Noname manuscript No.
(will be inserted by the editor)

MDD Propagators with Explanation

Graeme Gange · Peter J. Stuckey · Radoslaw Szymanek

Abstract Multi-valued decision diagrams (MDDs) are a convenient approach to representing many kinds of constraints including `table` constraints, `regular` constraints, complex set and multiset constraints, as well as ad-hoc problem specific constraints. This paper introduces an incremental propagation algorithm for MDDs, and explores several methods for incorporating explanations with MDD-based propagators. We demonstrate that these techniques can provide significantly improved performance when solving a variety of problems.

1 Introduction

In many problem domains, it is necessary to efficiently enforce either ad-hoc problem specific constraints or common constraints which are not supported by the chosen solver software. In these cases, it is normally necessary to either build a new propagator for the needed constraint, or to use a decomposition of the constraint. Neither of these is ideal – building a new global propagator requires nontrivial effort, and decompositions may have poor performance and propagation.

Decision diagrams are a convenient solution to this problem. Efficient algorithms for manipulating Binary Decision Diagrams (BDDs) [5] and Multi-valued Decision Diagrams (MDDs) allow us to easily construct representations of a variety of constraints; and rather than construct propagators for each new constraint, we merely need to use a generic MDD propagator working with MDD representation of the required constraint. Common constraints that can be efficiently represented using MDDs include `table` constraints [6], `regular` constraints [17] and complex set and multiset constraints [9].

G.Gange · P. Stuckey
National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
E-mail: {ggange,pjs}@csse.unimelb.edu.au

R. Szymanek
École Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory (LIA)

The `regular` constraint [17] provides a convenient means for encoding a variety of global constraints, including placement [14], `sequence` [12] and `slide` [3] constraints. Accordingly, a number of methods have been proposed to enforce consistency over these constraints. Pesant [17] introduced propagators for the `regular` constraint, which maintain explicit sets of supports for nodes and edges. Decompositions into ternary constraints [18] and clauses [19] have also been proposed. More recently, MDD-based propagators [6] have also been used.

A number of algorithms have been developed for enforcing consistency over constraints using BDDs, either by constructing new BDDs during propagation [9, 10] or by traversing a static representation of the constraint BDD [8]. Similar traversal-based algorithms have been developed for MDD constraints [6], as have approaches which construct MDDs dynamically as a constraint store [1, 11]. The algorithm of [6] also incorporates some incremental properties, recording those subgraphs which can no longer contain a solution; however, as it traverses from the root, it must still explore large sections of the graph for which the corresponding domains may be unchanged during a given step.

In this paper we present the first MDD propagator which *explains* its propagations. Recent work on lazy clause generation [16] has demonstrated that combining general constraint propagation with SAT-based conflict analysis and activity-based search can provide a significant advantage against both purely propagation-based solvers and a direct SAT-encoding of finite-domain constraint problems. For learning to occur, a propagator must be able to construct an *explanation* for any domain changes it causes.

Explanation for MDD propagators is particularly interesting since a propagation in an MDD may have many possible explanations. Also the overhead of explaining an MDD propagation is considerable, each domain change may require as much work to explain as the whole propagation of the MDD. Hence there are interesting challenges in building an MDD propagator with explanation.

The contributions of this paper are:

- The first description of how to build MDD propagators that explain themselves.
- A new incremental approach to MDD propagation using watched literals that avoids traversing parts of the MDD for which the corresponding domains have not changed.
- A novel incremental approach to explanation for MDD propagators, which avoids traversing the entire MDD to create each explanation.
- Improvements to explanation in order to create shorter explanations and reduce the overhead of explanation.
- Experimental results showing that the combination of MDD propagation with explanation can solve some problems substantially faster than other methods.

The remainder of this paper is organized as follows: In Section 2 we introduce MDDs and the standard propagation algorithm [6]. In Section 3 we show how we can define an incremental propagator for MDDs using watched literals which never visits parts of the MDD which are unchanged after changes in domains. In Section 4 we show how we can explain MDD propagation, first using a non-incremental approach which explores the entire MDD to make an explanation, and then incrementally. In Section 5 we give experimental results for various combinatorial problems where we compare various versions of our MDD propagators,

and also compare against state-of-the-art alternative methods for solving these problems. Finally in Section 6 we conclude.

2 Propagating MDDs

We assume we are solving a constraint satisfaction problem over set of variables $x \in \mathcal{V}$, each of which takes values from a given finite set of values or *domain*. Let D be a domain then, $D(x)$ is the set of possible values for variable x. Define $D \sqsubseteq D'$ iff $D(x) \subseteq D'(x), \forall x \in \mathcal{V}$. The constraints of the problem are represented by propagators f which are functions from domains to domains which are monotonically decreasing $f(D) \sqsubseteq f(D')$ whenever $D \sqsubseteq D'$, and contracting $f(D) \sqsubseteq D$.

In a lazy clause generation solver integer domains are also represented using Boolean variables. Each variable x with initial domain $D(x) = [l..u]$ is represented by two sets of Boolean variables $[\![x = d]\!], l \leq d \leq u$ and $[\![x \leq d]\!], l \leq d < u$ which define which values are in $D(x)$. For example if variable x has initial domain [0..5] and at some later stage $D(x) = \{1, 3\}$ then the literals $[\![x \leq 3]\!], [\![x \leq 4]\!], \neg[\![x \leq 0]\!], \neg[\![x = 0]\!]$, $\neg[\![x = 2]\!], \neg[\![x = 4]\!], \neg[\![x = 5]\!]$ will hold. Explanations are defined by clauses over this Boolean representation of the variables.

2.1 MDDs

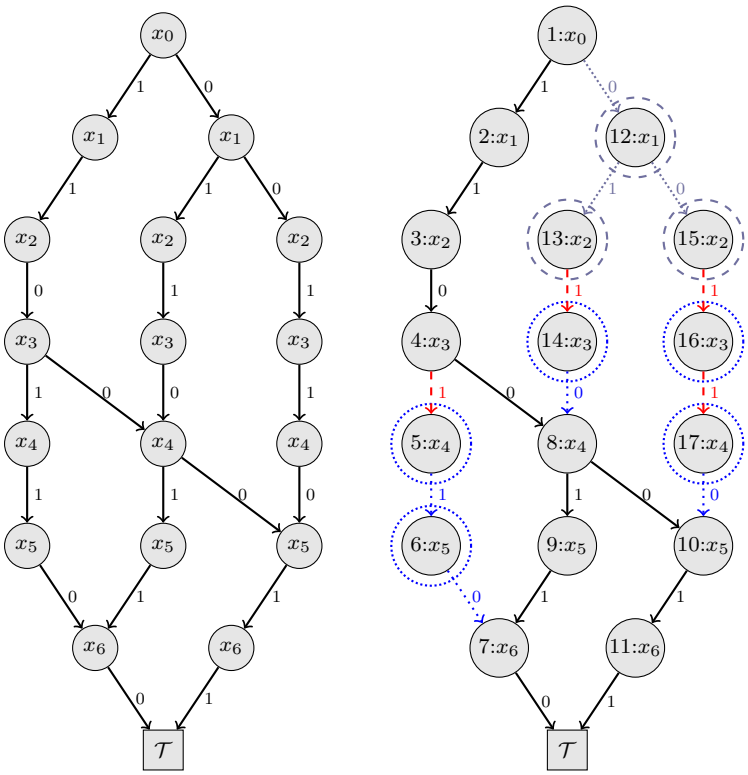
A multi-valued decision diagram [20], G, is a directed acyclic graph representing a Boolean valued function over a set of variables. Each internal node in an MDD G, $n_0 = node(x, [(v_1, n_1), (v_2, n_2), \cdots, (v_k, n_k)])$ is labeled with a variable x and outgoing arcs consisting of a value v_i and a destination node n_i. Define $node.var$ as the variable label appearing in the node, i.e. $n_o.var = x$. Each value v_i is in the initial domain of x. There is a final node $\mathcal{T}$ which represents *true*. Let $G.root$ be the root node of an MDD G. We can understand an MDD node G where $n_0 = G.root$ as representing the constraint $\ll n_0 \gg$ where

