A Constraint Store Based on Multivalued Decision Diagrams

H. R. Andersen[1], T. Hadzic[1], J. N. Hooker[2], and P. Tiedemann[1]

[1] IT University of Copenhagen
hra,tarik,petert@itu.dk
[2] Carnegie Mellon University
john@hooker.tepper.cmu.edu

Abstract. The typical constraint store transmits a limited amount of information because it consists only of variable domains. We propose a richer constraint store in the form of a limited-width multivalued decision diagram (MDD). It reduces to a traditional domain store when the maximum width is one but allows greater pruning of the search tree for larger widths. MDD propagation algorithms can be developed to exploit the structure of particular constraints, much as is done for domain filtering algorithms. We propose specialized propagation algorithms for alldiff and inequality constraints. Preliminary experiments show that MDD propagation solves multiple alldiff problems an order of magnitude more rapidly than traditional domain propagation. It also significantly reduces the search tree for inequality problems, but additional research is needed to reduce the computation time.

1 Introduction

Propagation through a constraint store is a central idea in constraint programming, because it addresses a fundamental issue in the field. Namely, how can a solver that processes individual constraints recognize the global structure of a problem? Global constraints provide part of the answer, because they allow the solver to exploit the global structure of groups of constraints. But the primary mechanism is to propagate the results of processing one constraint to the other constraints. This is accomplished by maintaining a constraint store, which pools the results of individual constraint processing. When the next constraint is processed, the constraint store is in effect processed along with it. This partially achieves the effect of processing all of the original constraints simultaneously.

In current practice the constraint store is normally a domain store. Constraints are processed by filtering algorithms that remove values from variable domains. The reduced domains become the starting point from which the next constraint is processed. An advantage of a domain store is that domains provide a natural input to filtering algorithms, which in effect process constraints and the domain store simultaneously. A domain store also guides branching in a natural way. When branching on a variable, one can simply split the domain in the current domain store. As domains are reduced, less branching is required.

A serious drawback of a domain store, however, is that it transmits relatively little information from one constraint to another, and as a result it has a limited

effect on branching. A constraint store can in general be regarded as a relaxation of the original problem, because any set of variable assignments that satisfies the original constraint set also satisfies the constraint store. A domain store is a very weak relaxation, since it ignores all interactions between variables. Its feasible set is simply the Cartesian product of the variable domains, which can be much larger than the solution space.

This raises the question as to whether there is a natural way to design a constraint store that is more effective than a domain store. Such a constraint store should have several characteristics:

1. It should provide a significantly stronger relaxation than a domain store. In particular, it should allow one to adjust the strength of the relaxation it provides, perhaps ranging from a traditional domain store at one extreme to the original problem at the other extreme.
2. It should guide branching in a natural and efficient way.
3. In optimization problems, it should readily provide a bound on the optimal value (to enable a branch-and-bound search).
4. It should provide a natural input to the "filtering" algorithms that process each constraint.

In this context, a "filtering" algorithm would not simply filter domains but would tighten the constraint store in some more general way.

We propose a *multivalued decision diagram* (MDD) [1] as a general constraint store for constraint programming, on the ground that it clearly satisfies at least the first three criteria, and perhaps after some algorithmic development, the fourth as well. MDDs are a generalization of binary decision diagrams (BDDs) [2, 3], which have been used for some time for circuit design/verification [4, 5] and very recently for optimization [6–8]. The MDD for a constraint set is essentially a compact representation of a branching tree for the constraints, although it can grow exponentially, depending on the structure of the problem and the branching order. We therefore propose to use a *relaxed* MDD that has limited width and therefore bounded size. The MDD is relaxed in the sense that it represents a relaxation of the original constraint set.

An MDD-based constraint store satisfies the first three criteria as follows:

1. By setting the maximum width, one can obtain an MDD relaxation of arbitrary strength, ranging from a traditional domain store (when the maximum width is fixed to 1) to an exact representation of the original problem (when the width is unlimited).
2. One can easily infer a reduced variable domain on which to branch at any node of a branching tree.
3. A relaxed MDD provides a natural bounding mechanism for any separable objective function. The cost of a particular shortest path in the MDD is a lower bound on the optimal value.

The primary research issue is whether fast and effective "filtering" algorithms can be designed for limited-width MDDs. Rather than reduce domains, these

algorithms would modify the current MDD without increasing its maximum width. The feasibility of this project can be demonstrated only gradually by examining a number of constraints and devising MDD filters for each, much as was done for conventional domain filtering. As a first step in this direction we present an MDD filter for the all-different (alldiff) constraint and for inequality constraints (i.e., constraints that bound the value of a separable function).

2 Related Work

The limitations of the domain store's ability to make inferences across constraints are well known, but few attempts have been made to generalize it. In [9] meta-programming is proposed for overcoming the limitations. The idea is to allow *symbolic propagation* by giving the programmer detailed access to information about constraints at runtime and thereby allow for early detection of failures by symbolic combination of constraints.

Decision diagrams have been used previously for Constraint Satisfaction as a compact way of representing ad-hoc global constraints [10] as well as for no-good learning both in CSP and SAT solving [11]. This is however quite different from the goal of this paper, which is to utilize an MDD as a relaxation of the solution space.

