, I'\}$$

This inequality propagator achieves MDD consistency as an edge e is always removed unless there exists a feasible solution to the inequality that supports it [1].

4.3 Propagating Two-sided Inequality Constraints

In this section, we present a generalization of the equality propagator described by Hadzic et al. [3]. It extends the inequality propagator of Section 4.2, but now we store all path lengths instead of only the shortest and longest paths.

Suppose we are given an inequality constraint $l \leq \sum_{j \in J} f_j(x_j) \leq u$, where l and u are numbers such that $l \leq u$. Let $I^{\downarrow}(s)$ be the set of all path lengths from $\mathbf{T}$ to s, and $I^{\uparrow}(s)$ the set of all path lengths from s to $\mathbf{1}$. We delete an edge $e \in E(s,t)$ when

$$v + e + v' \notin [l, u], \text{ for all } v \in I^{\downarrow}(s), v' \in I^{\uparrow}(t)$$

The local information is computed by defining for $e \in E(s,t)$

$$I^{\downarrow}(s) \otimes e = \begin{cases} \{v + e \mid v \in I^{\downarrow}(s)\} & \text{if } L(s) \in J \\ I^{\downarrow}(s) & \text{otherwise} \end{cases}$$

$$I^{\uparrow}(t) \otimes e = \begin{cases} \{v + e \mid v \in I^{\uparrow}(t)\} & \text{if } L(s) \in J \\ I^{\uparrow}(t) & \text{otherwise} \end{cases}$$

with $I^{\downarrow}(\mathbf{T}) = I^{\uparrow}(\mathbf{1}) = \emptyset$. Also $I \oplus I' = I \cup I'$.

When we delete an edge, the information stored at all predecessors and successors becomes 'stale', and the information for these nodes must be recomputed to guarantee MDD consistency. However, we will achieve MDD consistency if, every time we delete an edge, we update the node information for all predecessors and successors and repeat this filtering and updating until we reach a fixed point. This follows because the filtering condition above is both necessary and sufficient for an edge to be supported by a feasible solution. Observing that the information can be computed in pseudo-polynomial time, by Theorem 1 this algorithm runs in pseudo-polynomial time (see also [3]).

4.4 Propagating the AllDifferent Constraint

The constraint `AllDifferent`$(x_i, i \in J)$ requires that the variables x_i for $i \in J$ take pairwise distinct values. We can frame the `AllDifferent` propagator presented in Andersen et al. [1] in terms of our scheme. Let $I^{\downarrow}(s) = (A^{\downarrow}(s), S^{\downarrow}(s))$ and $I^{\uparrow}(s) = (A^{\uparrow}(s), S^{\uparrow}(s))$. Here $A^{\downarrow}(s)$ is the set of values that appear on all paths from $\mathbf{T}$ to s—that is, the set of values v such that on all $\mathbf{T}$–s paths, $x_j = v$ for some $j \in J$. $S^{\downarrow}(s)$ is the set of values v that appear on some $\mathbf{T}$–s path. $A^{\uparrow}(s)$ and $S^{\uparrow}(s)$ are defined similarly.

We can delete edge $e \in E(s,t)$ for $L(s) \in J$ when $e \in A^{\downarrow}(s) \cup A^{\uparrow}(t)$. We can also delete e when the variables above s, or the variables below s, form a Hall set. To make this precise, let $X_s^{\downarrow} = \{x_j \mid j \in J,\ j < L(s)\}$ be the set of variables in the `AllDifferent` constraint above s, and $X_t^{\uparrow} = \{x_j \mid j \in J,\ j > L(s)\}$ the set of variables below s. Then if $|X^{\downarrow}(s)| = |S^{\downarrow}(s)|$ (that is, $X_s^{\downarrow}$ is a Hall set), the

values in $S^{\downarrow}(s)$ cannot be assigned to any variable not in $X^{\downarrow}(s)$. So we delete e if $e \in S^{\downarrow}(s)$. Similarly, if $X^{\uparrow}(t)$ is a Hall set, we delete e if $e \in S^{\uparrow}(t)$.