$$\ll n_0 \gg \equiv \bigvee_{i=1}^{k} ((x = v_i) \wedge \ll n_i \gg)$$

and $\ll \mathcal{T} \gg \equiv true$. A binary decision diagram (BDD) is a special case of an MDD where the variables x appearing in the MDD are Boolean. We denote by $|G|$ the number of edges in MDD G.

We assume that MDDs are *ordered* and without *long edges*, that is there is mapping σ from variables in the MDD to distinct integers such that for each internal node n_0 of the form above $\sigma(n_i.var) = \sigma(n_0.var) + 1, \forall 1 \leq i \leq k$ where $n_i \neq \mathcal{T}$. The condition can be loosened to $\sigma(n_i.var) > \sigma(n_0.var)$ (which allows long edges) but this complicates the algorithms considerably, and in practice the complication usually overcomes any benefits of treating long edges directly (unlike the case for BDDs).

For convenience, we will refer to an edge e as a tuple (x, v_i, s, d) of a variable x, value v_i, source node $s = n_0$ and destination node $d = n_i$. We will refer to the components as $(e.var, e.val, e.begin, e.end)$. An edge $e = (x, v_i, s, d)$ is said to be *alive* if it occurs on some path from the root of the graph to the terminal $\mathcal{T}$.



(a) Constraint (b) $x_2 \neq 1, x_3 \neq 1$

Fig. 1: An example MDD for a `regular` constraint $0^\star 1100^\star 110^\star$ over the variables $[x_0, x_1, x_2, x_3, x_4, x_5, x_6]$, and the effect of propagating $x_2 \neq 1$ and $x_3 \neq 1$.

Otherwise, it is said to be *killed*. An edge e becomes killed if v_i is removed from the domain of x, all paths from the root r to s cross killed edges (*killed from above*), or all paths from d to $\mathcal{T}$ cross killed edges (*killed from below*).

We use $s.out\_edges$ to refer to all the edges of the form $(\_, \_, s, \_)$, those leaving node s, and $d.in\_edges$ to refer to edges of the form $(\_, \_, \_, d)$ those entering node d. Similar to above, a node is said to be *killed* if it does not occur on any reachable path from the root node r to $\mathcal{T}$. A node becomes killed if either all incoming or all outgoing edges become killed. As a result, we can determine if a given node n is killed by examining its incoming or outgoing edges. We use $G.edges(x, v_i)$ to record the set of edges of the form $(x, v_i, \_, \_)$ in MDD G.

Example 1 Consider the MDD (actually a BDD) shown in Figure 1(a). Suppose we set $x_2 \neq 1$ and $x_3 \neq 1$. The edges shown dashed in Figure 1(b) are killed by domain change. Then the edges $\{(x_0, 0, 1, 12), (x_1, 1, 12, 13), (x_1, 0, 12, 15)\}$ and nodes $\{12, 13, 15\}$ are killed from below while the edges $\{(x_4, 1, 5, 9), (x_5, 0, 6, 7), (x_3, 0, 14, 8), (x_4, 0, 17, 10)\}$ and nodes $\{5, 6, 14, 17\}$ in Figure 1(b) are killed from above. □

```
propagate(G) {
    reacht = {T}; % Reset the set of nodes that can reach true.
    unsupported = {}; % The set of (var,val) pairs which need support.
    for var in vars(G) {
        % Collect the set of values that need support.
        unsupported ⊔={(var, val) | val ∈ D(var)};
    }
    traverse(G.root); % Remove the supported values.
    return unsupported
}

traverse(node) {
    if node ∈ reacht { return true; } % Already visited.
    if node ∈ failed { return false; }% Already visited.
    for edge ∈ node.out_edges {
        if edge.val ∈ D(edge.var) {
            if traverse(edge.end) { % edge is alive.
                unsupported = unsupported \ {(edge.var, edge.val)};
                reacht ⊔={node};
                % Shortcircuit, if all values below here are supported.
                if ∄(var, val) ∈ unsupported where var ≥ edge.var { break; }
            }
        }
    }
    if node ∉ reacht { failed ⊔={node}; }
    return node ∈ reacht;
}
```

Fig. 2: MDD propagation from the root, remembering false nodes and shortcircuiting [6].

2.2 Propagation of an MDD from the Root

Propagation for an MDD [6] (or BDD [8]) from the root is reasonably straight forward. The graph is traversed from the root node, marking each reached node (so that it is not revisited) with whether or not the node still has a path to $\mathcal{T}$ given the current domains of variables. Any edge (x, v_i, s, d) on such a path gives support for the value v_i for x. Any values in the current domain of x that are not supported after the traversal is finished are removed. Cheng and Yap [6] made this process more incremental by recording nodes in the graph that were previously determined not to reach $\mathcal{T}$, and sped up the search by recording for which variables all values in the current domain are still supported.

Pseudo-code for the algorithm is shown in Figure 2. The algorithm uses three variables: *reacht* records the nodes that can reach $\mathcal{T}$ using the current variables domains, it is reset on each invocation to empty; *failed* records the nodes that cannot reach $\mathcal{T}$, nodes are added to it during forward computation, it is reset only on backtracking to a previous state; and *unsupported* records the (var, val) pairs of variables and value which have yet to have been shown to be supported. The shortcircuit enhancement of [6] is the **break** test. If all the values for variables greater than the current variable are already supported, then there is no need to examine the MDD below this point.

Example 2 Propagation of the MDD shown in Figure 1(b), after $x_2 \neq 1$ and $x_3 \neq 1$, traverses the MDD from the root visiting all nodes except $\{5, 6, 14, 16, 17\}$. The solid arcs of Figure 1(b) are determined to be on paths from the root to $\mathcal{T}$ and hence support values. There is no support found for $x_0 = 0$, $x_1 = 0$, or $x_5 = 0$ so their negations are new inferences made by the propagation algorithm. □

The only state of the non-incremental propagator that survives between propagations is *failed*. Cheng and Yap [6] show how this can be efficiently implemented using a sparse set datastructure, that allows membership testing, addition of a new element, and backtracking to previous states each in constant time.

3 Incremental Propagation

When a value v_i is removed from the domain of a variable x, the edges corresponding to that value are killed. An edge (x, v_i, s, d) being killed in this way can only cause changes if it is the last remaining outgoing edge of s, in which case it will kill s and all incoming nodes to s, or the last incoming edge of d, in which case it may kill all outgoing edges from d. Thus, if s (and d) have other incoming (and outgoing) edges remaining, we need not explore more distant parts of the graph. If this is not the case, however, we must repeat this process for the new edges that have been killed.