3 MDDs and Solution Spaces

We are given a constraint set $\mathcal{C}$ defined over variables $X = \{x_1, \ldots, x_n\}$ with finite domains $D_1, \ldots, D_n$ giving rise to the declared domain of solutions $D = D_1 \times \cdots \times D_n$. Each constraint $C \in \mathcal{C}$ is defined over a subset of variables scope$(C) \subseteq X$. An MDD for $\mathcal{C}$ can be viewed as a compact representation of a branching tree for the constraint set.

For example, an MDD for the two constraints

$$\begin{aligned} &x_1 + x_3 = 4 \ \vee \ (x_1, x_2, x_3) = (1, 1, 2), \\ &3 \leq x_1 + x_2 \leq 5 \end{aligned} \tag{1}$$

with $x_1, x_2, x_3 \in \{1, 2, 3\}$ appears in Fig. 1(a). The edges leaving node u_1 represent possible values of x_1, those leaving u_2 and u_3 represent values of x_2, and so forth. Only paths to 1 are shown.

Formally, an MDD M is a tuple (V, r, E, var, D), where V is a set of vertices containing the special terminal vertex 1 and a root $r \in V$, $E \subseteq V \times V$ is a set of edges such that (V, E) forms a directed acyclic graph with r as the source and 1 as the sink for all maximal paths in the graph. Further, $var : V \rightarrow \{1, \ldots, n+1\}$ is a labeling of all nodes with a variable index such that $var(1) = n + 1$ and D is a labeling $D_{u,v}$ on all edges (u, v) called the *edge domain* of the edge. We require that $\emptyset \neq D_{u,v} \subseteq D_{var(u)}$ for all edges in E and for convenience we take $D_{u,v} = \emptyset$ if $(u, v) \notin E$.

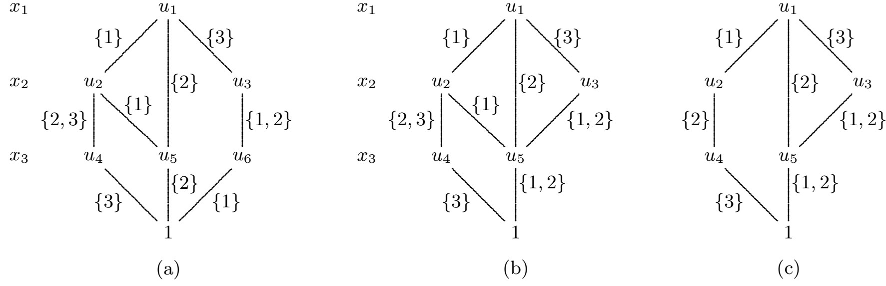


Fig. 1. (a) MDD for problem (1). (b) Same MDD relaxed to width 2. (c) Result of propagating alldiff in (b).

We work only with *ordered* MDDs. A total ordering $<$ of the variables is assumed and all edges (u, v) respect the ordering, i.e. $var(u) < var(v)$. For convenience we assume that the variables in X are ordered according to their indices. Ordered MDDs can be considered as being arranged in n *layers* of vertices, each layer being labeled with the same variable index. If $i = var(u)$ and $j = var(v)$, we say that (u, v) is a *long edge* if its skips variables ($i+1 < j$). We use $N_{u,v}$ for the set of indices $\{i + 1, i + 2, \dots, j - 1\}$ skipped by the long edge, with $N_{u,v} = \emptyset$ if (u, v) is not a long edge.

Each path from the root to 1 in an MDD represents a set of feasible solutions in the form of a Cartesian product of edge labels. Thus the path $(u_1, u_2, u_4, 1)$ in Fig. 1(a) represents the two solutions (x_1, x_2, x_3) in the Cartesian product $\{1\} \times \{2, 3\} \times \{3\}$. Note that the path $(u_1, u_5, 1)$ does not define the value of x_2, which can therefore take any value in its domain. This path represents the three solutions in $\{2\} \times \{1, 2, 3\} \times \{2\}$. Formally, a path $p = (u_0, \dots, u_m), u_0 = r, u_m = 1$ defines the set of solutions $Sol(p) = \prod_{i=0}^{m-1}(D_{u_i,u_{i+1}} \times \prod_{k \in N_{u_i,u_{i+1}}} D_k)$. Every path can in this way be seen as a traditional domain store. The solutions represented by an MDD are a union of the solutions represented by each path. One of the key features of an MDD is that due to sharing of subpaths in the graph, it can represent exponentially more paths than it has vertices. This means that we can potentially represent exponentially many domain stores as an approximation of the solution set.

An MDD M induces a Cartesian product *domain relaxation* $D^\times(M)$ whose k'th component is defined by

$$D_k^\times(M) = \begin{cases} D_k & \text{if there exists } (u, v) \text{ with } k \in N_{u,v}, \\ \bigcup\{D_{u,v} \mid var(u) = k\} & \text{otherwise.} \end{cases}$$