Finally, we compute the local information by defining for $e \in E(s,t)$

$$I^{\downarrow}(s) \otimes e = \begin{cases} I^{\downarrow}(s) \cup (\{e\}, \{e\}), & \text{if } L(s) \in J \\ I^{\downarrow}(s), & \text{otherwise} \end{cases}$$

$$I^{\uparrow}(t) \otimes e = \begin{cases} I^{\uparrow}(t) \cup (\{e\}, \{e\}), & \text{if } L(s) \in J \\ I^{\uparrow}(t), & \text{otherwise} \end{cases}$$

where the unions are taken componentwise and $I^{\downarrow}(\mathbf{T}) = I^{\uparrow}(\mathbf{1}) = (\emptyset, \emptyset)$. Also we define

$$I \oplus I' = (A \cap A', S \cup S').$$

4.5 Propagating the Among Constraint

The `Among` constraint restricts the number of variables that can be assigned a value from a specific subset of domain values. Formally, if $X = (x_1, \ldots, x_q)$ is a sequence of variables, S a set of domain values, and ℓ, u, q are constants with $0 \leq \ell \leq u \leq q$, then $\texttt{Among}(X, S, \ell, u)$ requires that

$$\ell \leq |\{i \in \{1, \ldots, q\} \mid v_i \in S\}| \leq u$$

We can reduce propagating $\texttt{Among}(X, S, \ell, u)$ to propagating a two-sided separable inequality constraint,

$$\ell \leq \sum_{x_i \in X} f_i(x_i) \leq u,$$

where

$$f_i(v) = \begin{cases} 1, & \text{if } v \in S \\ 0, & \text{otherwise.} \end{cases}$$

Because each $f_i(\cdot) \in \{0, 1\}$, we can compute the information in polynomial time, and by Theorem 1 MDD consistency can be achieved in polynomial time for `Among` constraints.

However, this filtering is too slow in practice. Instead, we propose to propagate bounds information instead. That is, we can use the inequality propagator for the pair of inequalities separately, and reason on the shortest and longest path lengths, as in Section 4.2.

4.6 Propagating the Element Constraint

We next consider constraints of the form $\texttt{Element}(x_i, (a_1, \ldots, a_m), x_j)$, where the a_k are constants. This means that the variable x_j must take the x_i^{th} value in the list $(a_1, \ldots, a_m)$; that is, $x_j = a_{x_i}$.

Because this is a binary constraint, we can achieve MDD consistency in polytime by defining $I^{\downarrow}(s), I^{\uparrow}(t)$ as in the proof of Corollary 1. Supposing that $i < j$, we delete an edge $e \in E(s,t)$ when (a) $L(s) = j$ and $e = a_k$ for no $k \in I^{\downarrow}(s)$, or (b) $L(s) = i$ and $a_e \notin I^{\uparrow}(t)$.

4.7 Propagating the Unary Resource Constraint

We consider unary resource constraints of the following form. We wish to schedule a set A of activities on a single resource (non-preemptively). Each activity $a \in A$ has a given release time r_a, deadline d_a, and processing time p_a. We model the problem using variables $X = (x_1, \ldots, x_{|A|})$, where $x_i = a$ implies that activity a is the ith activity to consume the resource. That is, we to find the order in which to process the activities, from which the start times can be immediately derived.

First, observe that in our representation the variables encode a permutation of A, which means that we can immediately apply the `AllDifferent` propagator from Section 4.4.