Similarly, an edge (x, v_i, s, d) can cause v_i to be removed from the domain of x if and only if all other edges supporting that value are killed. Thus, we want to efficiently determine whether or not a given edge is the last remaining edge for the given value. However, keeping edge counts for nodes and values is not desirable, as we would then have to restore these counts upon backtracking. Accordingly, we adopt a similar method to the two-literal watching scheme [15] used in SAT solvers.

We associate with each edge flags indicating whether (a) the edge is alive, and (b) whether it provides support for a value, the node above, or the node below. We initially mark one edge for each value as *watched*, along with one incoming and outgoing edge for each node. When an edge is removed, it is marked as killed, then the watch flags are examined. If none of the watch flags are set, the edge cannot cause any further changes to the graph or domains. If it is watched by a node, we must then search the corresponding node for a new watched edge; if none can be found, the node is killed, and further propagation occurs. Likewise, if it is watched by a value, we must then search for a new supporting edge; if none exists, the corresponding value is removed from the domain. Otherwise, the new edge is marked as watched, and the mark is removed from the old edge.

Pseudo-code for the algorithm is given in Figures 3 and 4. Algorithm propagate takes an MDD G and a set of pairs (var, val) where $var \neq val$ is the change in domains by new propagation. The MDD graph G maintains a status $G.status[e]$ for each edge e as either: alive, dom killed by domain change, below killed from below (no path to $\mathcal{T}$ from $e.end$), or above killed from above (no path from the root to $e.start$). It also maintains a watched edge for each node n's input ($n.watch\_in$), output ($n.watch\_out$), and each (var, val) pair ($G.support[var, val]$). For simplicity of presentation, the information about how each edge e is being watched is also recorded as $G.watched[e] \subseteq \{\mathsf{begin}, \mathsf{end}, \mathsf{val}\}$. If $\mathsf{begin} \in G.watched[e]$, the edge e

is watched by the node $e.begin$; likewise for end and val. The pseudo-code for upward_pass is omitted since it is completely analogous to downward_pass. The graph also maintains a trail of killed edges $G.trail$ (which is initially empty) and, for each (var, val) pair, a pointer to the trail when it was removed ($G.limit[var, val]$). The list *kfa* holds the set of nodes that may have been killed due to removal of incoming edges (killed from above); *kfb* is used similarly with regard to outgoing edges. Note that restoring the state of the propagator, restore_to shown in Figure 4, requires only restoring the status of killed edges to alive. The maximum trail size is the number of edges in G.

The incpropagate algorithm enforces domain consistency on the MDD. The complexity of incpropagate is $O(|G|)$ down a branch of the search tree (with a little care in implementation). In each forward computation each edge is only killed once. It is easy to see that each edge is only considered at most once in the inner **for** loop of incpropagate. Each node n can appear in *kfa* at most $|n.in\_edges|$ times, hence the **for** loop in downward_pass runs $O(|G|)$ times. By traversing $n.in\_edges$ (in downward_pass) from the previously watched edge, we can guarantee that we only traverse each edge twice down the branch of the search tree. Similarly when traversing $G.edges[var, val]$ (in collect) looking for new support, if we look from the previously watched edge we can guarantee we only traverse each edge at most twice down the branch of the search tree.

Example 3 Consider the MDD shown in Fig 1(a). If the values $x_2 = 1$ and $x_3 = 1$ are removed from the domain, we must mark the corresponding edges as removed. These edges are shown dashed in Fig 1(b).

Incremental propagation works as follow assuming the leftmost edge leaving and entering a node is watched, and the leftmost edge for each $x = d$ valuation is watched. The removal of the edge $(x_2, 1, 13, 14)$ removes the support for node 13 which is added to *kfb*, as denoted by operation $\cup\!\!=$, and node 14 which is added to *kfa*. Similarly 15 is added to *kfb* and 16 to *kfa* by the removal of $(x_2, 1, 15, 16)$. The removal of the edges $(x_3, 1, 4, 5)$ and $(x_3, 1, 16, 17)$ leave $kfa = \{5, 14, 16, 17\}$ and $kfb = \{4, 13, 15, 16\}$ before downward_pass execution.

We then perform the downward pass. We find no new supports from above for 5 which means we mark $(x_4, 1, 5, 6)$ as killed from above (above) and add 6 to *kfa* and add $(x_4, 1)$ to the queue of values to check *pinf*. Similarly we kill $(x_5, 0, 6, 7)$ and add 7 to *kfa* and $(x_5, 0)$ to *pinf*. We do find a new support from above for node 7. We similarly kill edges $(x_3, 0, 14, 8)$ but note since this is neither watched by its destination nor its value, nothing is added to *kfa* or *pinf*. We similarly kill the edge $(x_4, 0, 17, 10)$ but again this is not watched.

We then perform the upward pass. We find a new support for node 4 from below. We find no new supports for nodes 13 hence we kill edge $(x_1, 1, 12, 13)$ and add 12 to *kfb*, We similarly kill node 15 and edge $(x_1, 0, 12, 15)$ which adds $(x_1, 0)$ to *pinf*. Examining node 12 we find no support from below and kill $(x_0, 0, 1, 12)$ adding $(x_0, 0)$ to *pinf* (but not 1 to *kfb*). The killed from below nodes and edges are shown dashed in Figure 1(b), while the killed from above nodes and edges are shown dotted.

We finally consider $pinf = \{(x_4, 1), (x_5, 0), (x_1, 0), (x_0, 0)\}$. We find a new support $(x_4, 1, 8, 9)$ for $x_4 = 1$, therefore we remove *val* from $G.watches$ of edge $(x_4, 1, 5, 6)$ as denoted by $-\!\!=$ operation. We are not able to find new supports

```
incpropagate(G, changes) {
    kfa = {}; % The set of nodes that may have been killed from above.
    kfb = {}; % Nodes which may have been killed from below.
    pinf = {}; % (var,val) pairs that may be removed from the domain.
    count = length(G.trail); % Record how far to unroll the trail to get back to this state.

    for (var, val) in changes {
        G.limit[var, val] = count; % Mark the restoration point.
        % Kill all remaining edges for the value.
        for edge in G.edges(var, val) {
            if G.status[edge] ≠ alive { continue; }
            G.status[edge] = dom; % Mark the edge as killed due to external inference.
            insert(G.trail, edge); % Add the edge to the trail
            if begin ∈ G.watched[edge] {
                % If this edge supports the above node e.begin,
                % add the node to the queue for processing.
                kfb ⊔=={edge.begin};
            }
            if end ∈ G.watched[edge] {
                % Likewise, add the end node if it is supported by the edge.
                kfa ⊔=={edge.end};
            }
        }
    }
    pinf = downward_pass(G, kfa)
    if G.status[T.watch_in] ≠ alive {
        % If T is unreachable, the partial assignment is inconsistent.
        % Otherwise, propagating upwards is safe.
        return FAIL;
    }
    pinf ⊔= upward_pass(G, kfb)
    return collect(G, pinf);
}
```

Fig. 3: Top level of the incremental propagation algorithm.

for the other variable value pairs. Propagation determines that $x_5 \neq 0$, $x_1 \neq 0$ and $x_0 \neq 0$.

□

4 Explaining MDD Propagation

A nogood learning solver, upon reaching a conflict, analyses the inference graph to determine some subset of assignments that results in a conflict. This subset is then added to the solver as a *nogood* constraint, preventing the solver from making the same set of assignments again, and reducing the search space.