This relaxation shows that an MDD is a refined representation of the associated domain store. For a given set of solutions $S \subseteq D = D_1 \times \dots \times D_n$, it is convenient to let $S_{x_i=\alpha}$ be the subset of solutions in which x_i is fixed to α, so that $S_{x_i=\alpha} = \{(\alpha_1, \dots, \alpha_n) \mid \alpha_i = \alpha\}$. If MDD M represents solution set S, we let $M_{x_i=\alpha}$ be the MDD that represents $S_{x_i=\alpha}$ obtained by replacing the labels with $\{\alpha\}$ on every edge leaving a vertex u with $var(u) = i$. Also for a given D we let

```
msearch(Dpr, Mpr, D, M)
1:  D ← propagate(Dpr, D)
2:  M ← mprune(M, D) /* prune M wrt. D */
3:  repeat
4:      M ← mpropagate(Mpr, M)
5:      M ← mrefine(M)
6:  until no change in M
7:  D ← D^×(M) /* find the domain store relaxation of M */
8:  if ∃k. D_k = ∅ then return false
9:  else if ∃k.|D_k| > 1 then
10:     pick some k with |D_k| > 1
11:     forall α ∈ D_k: if msearch(Dpr, Mpr, D_{x_k=α}, M_{x_k=α}) then return true
12:     return false
13: else return true /* ∀k. |D_k| = 1, i.e., D is a single solution */
```

Fig. 2. The MDD-based search algorithm. The algorithms uses a set of domain propagators *Dpr* and a set of MDD-propagators *Mpr*. It works on a domain store relaxation D and an MDD constraint store M. In line 1 *propagate* is the standard domain propagation on D and in line 2 M is pruned with respect to the result of the domain propagation. Lines 3-6 mix MDD propagation and refinement until no further change is possible. This is followed by a new computation of the domain store relaxation, which is then used in guiding the branching in lines 10-11. An important operation is that of restricting the MDD store when a branch is made in line 11: $M_{x_k=\alpha}$. There are several options when implementing this operation as discussed in the main text.

$mprune(M, D)$ be the MDD that results from replacing each edge domain $D_{u,v}$ with $D_{u,v} \cap D_{var(u)}$.

We use an MDD of limited width as a constraint store (i.e., as a relaxation of the given constraint set). The *width* of an MDD is the maximum number of nodes on any level of the MDD. For example, Fig. 1(b) is a relaxed MDD for (1) that has width 2. It represents 14 solutions, compared with 8 solutions for the original MDD in Fig. 1(a). The relaxation is considerably tighter than the conventional domain store, which admits all 27 solutions in the Cartesian product $\{1, 2, 3\} \times \{1, 2, 3\} \times \{1, 2, 3\}$.

4 MDD-Based Constraint Solving

The MDD-based constraint solver is based on *propagation* and *search* just as traditional solvers. During search, branches are selected based on the domains represented in the MDD store. Propagation involves the use of a new set of MDD propagators and a *refinement* step. The overall algorithm is shown in Fig. 2.

The algorithm is a generalization of the normal constraint solving algorithm [12]. If lines 3–7 are omitted we get the domain store algorithm. The steps in line 1–2 are included for efficiency. If more powerful and efficient MDD propagators are developed, they might not be necessary. Further, the branching used in lines 10–11 could be replaced by more general branching strategies [12]. The operation

$M_{x_k=\alpha}$ can be implemented in several ways. The effect is that the constraint store is reduced to reflect the choice of $x_k = \alpha$. If the branching follows the MDD variable ordering, this is very simple as the new MDD store can be represented by moving the root to the node reached by following the edge corresponding to $x_k = \alpha$. If another order is used, the operation will have to be implemented through a marking scheme keeping track of the "live" parts of the MDD store, or by explicitly reducing the MDD store. The major logical inferences are made by the MDD (and domain) propagators.

Propagators work by pruning edge domains, and if a domain $D_{u,v}$ is pruned down to the empty set, the edge (u, v) is removed. This can lead to further reduction of the MDD, if after the removal of the edge some other edges do no longer have a path to 1 or can be reached by a path from r. MDD propagators are described in section 5.

The purpose of refinement is to refine the relaxation represented by the MDD so that further propagation can take place. Refinement is achieved through *node splitting* which refines the MDD by adding nodes while ensuring the maximal width is not exceeded (see section 6).

MDD-Based Constraint Optimization The MDD-based solver can be readily used for optimization. A separable objective function $\sum_i f_i(x_i)$ can be minimized over an MDD through efficient shortest path computations. Suppose for example we wish to minimize the function

$$20x_1^2 - 10x_2^2 + 15x_3^2 \tag{2}$$

subject to (1). The MDD is shown in Fig. 3(a). The corresponding edge weights appear in Fig. 3(b). Thus the weight of the edge (u_2, u_4) is $\min_{x_2 \in \{2,3\}}\{-10x_2^2\} = -90$, and the weight of the edge (u_1, u_5) is

$$\min_{x_1 \in \{2\}}\{20x_1^2\} + \min_{x_2 \in \{1,2,3\}}\{-10x_2^2\} = -10\,.$$

Formally, given a separable objective function $\sum_i f_i(x_i)$ each edge (u, v) with $i = var(u)$ is associated with the weight $w_f(u, v)$:

$$\min_{x_i \in D_{u,v}}\{f_i(x_i)\} + \sum_{k \in N_{u,v}} \min_{x_k \in D_k} f_k(x_k)\,. \tag{3}$$

A branch-and-bound mechanism computes a lower bound on the optimal value as the cost of the shortest path in the current MDD relaxation wrt. weights $w_f(u, v)$. The more relaxed the MDD representation, the weaker the bound. For example, the shortest path for the MDD in Fig. 3(b) is $(u_1, u_5, 1)$ and has cost 50. The lower bound wrt. the relaxed MDD of Fig. 1(b) (edge weights indicated in Fig. 3(c)) is 5. Finally, the lower bound based only on the domain store is -55.