To enforce the time windows, let $I^{\downarrow}(s)$ be the earliest start time of the activity in position $L(s)$ of the sequence, given the previous activities in the sequence. Let $I^{\uparrow}(t)$ be latest completion time of the activity in position $L(t)-1$, given the subsequent activities. Then we can delete edge $e \in E(s,t)$ if the time window is too small to complete activity e; that is, if

$$\max\{I^{\downarrow}(s), r_e\} + p_e > \min\{I^{\uparrow}(t), d_e\}$$

The local information is computed

$$I^{\downarrow}(s) \otimes e = \max\left\{I^{\downarrow}(s) + p_e,\ r_e + p_e\right\}$$
$$I^{\uparrow}(t) \otimes e = \min\left\{I^{\uparrow}(t) - p_e,\ d_e - p_e\right\}$$

with $I^{\downarrow}(\mathbf{T}) = -\infty$ and $I^{\uparrow}(\mathbf{1}) = \infty$. Also $I \oplus I' = (\min\{I^{\downarrow}, I^{\downarrow\prime}\}, \max\{I^{\uparrow}, I^{\uparrow\prime}\})$.

4.8 Using Conventional Domain Filters

An existing domain filter can be adapted to MDD propagation on the basis of local information. However, the resulting propagator may not achieve MDD consistency even when it achieves domain consistency.

The adaptation goes as follows. Following Andersen et al. [1], we define the *induced domain relaxation* $D^{\times}(M)$ of an MDD M to be a tuple of domains $(D_1^{\times}(M), \ldots, D_n^{\times}(M))$ where each $D_i^{\times}(M)$ contains the values that appear on level i of M. That is,

$$D_i^{\times}(M) = \bigcup_{\substack{s,\,t \\ L(s)\,=\,i\,=\,L(t)\,-\,1}} E(s,t)$$

We can perhaps delete edges from a given level i of M by selecting a node s on level i and applying the conventional filter to the domains

$$D_1^{\times}(M_s), \ldots, D_{i-1}^{\times}(M_s), E^{out}(s), D_{i+1}^{\times}(M_s), \ldots, D_n^{\times}(M_s) \tag{2}$$

where M_s is the portion of M consisting of all paths through node s. We remove values only from the domain of x_i, that is from $E^{out}(s)$, and delete the corresponding edges from M. This can be done for each node on level i and for each level in turn.

To compute $D^{\times}(M_s)$, we can regard it as local information $I^{\downarrow}(s)$ (bottom-up information $I^{\uparrow}(s)$ is not needed). Then for $e \in E(s,t)$

$$I^{\downarrow}(s) \otimes e = I^{\downarrow}(s) \cup (\emptyset, \ldots, \emptyset, \{e\}, \emptyset, \ldots, \emptyset)$$

where $\{e\}$ is component $L(s)$ of the vector, and the union is taken componentwise. Also $I \oplus I' = I \cup I'$.

This leads to the following lemma.

Lemma 1. *The induced domain relaxation $D^{\times}(M_s)$ can be computed for all nodes s of a given MDD M in polynomial time (in the size of the MDD).*

Again following Andersen et al. [1], we can strengthen the filtering by noting which values can be deleted from the domains $D_j^{\times}(M_s)$ for $j \neq i$ when (2) is filtered. If v can be deleted from $D_j^{\times}(M_s)$, we place the nogood $x_j \neq v$ on each edge in $E^{out}(s)$. Then we move the nogoods on level i toward level j. If $j > i$, for example, we filter (2) for each node on level i and then note which nodes on level $i+1$ have the property that all incoming edges have the nogood $x_j \neq v$. These nodes propagate the nogood to all their outgoing edges, and so forth until level j is reached, where all edges with nogood $x_j \neq v$ and label v are deleted.

5 Implementation Issues

We have implemented a C++ system for MDD-Based Constraint Programming. For a detailed description of the system, we refer to [7]. We next highlight some of the most important design decisions we have made.

Even though MDDs can grow exponentially large to represent a given constraint perfectly, the basis of MDD-based constraint programming is to control the size of the MDD by specifying a maximum width k. A MDD of width one is equivalent to the conventional domain store, while increasing values of k allow the MDD to converge to a perfect representation of the solution space. An MDD on n variables therefore contains $O(nk)$ nodes. Furthermore, by aggregating the domain values corresponding to multiple (parallel) edges between two nodes, the MDD contains $O(nk^2)$ edges. Therefore, for fixed k, all bottom-up and top-down passes take linear time.