The use of nogood learning has been shown to provide dramatic improvements to the performance of BDD-based constraint solvers [10]. In order to use an MDD propagator in a nogood learning solver it must be able to explain its inferences. These explanations form the inference graph, which is used to construct the nogood. The explanations can be constructed eagerly during propagation, or lazily

```
downward_pass(G, kfa) {
    pinf = {};
    for node in kfa {
        % Search for a new support
        for edge in node.in_edges {
            if G.status[edge] = alive {
                % Support found. Update the watches.
                G.watched[node.watch_in] -= {end};
                G.watched[edge] ∪={end};
                node.watch_in = edge;
                break;
            }
        }
        if is_dead(node.watch_in) {
            % The node is still dead
            % kill the outgoing edges.
            for edge in node.out_edges {
                if G.status[edge] ≠ alive { continue }
                G.status[edge] = above;
                insert(G.trail, edge);
                if end ∈ G.watched[edge] {
                    % If the edge supports a node,
                    % queue it for processing.
                    kfa ∪={edge.end};
                }
                if val ∈ G.watched[edge] {
                    % If the edge supports a value,
                    % add it to the queue.
                    pinf ∪={(edge.var, edge.val)};
                }
            }
        }
        return pinf;
    }
```

```
collect(G, pinf) {
    inf = {};
    for (var, val) in pinf {
        % Search for a new support.
        edge = G.support[var, val];
        for e in G.edges(var, val) {
            if G.status[e] = alive {
                % Support found.
                G.watched[edge] -= {val};
                G.watched[e] ∪={val};
                G.support[var, val] = e;
                break;
            }
        }
        edge = G.support[var, val];
        if G.status[edge] ≠ alive {
            % Still dead.
            inf ∪={(var, val)};
            G.limit[var, val] = count;
        }
    }
    return inf;
}
```

```
restore_to(G, var, val) {
    % Determine how far to unroll
    lim = G.limit[var, val];
    while length(G.trail) > lim {
        % Restore the propagator.
        edge = pop_last(G.trail);
        G.status[edge] = alive;
    }
}
```

Fig. 4: Pseudo-code for determining killed edges and possibly removed values in the downward pass, collecting inferred removals, and backtracking.

as needed for nogood construction. For more details on conflict generation we refer the reader to [16].

4.1 Non-incremental Explanation

The best current approach to explaining BDD propagation is due to Subbarayan [21]. Here we extend this approach to MDDs. It works in two passes, it first traverses the MDD backwards from the true node $\mathcal{T}$ marking which nodes can reach $\mathcal{T}$ in the current state assuming the negation of the inference to be explained holds. It then performs a breadth-first traversal from the root progressively adding back domain values as long as this does not create a path to $\mathcal{T}$. The algorithm creates a minimal explanation (removing any part of it does not create a correct explanation), but it requires traversing the entire MDD once for each new inference. Note it does not create a *minimum size explanation*, doing so is NP-hard [21]. Pseudo-code explaining the inference $var \neq val$ is given in Figure 5.

```
explain(G, var, val) {
    reacht = mark_reacht(G, var, val); % Find the set of nodes that can reach true.
    explanation = {};
    queue = {G.root};
    while queue ≠ ∅ {
        for node in queue {
            for e ∈ node.out_edges {
                if e.var ≠ var and e.end ∈ reacht {
                    explanation ⊔= (e.var, e.val)
                }
            }
        }
        nqueue = {}; % Record nodes of interest on the next level.
        for node in queue {
            for e ∈ node.out_edges {
                if (e.var == var and e.val == val)
                        or (e.var ≠ var and (e.var, e.val) ∉ explanation) {
                    nqueue ⊔= e.end
                }
            }
        }
        queue = nqueue;
    }
    return explanation;
}

mark_reacht(G, var, val) {
    reacht = {T}; % Reset the set of nodes that can reach true.
    queue = {T.in_edges}; % Reset the queue of nodes to be processed.
    for edge in queue {
        if edge.begin ∈ reacht continue;
        if edge.var == var {
            if edge.val == val {
                reacht ⊔= {edge.begin};
                queue ⊔= edge.begin.in_edges;
            }
        } else if G.status[edge] == alive and edge.begin ∉ reacht {
            reacht ⊔= {edge.begin};
            queue ⊔= {edge.begin.in_edges};
        }
    }
    return reacht;
}
```

Fig. 5: Non-incremental MDD explanation. Extended from [21].

Example 4 Consider explaining the inference $x_0 \neq 0$ discovered in Example 2. The mark_reacht call walks the MDD from $\mathcal{T}$ adding which nodes can reach $\mathcal{T}$ into *reacht* in the state where the inference was performed (Figure 1(b)) with the additional assumption that the reverse of the inference holds ($x_0 = 0$). It discovers that all nodes reach $\mathcal{T}$ except $\{1, 12, 13, 15, 16\}$. It then does a breadth-first traversal from the root looking for currently killed edges that if not excluded would create a path from the root to $\mathcal{T}$. From the root we only reach node 12 (under the assumption that $x_0 = 0$), from 12 we reach 13 and 15. Restoring the killed edge $(x_2, 1, 13, 14)$ would create a path to $\mathcal{T}$, hence we require $x_2 \neq 1$ in the reason. Once we have

this requirement, from 15 we cannot reach 16 and the algorithm stops with the explanation $\neg[\![x_2 = 1]\!] \rightarrow \neg[\![x_0 = 0]\!]$. □

4.2 Incremental Explanation

The non-incremental explanation approach above requires examining the entire MDD for each new inference made. This is a significant overhead although one should note that explanations are only required to be generated during the computation of a nogood from failure, not during propagation, hence not every inference will need to be explained. Once we are using incremental propagation, the overhead of constructing minimal explanations is relatively even higher.

It is difficult to see how to generate a minimal explanation incrementally, since the minimality relies on examining the whole MDD. Thus we give up on minimality and instead search for a *sufficiently small* reason without exploring the whole graph.

In order to achieve this, we make two observations. First, an edge being killed is most likely to have effects in the nearby levels; an edge at level j can kill a node at level $j + 2$ only if it is the final support to a node at level $j + 1$, which in turn remains the only support for a node at level $j + 2$. If we are searching for an explanation for the death of an edge, it is most likely to be near the edge being explained. This is particularly the case for constraints which are *local* in nature, where the possible values for x_j are most strongly constrained by the values of variables in nearby levels. Second, if the cause of the propagation is far from the killed node, this may indicate the presence of a narrow *cut* in the graph, which eliminates a large set of nodes. The goal of the incremental algorithm is to search the section of the graph where the explanation is likely to be, but follow chains of propagation to hopefully find any narrow cuts (which provide explanation for an entire subgraph).

Pseudo-code explaining the inference $var \neq val$ is given in Figure 6. Again the pseudo-code for explain_up is omitted since it is completely analogous to explain_down. The code makes use of function killed_below to check if a node has been killed from below. In practice, the results of this function are memoed to avoid recomputation. The function explain_down keeps track of pending nodes of the next level which may be required to be explained. We omit code to explain failure which is similar.

The algorithm first records the reason for the removal of each edge. We then traverse the graph from all the edges defining a removed value $var = val$ depending on how they were killed. For those killed from below we search breadth-first for edges below that were killed by domain reduction, whose endpoint was not also killed from below. They are added to the reason for the removal of $var = val$. We then traverse the edges which are not already part of the reason and add their child edges to check in the next level. Pending edges are edges whose end node may be required to be explained on the next level, but it can happen that before the current level is finished they are already explained. Hence the two pass approach.