If the bound is greater than the cost of the current incumbent solution b, the search is cut of. This is achieved by adding a new inequality constraint $\sum_i f_i(x_i) \leq b$ to the set of constraints and propagating it in future using a standard inequality propagator (discussed in section 5).

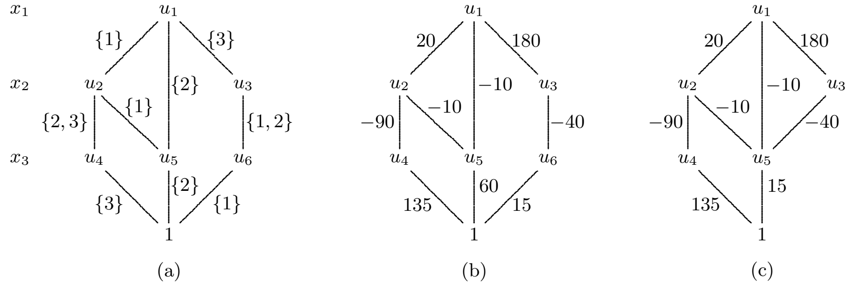


Fig. 3. (a) MDD for problem (1) copied from Fig. 1. (b) The same with edge weights for objective function (2). (c) Weights when the MDD is relaxed to width two.

5 Propagation

To characterize the strength of MDD propagators we define the notion of *MDD consistency* for a constraint C and an MDD M as the following condition: for every edge $(u, v) \in E$ and every value $\alpha \in D_{u,v}$, if α is removed from $D_{u,v}$ then there is at least one solution in M satisfying C that is lost. When the MDD is a single path, MDD consistency is equivalent to generalized arc consistency (GAC).

A key observation is that even if a constraint C has a polynomial-time algorithm that enforces GAC, it can be NP-hard to enforce MDD consistency for C on a polynomial sized MDD. This is because the Hamiltonian path problem, which is NP-hard, can be reduced to enforcing MDD consistency on an MDD of polynomial size. Consider the n-walk constraint on a graph G. Variable x_i is the ith vertex of G visited in the walk, and the only restriction is that x_{i+1} be vertex adjacent to x_i. Then $x_1, \ldots, x_n$ is a Hamiltonian path in G if and only if $x_1, \ldots, x_n$ is an n-walk on G and alldiff$(x_1, \ldots, x_n)$ is satisfied.

The MDD for the n-walk constraint has polynomial size because there are at most n nodes at each level in the MDD: From the root node there is an edge with domain $\{i\}$ to a node corresponding to each vertex i of G. From any node on level $k < n$ corresponding to vertex i there is an edge with domain $\{j\}$ to a node corresponding to each vertex j that is adjacent to i in G. Thus there may be no poly-time algorithm for MDD consistency even when there is a poly-time algorithm for domain consistency (unless P=NP).

Re-using domain propagators An intermediate step towards implementing customized MDD-based propagators is to reuse existing domain propagators. Though the above result on MDD consistency means that we should not expect a general polynomial time algorithm that can enforce MDD consistency, we can still reuse domain propagators to achieve tighter consistency than GAC.

Let *Dpr* be the set of domain propagators we wish to reuse. In order to apply such propagators to the MDD store, we must supply them with a domain in such a way that any domain reductions made by the propagators can be utilized for

edge domain reductions. To this end consider an edge (u, v) and let $M_{u,v}$ be the MDD obtained by removing all paths in M not containing the edge. Note that the domain relaxation $D^{\times}(M_{u,v})$ might be significantly stronger than $D^{\times}(M)$. We can now compute the simultaneous fixpoint D^{dom} of the propagators in Dpr over $D^{\times}(M_{u,v})$. For each assignment $x_k = \alpha$ consistent with $D^{\times}(M_{u,v})$ but not with D^{dom} we place a no-good $x_k \neq \alpha$ on the edge (u, v). We can use such a no-good to prune as follows. If (u, v) corresponds to the variable x_k we prune α from $D_{u,v}$ and remove the no-good. Otherwise, if (u, v) corresponds to a variable x_j,where $j < k$, we move the no-good to all incoming edges of u if the same no-good occurs on all other edges leaving u. In the case $j > k$, the no-good is moved to all edges leaving v if all incoming edges to v have the no-good. Intuitively, no-goods move towards the layer in the MDD which corresponds to the variable they restrict, and are only allowed to move past a node if all paths through that node agree on the no-good. This ensures that no valid solutions will be removed.

As an example, consider propagating an alldiff over Fig. 1(b). Examining (u_2, u_4) yields a no-good $x_2 \neq 3$ which can be immediately used for pruning D_{u_2,u_4}. Examining (u_1, u_2), yields the nogood $x_2 \neq 1$, which can be moved to (u_2, u_5) and (u_2, u_4), resulting in the former edge being removed. The result is shown in Fig. 1(c).

This type of filtering will reach a fixpoint after a polynomial number of passes through the MDD, and will hence only apply each domain propagator polynomially many times. However, since many domain store propagators can be quite costly to invoke even once it is important to develop specialized propagators that not only take direct advantage of the MDD store but also obtain stronger consistency.