In our system, we do not propagate the constraints until a fixed point. Instead, by default we allocate one bottom-up and top-down pass to each constraint. The bottom-up pass is used to compute the information $I^{\uparrow}$. The top-down pass processes the MDD a layer at a time, in which we first compute $I^{\downarrow}$, then refine the nodes in the layer, and finally apply the filtering conditions based on $I^{\uparrow}$ and $I^{\downarrow}$.

Our outer search procedure is currently implemented using a priority queue, in which the search nodes are inserted with a specific weight. This allows to easily encode depth-first search or best-first search procedures. Each search tree node contains a copy of the MDD of its parent, together with the associated branching decision. When applying a depth-first search strategy, the total amount of

space required to store all MDDs is polynomial, namely $O(n^2k^2)$. We note that 'recomputation' strategies that are common in conventional CP systems, cannot be easily applied in this context, because the MDD may change from parent to child node, due to the node refinement procedure.

6 Experimental Results

In this section, we provide detailed experimental evidence to support the claim that MDD-based constraint programming can be a viable alternative to constraint programming based on the domain store. All the experiments are performed using a 2.33GHz Intel Xeon machine with 8GB memory, using our MDD-based CP solver. For comparison reasons, our solver applies a depth-first search, using a static lexicographic-first variable selection heuristic, and a minimum-value-first value selection heuristic. We vary the maximum width of the MDD, while keeping all other settings the same.

Multiple Among Constraints We first present experiments on problems consisting of multiple `Among` constraints. Each instance contains 50 (binary) variables, and each `Among` constraint consists of 5 variables chosen at random, from a normal distribution with a uniform-random mean (from [1..50]) and standard deviation $\sigma = 2.5$, modulo 50. As a result, for these `Among` constraints the variable indices are near-consecutive, a pattern encountered in many practical situations. Each `Among` has a fixed lower bound of 2 and upper bound of 3, specifying the number of variables that can take value 1. In our experiments we vary the number of `Among` constraints (from 5 to 200, by steps of 5) in each instance, and we generate 100 instances for each number. We note that these instances exhibit a sharp feasibility phase transition, with a corresponding hardness peak, as the number of constraints increases. We have experimented with several other parameter settings, and we note that the reported results are representative for the other parameter settings; see Hoda [7] for more details.

In Figure 2, we provide a scatter plot of the running times for width 1 versus width 4, 8 , and 16, for all instances. Note that this is a log-log plot. Points on the diagonal represent instances for which the running times, respectively number of backtracks, are equal. For points below the diagonal, width 4, 8, or 16 has a smaller search tree, respectively is faster, than width 1, and the opposite holds for points above the diagonal.

We can observe that width 4 already consistently outperforms the domain store, in some cases up to six orders of magnitude in terms of search tree size (backtracks), and up to four orders of magnitude in terms of computation time. For width 8, this behavior is even more consistent, and for width 16, all instances can be solved in under 10 seconds, while the domain store needs hundreds or thousands of seconds for several of these instances.

Nurse Rostering Instances We next conduct experiments on a set of instances inspired by nurse rostering problems, taken from [8]. The instances are of three different classes, and combine constraints on the minimum and maximum number of working days for sequences of consecutive days of given lengths.