A greedier algorithm which just tried to find a "close" reason why $var = val$ has been removed would stop the search whenever it reached a edge killed by domain reduction. On first sight this might seem to be preferable, as it traverses less of the MDD and gives a more "local" explanation. Our experiments showed two deficiencies: in many cases this killed edge may be redundant as it is explained

```
inc_explain(G, var, val) {
    kfa = {}; % edges killed from above
    kfb = {}; % edges killed from below
    for edge in G.edges(var, val) {
        % Split possible supports
        if killed(edge, above) {
            kfa ⊔={edge};
        } else {
            kfb ⊔={edge};
        }
    }
    % Explain all those killed from below
    return explain_down(kfb)
    % And all those killed from above
        ∪ explain_up(kfa);
}
```

```
killed_below(G,node) {
    % node is killed below if
    for edge in node.out_edges {
        s = G.status[edge]
        if s ∈ {alive, above}
            % No outgoing edge is alive
            % or killed from above
            return false;
    }
    return true;
}
```

```
explain_down(kfb) {
    reason = {};
    % Breadthfirst traverse the MDD downwards
    while ¬is_empty(kfb) {
        % Scan the current level for edges
        % that will need explaining.
        pending = {};
        for e in kfb {
            % For each edge requiring explanation
            if G.status[e] = dom and
                ¬killed_below(G, e.end) {
                % There is no later explanation,
                % so add (e.var, e.val) to the reason.
                reason ⊔={(e.var, e.val)};
            } else {
                pending ⊔={e};
            }
        }
        next = {};
        % Collect the edges that haven't been
        % explained at this level.
        for e in pending {
            if (e.var, e.val) ∉ reason
                % If e is not explained already
                % collect its outgoing edges
                next ⊔= e.end.out_edges;
        }
        % Continue with the next layer of edges.
        kfb = next;
    }
    return reason;
}
```

Fig. 6: Pseudo-code for explanation.

by other edges killed higher up that are still required to be part of the explanation for other reasons; and it failed to find "narrow cuts" in the MDD which lead to more reusable explanations.

Example 5 Consider the explanation of $x_0 \neq 0$ determined in Example 3. The edge $(x_0, 0, 1, 12)$ is marked as below, so it is added to *kfb*. In explain_down we add 12 as a pending node. We then insert $(x_1, 1, 12, 13)$ and $(x_1, 0, 12, 15)$ into *next* and restart the **while** loop. Nodes 13 and 15 become pending and in the next iteration of the **while** loop *kfb* is $\{(x_2, 1, 13, 14), (x_2, 1, 15, 16)\}$. The first edge is killed by domain and its end node 14 is not killed from below so we add $(x_2, 1)$ to reason. For the second edge node 16 is killed from below, so $(x_2, 1, 15, 16)$ is a pending edge. In the second **for** loop over *kfb* we determine that is already explained by *reason*. The algorithm terminates with the reason $\{(x_2, 1)\}$. This becomes the clause $\neg[\![x_2 = 1]\!] \rightarrow \neg[\![x_0 = 0]\!]$. □

Example 6 Unfortunately, these explanations are not guaranteed to be minimal. Consider again the constraint demonstrated in Example 3, but instead with $x_3 \neq 1$ fixed first, and $x_0 \neq 0$ fixed later. This kills nodes $\{15, 16\}$ from below, and nodes $\{5, 6, 12, 13, 14, 17\}$ killed from above. In order to explain $x_2 \neq 1$, we must determine reasons for the edges $(x_2, 1, 13, 14)$ and $(x_2, 1, 15, 16)$. Explaining $(x_2, 1, 13, 14)$

gives us $\{x_0 \neq 0\}$. As $(x_2, 1, 15, 16)$ was killed from below, we also add $x_3 \neq 1$ to the reason, even though $x_0 \neq 0$ already explains this edge.

The algorithm inc_explain is $O(|G|)$ for a single execution, no better than the non-incremental explanation in the worst case. However, if the constraint is reasonably local in nature, significantly fewer edges will be explored – if an explanation e contains variables V_e, the algorithm will explore at most those edges between $min(V_e)$ and $max(V_e)$.

4.3 Shortening Explanations for Large Domains

Both the non-incremental and incremental algorithms for MDD explanation collect explanations of the form $(\wedge\neg[\![x_i = v_{ij}]\!]) \rightarrow \neg[\![x = v]\!]$. These are guaranteed to be correct, and, in the non-incremental case, minimal. But they may be very large since for a single variable x_i with large initial domain $D(x_i)$ we may have up to $|D(x_i)| - 1$ literals involved.

A first simplification is to replace any subexpression $\wedge_{d \in D(x_i), d \neq d'} \neg[\![x_i = d]\!]$ by the equivalent expression $[\![x_i = d']\!]$. This serves to shorten explanation clauses considerably without weakening them. But it does not occur that commonly. A second simplification is to replace $\wedge_{d \in S} \neg[\![x_i = d]\!]$ by $\neg[\![x_i \leq l - 1]\!] \wedge [\![x_i \leq u]\!] \wedge \wedge_{d \in S \cap [\, l \,..\, u\,]} \neg[\![x_i = d]\!]$ where $l = \min(D(x_i) - S)$ and $u = \max(D(x_i) - S)$ are the least and greatest values of x_i consistent with the formula. Again this can sometimes shorten clauses considerably, but sometimes is of no benefit.

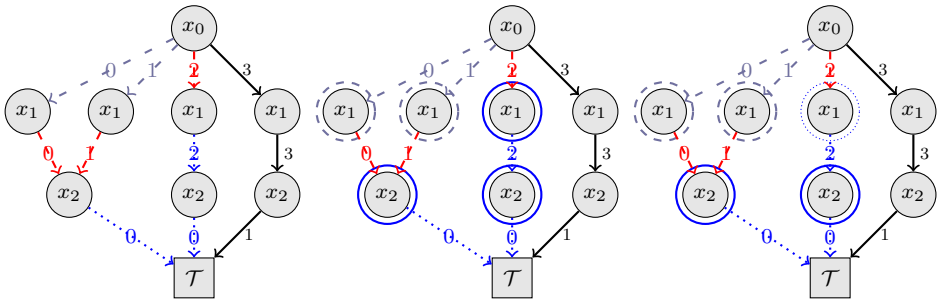
Finally we can choose to weaken the explanation. Suppose that in the current state $D(x_i) = d'$, that is x_i is fixed to d', then we can choose to replace $\wedge_{d \in S} \neg[\![x_i = d]\!]$, where $|S| > 1$ and $d' \notin S$ by $[\![x_i = d']\!]$. This shortens the explanation, but weakens it.

While we could perform this as a postprocess by first creating an explanation and then weakening it, doing so will make the explanations far from minimal. Hence we need to adjust the explanation algorithms so that as soon as they collect (x_i, e) and (x_i, e') in a reason, when in the current state $x_i = d'$ we in effect add all of $(x_i, e''), e'' \in D(x_i) - \{d'\}$ to the reason being generated (which will simplify to a single literal $[\![x_i = d']\!]$ in the explanation).