Inequality Propagator Consider a constraint which is an inequality over a separable function:

$$\sum_i f_i(x_i) \leq b\,.$$

We can propagate such a constraint on an MDD by considering shortest paths in the MDD wrt. edge weights $w_f(u, v)$ (equation (3) in section 4). For each node u we calculate $L_{up}(u)$ and $L_{down}(u)$ as the shortest path from the root to u and from u to 1 respectively. L_{up} and L_{down} can be computed for all nodes simultaneously by performing a single bottom-up and top-down pass of the MDD. A value α can now be deleted from an edge domain $D_{u,v}$ iff every path through the edge (u, v) has weight greater than b under the assumption that $x_{var(u)} = \alpha$, that is if

$$L_{up}(u) + w_f(u, v) + L_{down}(v) > b\,.$$

Further details (for propagation in BDDs) can be found in [8]. We observe that this inequality propagator achieves MDD consistency as a value α in an edge domain $D_{u,v}$ is always removed unless there exists a solution to the inequality in the MDD going through the edge (u, v).

Alldiff Propagator As implied by the hardness proof of MDD consistency, a polynomial time MDD consistent propagator for the alldiff constraint would imply that P=NP. In this section we therefore suggest a simple labeling scheme that provides efficient heuristic propagation of an alldiff constraint over the MDD store.

To each node u we attach four labels for each alldiff constraint C: ImpliedUp, ImpliedDown, AvailUp and AvailDown. The label $\text{ImpliedUp}_C(u)$ is the set of all values α such that on all paths from the root to u there is an edge (v, w) where $x_{var(v)} \in \text{scope}(C)$ and $D_{v,w} = \{\alpha\}$. The label $\text{ImpliedDown}_C(u)$ is defined similarly for paths from u to the 1-terminal. Given these labels for a node u we can remove the value α from the domain of an edge (v, w) if $\alpha \in \text{ImpliedUp}(v)$ or $\alpha \in \text{ImpliedDown}(w)$.

In addition we use the label $\text{AvailUp}_C(u)$, which contains values α such that there exists at least one path from the root to u containing some edge (v, w) where $x_{var(v)} \in \text{scope}(C)$ and $\alpha \in D_{v,w}$.

Given some node u, consider the set of variables $X_a = \text{scope}(C) \cap \{x_k \mid k < var(u)\}$. If $|X_a| = |\text{AvailUp}_C(u)|$ (i.e., X_a forms a Hall Set [13]) the values in $\text{AvailUp}_C(u)$ cannot be assigned to any variables not in X_a. This allows us to prune these values from the domain of any outgoing edge of u. AvailDown_C is defined and used analogously.

When the labels of the parents of a node u are known, it is easy to update the labels ImpliedUp and AvailUp of u as follows. Let $\{(u_1, \alpha_1), \ldots, (u_k, \alpha_k)\}$ be pairs of parents and values, corresponding to incoming edges to u (nodes are repeated if the edge domain label on an edge contains more than one value). We then have

$$\text{ImpliedUp}_C(u) = \bigcap_{1 \le j \le k} (\text{ImpliedUp}_C(u_j) \cup \{\alpha_j\})$$

$$\text{AvailUp}_C(u) = \bigcup_{1 \le j \le k} (\text{AvailUp}_C(u_j) \cup \{\alpha_j\})$$

Using this formulation it is easy to see that computing ImpliedUp and AvailUp as well as pruning values based on these can be trivially accomplished in a single top-down pass of the MDD store. Similarly ImpliedDown and AvailDown can be computed in a single bottom-up pass.

6 Refining

Refining is achieved by *node splitting* in an MDD M. A node $u \in V$ is selected for splitting. We then add a new node u' to V and add edges as follows. Let $In_u = \{s \in V \mid (s, u) \in E\}$ be the set of vertices with an edge to u and let $Out_u = \{t \in V \mid (u, t) \in E\}$ be the set of vertices with an edge from u. The new edges are then $E' = \{(s, u') \mid s \in In_u\} \cup \{(u', t) \mid t \in Out_u\}$. The domains for the outgoing edges of u' are the same as for u: $D_{u',t} = D_{u,t}$ for all $t \in Out_u$. For the ingoing edges we select some domains $D_{s,u'} \subseteq D_{s,u}$ and remove these values from the domains in $D_{s,u}$, i.e., we update $D_{s,u}$ to $D_{s,u} \setminus D_{s,u'}$. A good selection

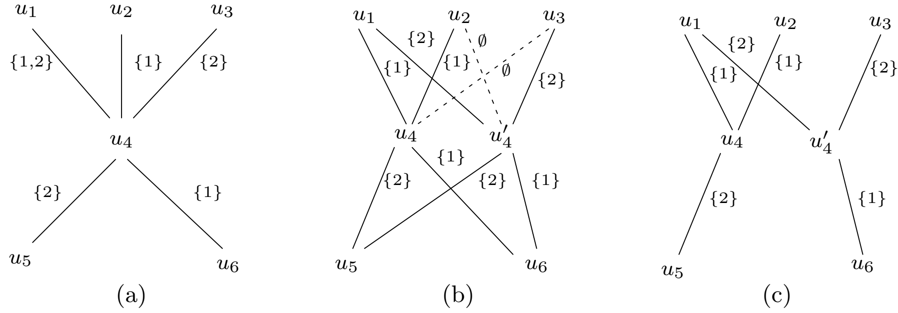


Fig. 4. (a) Part of an MDD store representing a relaxation of a global alldiff, just before splitting on the node u_4. Note that while there are obvious inconsistencies between the edge domains (such as label 1 in domains of (u_1, u_4) and (u_4, u_6)), we cannot remove any value. (b) A new node u_4' has been created and some of the edge domain values to u_4 have been transferred to u_4'. There are no labels on (u_2, u_4') and (u_3, u_4), so the edges need not be created. (c) After the split we can prune inconsistent values and as a result remove edges (u_4, u_6) and (u_4', u_5).