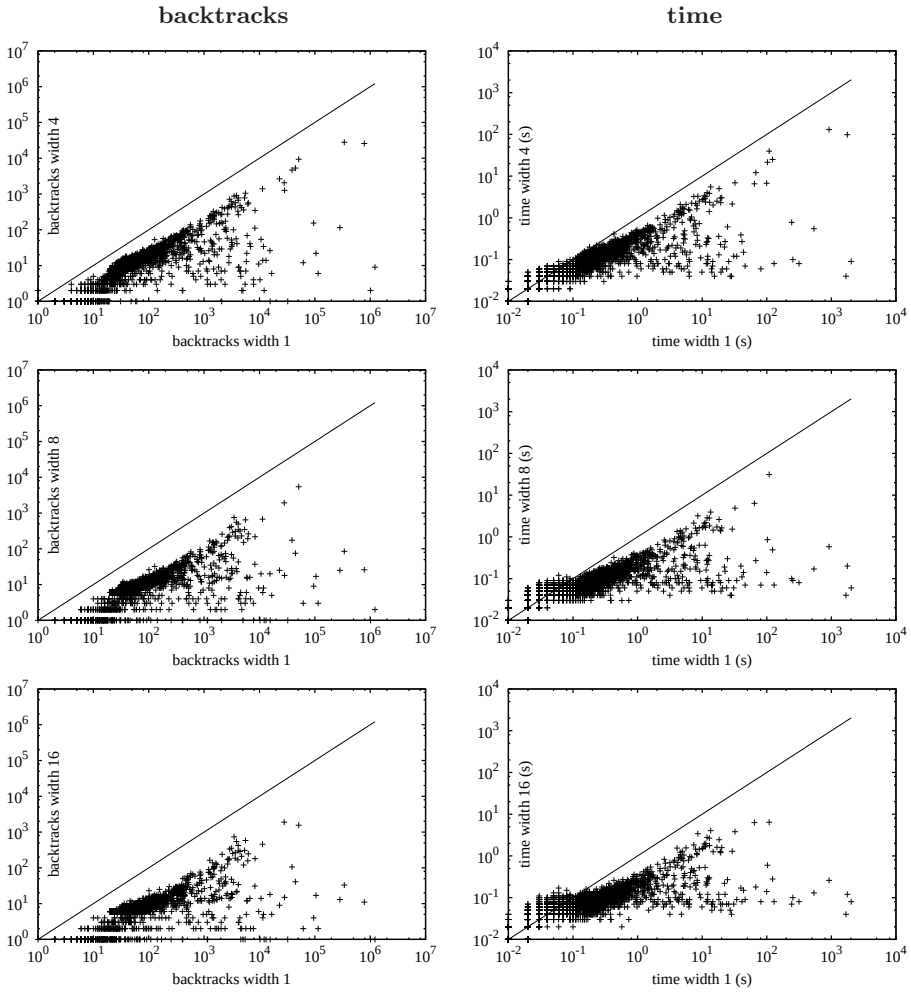

Fig. 2. Scatter plots comparing width 1 versus width 4, 8, and 16 (from top to bottom) in terms of backtracks (left) and computation time in seconds (right) on multiple `Among` problems.

That is, class *C-I* demands to work at most 6 out of each 8 consecutive days (max6/8) and at least 22 out of every 30 consecutive days (min22/30). For class *C-II* these numbers are max6/9 and min20/30, and for class *C-III* these numbers are max7/9 and min22/30. In addition, all classes require to work between 4 and 5 days per calendar week. The planning horizon ranges from 40 to 80 days.

The results are presented in Table 1. We report the total number of backtracks upon failure (BT) and computation time in seconds (CPU) needed by our MDD solver for finding a first feasible solution, using widths 1, 2, 4, 8, 16, 32, and 64. Again, the MDD of width 1 corresponds to a domain store. For all problem classes we observe a nearly monotonically decreasing sequence of backtracks