Example 7 Consider the MDD state shown in Figure 7(a) after the external inferences that $x_1 \neq 0$, $x_1 \neq 1$, and $x_0 \neq 2$. The two leftmost x_1 nodes are killed from below, while the third x_1 node is killed from above. In explaining the inference $x_2 \neq 0$, the incremental explanation algorithm starts at the edges to be explained, then collects $x_1 \neq 0$ and $x_1 \neq 1$ as values that must remain removed. Since the edge $x_1 = 2$ has not yet been explained, the algorithm continues, fixing $x_0 \neq 2$. We can then shorten this explanation to $x_1 = 3 \wedge x_0 \neq 2$. However, if we weaken the explanation during construction, we detect that $x_1 \neq 0 \wedge x_1 \neq 1$ can be weakened to $x_1 = 3$, which eliminates the remaining $x_1 \neq 2$ edge, giving us a final explanation of $x_1 = 3 \rightarrow x_2 \neq 0$. □

5 Experimental Results

Experiments were conducted on a 3.00GHz Core2 Duo with 2 Gb of RAM running Ubuntu GNU/Linux 8.10. Our solver is a modified version of MiniSAT2 (release



(a) Constraint (b) Without weakening (c) Inline weakening

Fig. 7: Explaining the inference $x_2 \neq 0$. The incremental explanation algorithm will generate the explanation $x_0 \neq 2 \wedge x_1 \neq 0 \wedge x_1 \neq 1$. This can then be shortened to $x_0 \neq 2 \wedge x_1 = 3$. If weakening is performed during explanation, $x_1 \neq 0 \wedge x_1 \neq 1$ will immediately be shortened, and the edge $x_0 = 2$ will never be reached, yielding the explanation $x_1 = 3$.

070721), augmented with MDD propagators. Explanations are constructed on demand during conflict analysis, and added to the clause database as learned clauses.

We compare a number of variations of our solver: base propagation and non-incremental explanation; ip is the incremental propagation approach described herein, with non-incremental explanation; weak is the incremental propagation approach with non-incremental explanation, using weakened explanations as described in Section 4.3. ipe is the incremental propagation approach with incremental explanation; and ipe_weak is the incremental propagation approach with incremental explanation using weakened explanations. All times are given in seconds.

5.1 Nonograms

Nonograms are a set of puzzles that have been studied both in terms of constraint programming, and in their own right, and a number of standalone solvers have been designed to solve these problems. A nonogram consists of an $n \times m$ matrix of blocks which may or may not be filled. Each row and column is marked with a sequence of numbers $[n_0, n_1, ..., n_k]$. This constraint indicates that there must be a sequence of n_0 filled squares, followed by one or more empty squares, followed by n_1 filled squares, and so on. Nonogram solvers are often used to assist puzzle design – rather than finding a single solution, a solver is used to determine uniqueness of a solution.

In all cases, the model is constructed introducing a Boolean variable for each square in the matrix, and converting each row and column constraint into a DFA, then expanding the DFA into a BDD.

An example nonogram is given in Figure 8a. The $[2, 2]$ next to the second row indicates that there must be a block of 2 filled blocks, followed by a gap, then another 2 filled squares. This constraint is converted into a DFA. The solution to this puzzle is given in Figure 8b.

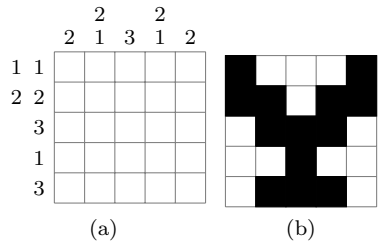


Fig. 8: (a) An example nonogram puzzle, (b) the corresponding solution.

Without learning							
Problem	PBNSOLVE		GECODE		Seq		
	time	fails	time	fails	base	ip	fails
1_dancer.inst	**0.0**	0	**0.0**	0	0.03	0.03	0
6_cat.inst	**0.0**	0	**0.0**	0	0.04	0.04	0
21_skid.inst	**0.0**	0	**0.0**	3	0.04	0.04	0
27_bucks.inst	**0.0**	2	**0.0**	9	0.06	0.06	2
23_edge.inst	**0.0**	22	**0.0**	25	0.03	0.03	26
2413_smoke.inst	**0.0**	7	**0.0**	8	0.05	0.05	9
16_knot.inst	**0.0**	0	**0.00**	0	0.08	0.08	0
529_swing.inst	**0.0**	0	0.01	0	0.16	0.15	0
65_mum.inst	**0.0**	22	0.01	22	0.10	0.10	23
1694_tragic.inst	**0.03**	193	0.11	255	0.23	0.18	256
1611_merka.inst	**0.00**	25	0.02	13	0.29	0.26	14
436_petro.inst	**0.07**	246	69.20	106919	34.54	15.89	106920
4645_m_and_m.inst	**0.07**	180	0.64	428	0.78	0.39	429
3541_signed.inst	**0.03**	146	8.44	6484	4.96	1.78	6485
803_light.inst	0.42	995	—	—	—	—	—
6574_forever.inst	3.94	147112	4.90	30900	2.88	1.47	30901
2040_hot.inst	0.90	2508	—	—	—	—	—
6739_karate.inst	0.90	9959	53.19	170355	33.80	13.15	170356
8098_domIII.inst	11.82	208689	—	—	366.96	247.61	8351050
2556_flag.inst	0.49	22184	3.02	16531	1.94	0.67	16532
2712_lion.inst	6.84	44214	—	—	—	—	—
$\sum$	25.51	436504	—	—	—	—	—

Table 1: Unique-solution performance results on hard nonogram instances from [24], using solvers without learning.

The nonogram puzzle instances are taken from [24], which compares 13 different solvers for the problem on a 2.6GHz AMD Phenom quad-core processor with 8Gb of memory. These solvers either find two distinct solutions, or prove that there is a unique solution. Only two are listed as solving all problems (PBNSolve and BGU) solving them in a total of 23.79s and 251.38s respectively. One solves all but one problem (Kjellerstrand/LazyFD), taking 427s to do so. The other solvers all fail to solve at least two of the instances within 30 minutes.

The results in Tables 1 and 2 compare various approaches: the best solver from [24] PBNSOLVE 1.09, GECODE 3.10, and our solvers. The tables show the average time (over 25 runs) in seconds and the number of failures in the search for each instance. The sums of each column are given in row $\sum$. Since the MDDs are BDDs in this case weakening (Section 4.3) is not applicable. Note that base