of the node u and the new ingoing edge domains will help the future MDD propagators in removing infeasible solutions from the MDD store. A good split should yield an MDD relaxation that is fine-grained enough to help propagators detect inconsistencies, but that is also coarse enough so that each removal of a value from an edge domain corresponds to eliminating a significant part of the solution space of the MDD. Fig. 4 shows an example of a node split and subsequent propagation.

The selection of the splitting node u and the ingoing edge domain splits is performed by a *node splitting heuristic*. We will use a heuristic that tries to select nodes that 1) are involved in large subsets of the solution space, and 2) have the potential of removing infeasible solutions during subsequent propagation steps. These criteria – estimating the strength of a split – must be balanced against the limit on the width of the MDD store and the increase in memory usage resulting from adding the new node and edges. For alldiffs C, the heuristic can utilize the labels ImpliedUp_C computed by the alldiff propagators. If $var(u) \in \text{scope}(C)$ then the larger the size of the ImpliedUp-set the more pruning could happen after splitting the node u. Also, the ability to prune significantly increases with the number of different constraints involving $var(u)$. Therefore, an estimate of strength for u could be the product of the sizes of the ImpliedUp_C-sets for each alldiff C s.t. $var(u) \in \text{scope}(C)$.

7 Computational Results

To obtain experimental results we integrated the MDD store into Gecode [14–16], a standard domain-store based solver. We implemented the MDD store as a specialized global constraint. We ensure that the MDD store is propagated

only after all other standard propagators in the model have stabilized, so the order of propagation is as indicated by Fig. 2. All experiments were run without recomputation (a memory management technique for backtracking search [12]). Since this slightly increased the performance of Gecode without the MDD store, this did not bias the results. All experiments were run on a desktop PC with an Intel Core 2 Duo 3.0 Ghz CPU and 2GB of memory.

In experiments involving alldiffs, we used both Gecode's domain propagator and our own MDD propagator. The domain propagator was set to provide domain consistency and the MDD propagator used only the up version of each label. Experiments involving optimization or linear inequalities over the MDD store used only the MDD-versions of inequality propagators. All instances were randomly generated.

We first compared the *size of the search tree* of domain-based search ST_D and MDD-based search ST_M. This gives an idea of how the extra processing at each node influences the search tree. We then took into account this extra processing, by counting the *total number of node-splits* NS_M and comparing ST_D against $ST_M + NS_M$. Finally, we compared the total running times T_D and T_M for domain-based and MDD-based search respectively.

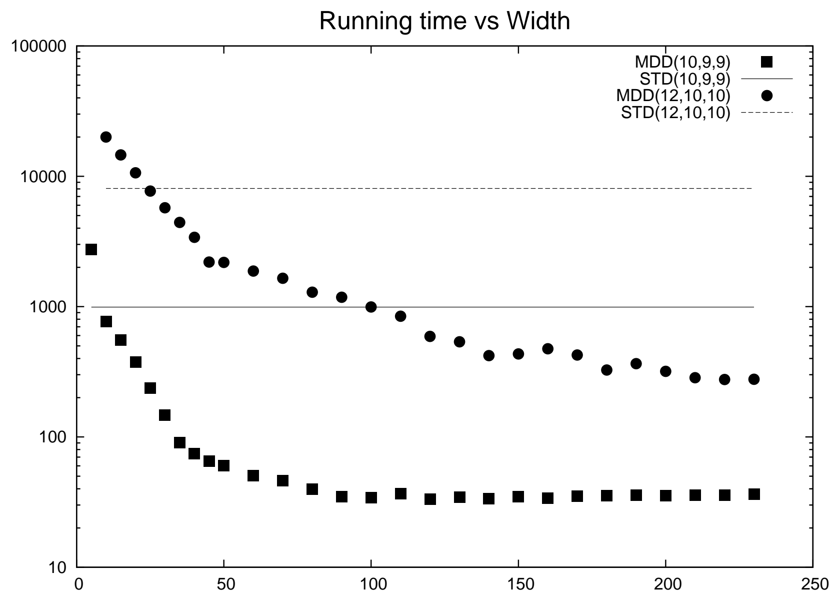
The first set of experiments was carried out for several different MDD widths. The problems consisted of three overlapping alldiff constraints. Each instance was parameterized by the triple (n, r, d), where n is the number of variables, r the size of the scope of each alldiff, and d the size of a domain. We ordered variables in decreasing order of the number of alldiffs covering them. This variable ordering was then used both as the ordering in the MDD store and for the search.

Remarkably, the MDD-based search never backtracked. The MDD-based refinement was strong enough to either detect infeasibility immediately, or to guide branching so that no backtracking was necessary. Hence, the size of the MDD-search tree is incomparably small – at most n. We therefore report only the comparison of ST_D against the $ST_M + NS_M$. This is shown in Fig. 5(a), for different widths. Fig. 5(b) compares the total running time for different widths. We can see, that when the total search effort is taken into account, the MDD-store still outperforms domain-based search by several orders of magnitude. We also see that the actual time closely corresponds to the number of node splits.