instance		width 1		width 2		width 4		width 8		width 16		width 32		width 64	
	size	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
C-I	40	61,225	55.63	22,443	28.67	8,138	12.64	1,596	3.84	6	0.07	3	0.09	2	0.10
	50	62,700	88.42	20,992	48.82	3,271	12.04	345	2.76	4	0.08	3	0.13	3	0.16
	60	111,024	196.94	38,512	117.66	3,621	19.92	610	6.89	12	0.24	8	0.29	5	0.34
	70	174,417	375.70	64,410	243.75	5,182	37.05	889	12.44	43	0.80	13	0.59	14	0.90
	80	175,175	442.29	64,969	298.74	5,025	44.63	893	15.70	46	1.17	11	0.72	12	1.01
C-II	40	179,743	173.45	60,121	79.44	17,923	32.59	3,287	7.27	4	0.07	4	0.07	5	0.11
	50	179,743	253.55	73,942	166.99	9,663	38.25	2,556	18.72	4	0.09	3	0.12	3	0.18
	60	179,743	329.72	74,332	223.13	8,761	49.66	1,572	16.82	3	0.13	3	0.18	2	0.24
	70	179,743	391.29	74,332	279.63	8,746	64.80	1,569	22.35	4	0.18	2	0.24	2	0.34
	80	179,743	459.01	74,331	339.57	8,747	80.62	1,577	28.13	3	0.24	2	0.32	2	0.45
C-III	40	91,141	84.43	29,781	38.41	5,148	9.11	4,491	9.26	680	1.23	7	0.18	6	0.13
	50	95,484	136.36	32,471	75.59	2,260	9.51	452	3.86	19	0.43	7	0.24	3	0.20
	60	95,509	173.08	32,963	102.30	2,226	13.32	467	5.47	16	0.50	6	0.28	3	0.24
	70	856,470	1,986.15	420,296	1,382.86	37,564	186.94	5,978	58.12	1,826	20.00	87	3.12	38	2.29
	80	882,640	2,391.01	423,053	1,752.07	33,379	235.17	4,236	65.05	680	14.97	55	3.27	32	2.77

Table 1. The effect of the MDD width on time in seconds (CPU) and backtracks (BT) when finding one feasible solution on nurse rostering instances.

and solution time as we increase the width up to 64. Furthermore, the rate of decrease appears to be exponential in many cases, and again higher widths can yield savings of several orders of magnitude. A typical result (the instance *C-III* on 60 days) shows that where an MDD of width 1 requires 95,509 backtracks and 173.08 seconds of computation time, an MDD of width 32 only requires 6 backtracks and 0.28 seconds of computation time to find a first feasible solution.

7 Conclusion

We have introduced a generic scheme for propagating constraints in MDDs, and showed that all existing MDD-based constraint propagators are instantiations of this scheme. Furthermore, our scheme can be applied to systematically design propagators for other constraints, and we have illustrated this explicitly for the `Among`, `Element`, and unary resource constraints. We further provide experimental results for the first pure MDD-based constraint programming solver, showing that MDD-based constraint programming can yield savings of several orders of magnitude in time and search space as compared to the conventional domain store.

Bibliography

[1] 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.

[2] K. Cheng and R. Yap. Maintaining Generalized Arc Consistency on Ad Hoc r-Ary Constraints. In *Proceedings of CP*, volume 5202 of *LNCS*, pages 509–523. Springer, 2008.

[3] T. Hadzic, J.N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate Compilation of Constraints into Multivalued Decision Diagrams. In *Proceedings of CP*, volume 5202 of *LNCS*, pages 448–462. Springer, 2008.

[4] T. Hadzic, J.N. Hooker, and P. Tiedemann. Propagating Separable Equalities in an MDD Store. In *Proceedings of CPAIOR*, volume 5015 of *LNCS*, pages 318–322. Springer, 2008.

[5] T. Hadzic, E. O'Mahony, B. O'Sullivan, and M. Sellmann. Enhanced Inference for the Market Split Problem. In *Proceedings of ICTAI*, pages 716–723. IEEE, 2009.

[6] P. Hawkins, V. Lagoon, and P.J. Stuckey. Solving Set Constraint Satisfaction Problems Using ROBDDs. *JAIR*, 24(1):109–156, 2005.

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

[8] W.-J. van Hoeve, G. Pesant, L.-M. Rousseau, and A. Sabharwal. New Filtering Algorithms for Combinations of Among Constraints. *Constraints*, 14:273–292, 2009.