With learning										
Problem	Seq					VSIDS				
	base	ip	fails	ipe	fails	base	ip	fails	ipe	fails
1_dancer	0.03	0.03	0	0.03	0	0.03	0.03	0	0.03	0
6_cat	0.04	0.04	0	0.04	0	0.04	0.05	0	0.04	0
21_skid	0.04	0.05	0	0.05	0	0.04	0.05	0	0.04	0
27_bucks	0.06	0.06	2	0.06	2	0.06	0.06	2	0.06	2
23_edge	0.03	0.03	18	0.03	22	0.03	0.03	10	0.03	10
2413_smoke	0.05	0.05	7	0.05	8	0.05	0.05	4	0.04	3
16_knot	0.08	0.08	0	0.08	0	0.08	0.08	0	0.08	0
529_swing	0.15	0.15	0	0.16	0	0.16	0.15	0	0.16	0
65_mum	0.11	0.11	22	0.11	22	0.11	0.10	11	0.11	8
1694_tragic	0.20	0.20	141	0.18	123	0.22	0.20	107	0.18	102
1611_merka	0.29	0.27	13	0.28	10	0.29	0.28	11	0.27	11
436_petro	0.53	0.36	3068	0.54	5173	0.12	0.11	21	0.11	64
4645_m_and_m	0.40	0.30	130	0.28	128	0.39	0.30	146	0.28	131
3541_signed	0.40	0.31	337	0.31	425	0.69	0.45	531	0.29	292
803_light	0.37	0.23	1585	0.19	1064	0.12	**0.10**	40	**0.10**	39
6574_forever	0.07	**0.06**	207	0.07	258	0.07	**0.06**	128	0.07	199
2040_hot	1.08	0.73	4708	0.72	5527	0.48	0.36	221	**0.29**	141
6739_karate	0.66	0.48	4525	0.39	3717	0.16	0.15	150	**0.13**	92
8098_domIII	8.36	7.06	147444	6.14	130704	0.15	0.12	2089	**0.10**	1652
2556_flag	0.18	0.16	179	0.17	389	0.16	**0.15**	25	**0.15**	16
2712_lion	11.03	8.17	39193	6.89	29673	2.12	1.27	6940	**0.30**	898
$\sum$	24.16	18.93	201579	16.77	177245	5.57	4.15	10436	**2.86**	3660

Table 2: Unique-solution performance results on hard nonogram instances from [24] using sequential and VSIDS search strategies.

and ip perform exactly the same search. We use two search strategies: (Seq) filling in the matrix in order from left-to-right and top-to-bottom which is also used by GECODE; and (VSIDS) using activity based VSIDS search [25] which concentrates on the exploring decisions that have been most active in contributing to failure.

Table 1 compares the non-learning approaches. Note that VSIDS search is only applicable with learning since activity is derived from learning. The results here show that specialized code PBNSolve (which is not based on constraint programming) is highly competitive. Gecode and our approaches have the same sequential search. Clearly incremental propagation is advantageous over the base approach in terms of speed.

Table 2 compares our algorithms with learning. Clearly learning makes an enormous difference on these benchmarks. First, note that even base is competitive with the best reported solution. Next, the results show that incremental propagation is clearly beneficial, although it can increase search space because it creates non-minimal explanations.

Since the survey benchmarks were very easy for the MDD propagators we also experimented on a hard artificial class of nonogram problems *domino logic*, or n-Dom problems, described at [23]. These are very hard to solve, no solver on the website can solve instances beyond size $n = 16$. These instances are constructed from n identical rotated V shapes, illustrated in Figure 9. Comparative results are shown in Tables 3 and 4. As before, incremental propagation is beneficial, and learning is vital. The BDDs for these constraints are very narrow (between 2 and 6 nodes), so explanation generation accounts for only 1% of execution time;

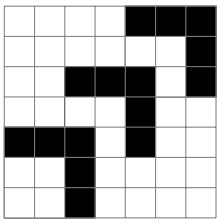


Fig. 9: An example of the *domino logic* problem set, with $n = 3$.

Without learning							
n	PBNSOLVE		GECODE		Seq		
	time	fails	time	fails	base	ip	fails
05	**0.00**	121	0.01	163	0.04	0.04	163
06	0.04	718	0.11	2K	0.10	0.08	2K
07	0.34	8K	1.70	29K	0.98	0.76	29K
08	2.14	30K	29.89	435K	16.59	11.88	435K
09	17.09	209K	—	—	362.63	247.50	8351K
10	169.89	1745K	—	—	—	—	—
11	—	—	—	—	—	—	—

Table 3: Unique solution performance results for non-learning solvers on *domino logic* nonograms. None of these solvers solve any problem of size greater than 10.

With learning										
n	Seq					VSIDS				
	base	ip	fails	ipe	fails	base	ip	fails	ipe	fails
5	0.03	0.02	60	0.04	76	0.05	0.03	34	0.04	41
6	0.04	0.04	328	0.04	341	0.04	**0.03**	135	0.04	108
7	0.10	0.08	2K	0.08	2K	**0.04**	0.05	330	**0.04**	345
8	0.48	0.38	12K	0.54	16K	**0.06**	**0.06**	733	**0.06**	670
9	8.37	7.04	147K	6.12	131K	0.14	0.14	2K	**0.10**	2K
10	91.63	79.42	1069K	87.96	1174K	0.33	0.23	5K	**0.15**	3K
11	—	—	—	—	—	0.53	0.39	7K	**0.28**	5K
12	—	—	—	—	—	1.38	0.98	15K	**0.70**	11K
13	—	—	—	—	—	2.33	1.68	24K	**1.06**	15K
14	—	—	—	—	—	5.59	3.88	51K	**2.44**	32K
15	—	—	—	—	—	7.86	5.40	65K	**3.14**	37K
16	—	—	—	—	—	18.03	12.40	123K	**5.76**	60K
17	—	—	—	—	—	68.32	50.48	382K	**12.38**	109K
18	—	—	—	—	—	101.31	74.19	500K	**30.83**	227K
19	—	—	—	—	—	118.16	83.57	539K	**65.66**	395K
20	—	—	—	—	—	384.99	293.43	1341K	**124.45**	606K
$\sum$	—	—	—	—	—	709.16	526.94	3056K	**247.13**	1503K

Table 4: Unique solution performance results on the *domino logic* nonogram instances using sequential and VSIDS search strategies.

differences between the execution time of ip and ipe are due to differences in search. While ipe seems to perform slightly worse than ip using a sequential search method, it appears to drive VSIDS consistently towards better search decisions, reducing time and backtracks by up to 50%.

5.2 Nurse Scheduling

The second set of experiments use nurse scheduling benchmarks from Section 6.2 of [4], where nurses are rostered to day shifts, evening shifts, night shifts and days off. In model 1, each nurse must work 1 or 2 night shifts in every 7 days, 1 or 2 evening shifts, 1 to 5 day shifts and 2 to 5 days off. In model 2, nurses must work 1 or 2 night shifts every 7 days, and 1 or 2 days off every 5 days (which makes day and evening shifts indistinguishable). In both models, a nurse cannot work a second shift within 12 hours of the first. The constraints are encoded as a single `regular` constraint per nurse and a `global_cardinality` [2] constraint per shift, converted to MDDs. We use the 50 instances of 28 day schedules used in [4] for each model with a 5 minute time limit, plus the next 50 instances from the N30 dataset, available at [22]. Results for both GECODE and the non-learning solvers are omitted, as they were unable to solve any instances in 5 minutes.

Tables 5 and 6 show the results on the nurse scheduling benchmarks using sequential search (assigning each nurse for day 1, then each nurse for day 2, etc.), first fail search (picking the nurse/day pair with the smallest remaining domain, breaking ties according to the sequential search), and VSIDS search. The first line is for direct comparison: it gives the average time and fails for problems solved by all solvers using the given search strategy. The second line gives number of problems solved by each solver and the average solving time and average failures for these solved problems. Comparing to the best results from [4] (which used first fail and a 100 second time limit) our `base` solver solves more instances (24 versus 9, and 32 versus 8).[1]

Comparing `base` versus `ip` we see that incremental propagation is usually beneficial. For the problems with more backtracking required on average we can see that the incremental propagation can be substantially faster than non-incremental propagation. Incremental explanation for these problems usually reduces the number of problems that can be solved.