Width clearly has a dramatic effect both on the total computational effort $ST_M + NS_M$, as well as the running time T_M. The smallest width tested with the MDD store is 5 for the $(10, 9, 9)$ instances and 10 for the $(12, 10, 10)$ instances. The runtime is worse with the MDD store for small widths but becomes more than an order of magnitude better for larger widths. In the $(10, 9, 9)$ instances a width of 10 suffices to outperform the domain store, while 25 is required in the $(12, 10, 10)$ instances.

The second set of experiments minimized $\sum_i c_i x_i$ subject to $\sum_i a_i x_i \leq \alpha L$ and a single alldiff, where L is the maximum value of the left-hand side of the inequality constraint and $\alpha \in [0, 1]$ is a tightness parameter. The instances were parameterized by (n, k), where n is the total number of variables, and k the number of variables in the alldiff. The coefficients a_i and c_i were drawn

(a)



(b)

Fig. 5. (a) The two curves labeled MDD(n, r, d) show the total computational effort $ST_M + NS_M$ that resulted from an MDD constraint store on instances with parameters (n, r, d); every problem was solved without backtracking. The two lines labeled STD show the search tree size that resulted from a standard domain store. Each instance set consists of 60 randomly generated instances. Twenty instances were feasible for the $(10, 9, 9)$ set, 16 for the $(12, 10, 10)$ instances. (b) The same as before, but showing the actual running time in milliseconds. Note the close relation between $ST_M + NS_M$ and the time used.

randomly from the interval $[1, 100]$. We made the same comparisons as in the previous experiment.

The MDD store resulted in a consistently smaller search tree (Fig. 6(b)). The size of the search tree explored using the mdd store (ST_M) was as little as one-fifth the number explored by Gecode (ST_D). The total number of splits NS_M was also significantly smaller than ST_D in most cases (Fig. 6(a)). However, the computation time T_M was an order of magnitude greater than T_D. Similar results were obtained when minimizing the same objective functions subject to a single alldiff.

8 Conclusions and Future Work

Preliminary experiments indicate that an MDD-based constraint store can substantially accelerate solution of multiple-alldiff problems. Even a rather narrow MDD allows the problems to be solved without backtracking, while the traditional domain store generates about a million search tree nodes. As the MDD width increases, the MDD-based solution becomes more than an order of magnitude faster than a solver based on traditional domain filtering.

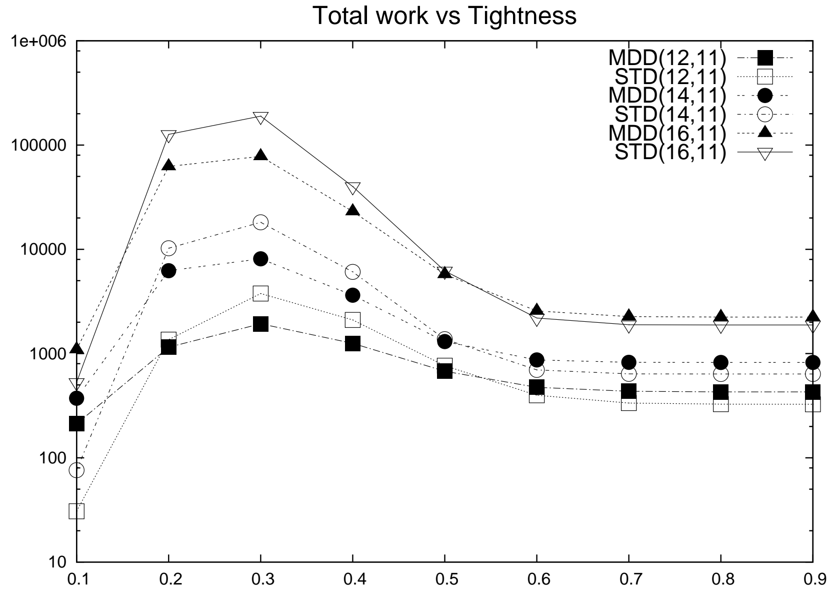
Conventional wisdom tells us that in constraint satisfaction problems, reducing the search tree through enhanced processing at each node often does not justify the additional overhead—perhaps because the added information is not transmitted in a conventional domain store. Our results suggest that intensive processing can be well worth the cost when one uses a richer constraint store.

We obtained less encouraging results for the optimization problems involving inequalities, where more intensive processing at each node is generally worthwhile. The MDD store required less branching than a traditional domain store, but the computation time was greater. An important factor affecting performance is that we cannot identify a feasible solution until we have branched deeply enough in the search tree to fix every variable to a definite value. This is time consuming because MDDs are processed at each node along the way.

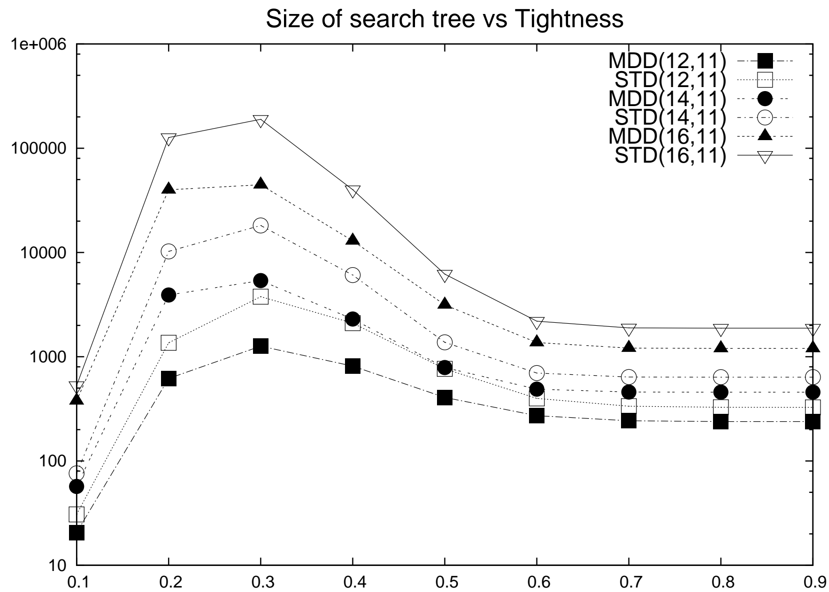
This problem is avoided in branch-and-cut methods for integer programming by solving the relaxation at each node and checking whether the solution happens to be integral. Very often it is, which allows one to find feasible solutions high in the search tree. This raises the possibility that we might "solve" the MDD relaxation (by heuristically selecting solutions that it represents) and check whether the solution is feasible in the original problem. This is an important project for future research and could substantially improve performance.

Overall, the results suggest that an MDD-based constraint store can be an attractive alternative for at least some types of constraints. Naturally, if an MDD is not useful for a certain constraint, the solver can propagate the constraint with traditional domain filtering.

A number of issues remain for future research, other than the one just mentioned. MDD-based propagators should be built for other global constraints, additional methods of refining the MDD developed, and MDD-based constraint stores tested on a wide variety of problem types.



(a)



(b)

Fig. 6. (a) Total computation effort vs. the tightness α when minimizing $\sum_i c_i x_i$ subject to $\sum_i a_i x_i \leq \alpha L$ and a single alldiff, where L is the maximum size of the left-hand side of the inequality. MDD(n, k) denotes the MDD-based results for instances parameterized by (n, k), and STD denotes results based on standard domain filtering. For MDD-based results, computational effort is the sum $ST_M + SN_M$ of the number search tree nodes and the number of splits, and for domain filtering it is the number ST_D of search tree nodes. (b) Number of search tree nodes vs. the tightness α when minimizing $\sum_i c_i x_i$ subject to $\sum_i a_i x_i \leq \alpha L$ and a single alldiff.

Acknowledgments We would like to thank the anonymous reviewers for their valuable suggestions which helped us improve the presentation significantly.

References

1. Kam, T., Villa, T., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Multi-valued decision diagrams: Theory and applications. International Journal on Multiple-Valued Logic **4** (1998) 9–62
2. Andersen, H.R.: An introduction to binary decision diagrams. Lecture notes, available online, IT University of Copenhagen (1997)
3. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers **C-27** (1978) 509–516
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35** (1986) 677–691
5. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell Systems Technical Journal **38** (1959) 985–999
6. Becker, Behle, Eisenbrand, Wimmer: BDDs in a branch and cut framework. In: International Workshop on Experimental and Efficient Algorithms (WEA), LNCS. Volume 4. (2005)
7. Hadzic, T., Hooker, J.: Postoptimality analysis for integer programming using binary decision diagrams. Technical report, Carnegie Mellon University (2006) Presented at GICOLAG workshop (Global Optimization: Integrating Convexity, Optimization, Logic Programming, and Computational Algebraic Geometry), Vienna.
8. Hadzic, T., Hooker, J.N.: Cost-bounded binary decision diagrams for 0-1 programming. Technical report, Carnegie Mellon University (to appear)
9. Muller, T.: Promoting constraints to first-class status. In: Proceedings of the First International Conference on Computational Logic, London. (2000)
10. Cheng, K.C., Yap, R.H.: Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In: Proceeedings of The European Conference on Artificial Intelligence. (2006)
11. Hawkins, P., Stuckey, P.J.: A hybrid BDD and SAT finite domain constraint solver. In Hentenryck, P.V., ed.: Proceedings of the Practical Applications of Declarative Programming, 8th International Symposium. Volume 3819 of LNCS., Springer (200) 103–117
12. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. In Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier Science Publishers, Amsterdam, The Netherlands (2006) 495–526
13. van Hoeve, W.J.: The alldifferent constraint: A survey. In: Sixth Annual Workshop of the ERCIM Working Group on Constraints. (2001)
14. Schulte, C.: Programming Constraint Services. Volume 2302 of Lecture Notes in Artificial Intelligence. Springer-Verlag (2002)
15. Schulte, C., Stuckey, P.J.: Speeding up constraint propagation. In Wallace, M., ed.: Tenth International Conference on Principles and Practice of Constraint Programming. Volume 3258 of Lecture Notes in Computer Science., Toronto, Canada, Springer-Verlag (September 2004) 619–633
16. Schulte, C., Szokoli, G., Tack, G., Lagerkvist, M., Pekczynski, P.: Gecode. Software download and online material at the website: www.gecode.org