The results show that weakening can be beneficial even with the very small domains of this benchmark. Weakening improves on almost all examples for model 2, except first fail with incremental explanation. Although it requires more search it is almost always faster and sometimes more robust.

5.3 Pentominoes

For another set of experiments we consider the pentomino problems, which involve placing a set of 5-block shapes in such a way as to fill a given area. The most common variant of the puzzle is to place 12 of these shapes, which may be rotated or reflected, inside a rectangle with an area of 60 units (one of $3 \times 20, 4 \times 15, 5 \times 12, 6 \times 10$). A model of the pentomino problems using the `regular` constraint is described in [13].

Tables 7 to 10 compare the performance of our explaining MDD propagators with the conventional `regular` constraints of [13] (implemented in GECODE 3.1.0). We tested both the finite domain and Boolean models of the pentomino problems,

[1] The experiments from [4] are run on a Pentium 4 3.20GHz machine with 1Gb RAM.

Search	base	fails	ip	fails	weak	fails
Seq	1.43	856.83	**0.83**	862.11	1.45	1277.83
	49 / 7.52	6059.65	**49 / 5.63**	6449.96	49 / 6.70	7212.41
FF	14.30	12735.22	**6.76**	13329.56	16.09	19907.58
	41 / 25.76	19191.93	**41 / 12.68**	19721.29	39 / 28.97	27757.59
VSIDS	7.67	4049.46	2.90	2468.26	4.98	3404.62
	73 / 10.84	5981.86	71 / 5.10	4879.04	73 / 10.95	6858.56

Search	ipe	fails	ipe_weak	fails
Seq	1.21	1191.04	3.98	2134.45
	47 / 1.21	1191.04	47 / 3.98	2134.45
FF	7.22	12436.06	13.65	13321.17
	38 / 10.78	14856.13	37 / 18.59	16945.57
VSIDS	**2.76**	3485.91	3.97	4121.40
	72 / 6.54	6647.18	**73 / 7.10**	6838.68

Table 5: Nurse sequencing, multi-sequence constraints, model 1.

Search	base	fails	ip	fails	weak	fails
Seq	0.64	490.53	0.43	527.36	0.44	291.08
	70 / 3.26	4956.57	70 / 1.85	5011.67	**70 / 1.31**	3021.19
FF	1.29	1015.11	0.58	890.89	**0.52**	806.81
	41 / 14.84	19189.46	**41 / 6.98**	15314.46	41 / 15.23	27240.39
VSIDS	1.07	1021.73	1.56	2877.52	0.66	750.35
	86 / 4.44	3421.27	87 / 6.02	8687.74	**87 / 4.10**	5821.41

Search	ipe	fails	ipe_weak	fails
Seq	**0.41**	508.11	0.53	774.44
	69 / 4.17	5669.68	69 / 5.70	10942.51
FF	0.54	1332.59	0.68	1378.43
	37 / 0.54	1332.59	39 / 7.14	20791.82
VSIDS	2.38	4574.18	**0.65**	1184.52
	84 / 2.38	4574.18	85 / 0.65	1186.12

Table 6: Nurse sequencing, multi-sequence constraints, model 2

using a time limit of 20 minutes. All approaches use the same search strategy as in [14] which is slightly different to the default strategy in the Gecode model.

First incremental propagation is always better than non-incremental explanation, and indeed the difference without learning is quite substantial. Next, we find that incremental explanations provide a significant improvement over non-incremental explanations. On the finite-domain model, weak explanations provide a significant improvement in propagation speed with relatively little increase in search – the combination of the two techniques is on all instances the best algorithm, and significantly outperforms the conventional `regular` constraints. While the base solver is not as good as GECODE on the FD model, once we add learning and weakening our results are substantially better. Note that all the variants of the learning solver have very similar failure counts; the difference in performance is due primarily to faster propagation and explanation algorithms.

Without learning					
Size	gecode	fails	base	ip	fails
3x20	19.87	36K	33.64	7.63	36K
4x15	338.60	649K	546.86	131.41	651K
5x12	—	—	—	519.81	2482K
6x10	—	—	—	—	—
$\sum$	—	—	—	—	—

With learning									
Size	base	ip	fails	weak	fails	ipe	fails	ipe_weak	fails
3x20	9.68	4.99	11K	4.02	10K	3.26	10K	**2.57**	9K
4x15	300.91	172.90	380K	165.41	382K	128.85	379K	**97.58**	353K
5x12	—	730.34	1571K	761.40	1657K	590.18	1656K	**497.88**	1586K
6x10	—	—	—	—	—	—	—	—	—
$\sum$	—	—	—	—	—	—	—	—	—

Table 7: Time to find all solutions for pentominoes, with FD model and no symmetry breaking.

Without learning					
Size	gecode	fails	base	ip	fails
3x20	5.22	9K	9.20	2.29	9K
4x15	131.78	239K	209.41	50.70	239K
5x12	464.75	788K	653.59	173.04	789K
6x10	—	—	—	447.97	1840K
$\sum$	—	—	—	674.00	2877K

With learning									
Size	base	ip	fails	weak	fails	ipe	fails	ipe_weak	fails
3x20	4.84	2.62	4K	2.34	4K	1.58	4K	**1.35**	4K
4x15	106.60	62.70	128K	49.64	116K	45.23	129K	**34.76**	121K
5x12	379.78	233.76	465K	205.10	444K	183.23	481K	**148.44**	448K
6x10	949.09	558.72	1176K	511.00	1106K	480.99	1235K	**390.51**	1111K
$\sum$	1440.31	857.80	1774K	768.08	1670K	711.03	1850K	**575.06**	1684K

Table 8: Time to find all solutions for pentominoes, with FD model and symmetries removed.

5.4 Other Problems

There are some circumstances, however, where learning is not particularly helpful. When solving the `crossword` instances used in [6], learning does not produce any re-usable nogoods at all; the number of bactracks of the learning solver is exactly the same as a purely propagation-based solver. As the MDDs for these constraints are quite *wide* – some propagators have an average width of ∼2700 nodes, rather than 10–30 for most of the `regular` constraints we have considered – and have very little sharing between nodes, explanation generation is very expensive, and constructs quite large explanations – ∼1000 literals/nogood, as opposed to 10–100. Since maintaining these nogoods is pure overhead, this results in the learning solver being an order of magnitude slower on some instances.

On another common table benchmark, the `Renault` problems [7], we have rather the opposite problem. Despite having a large extensional representation, the result-

Without learning					
Size	gecode	fails	base	ip	fails
3x20	5.32	36K	2.66	1.82	36K
4x15	88.68	649K	47.67	32.50	651K
5x12	346.42	2478K	185.73	127.98	2482K
6x10	907.59	5998K	477.58	325.84	6008K
$\sum$	1348.01	9161K	713.64	488.14	9176K

With learning					
Size	base	ip	fails	ipe	fails
3x20	1.00	0.78	9K	**0.72**	9K
4x15	41.17	33.41	398K	**28.27**	424K
5x12	176.49	144.59	1677K	**120.74**	1749K
6x10	437.29	353.48	4089K	**299.71**	4277K
$\sum$	655.95	532.26	6173K	**449.44**	6460K

Table 9: Time to find all solutions for pentominoes, with Boolean model and no symmetry breaking.

Without learning					
Size	gecode	fails	base	ip	fails
3x20	1.39	9K	0.96	0.58	9K
4x15	33.77	239K	22.46	13.40	239K
5x12	113.62	788K	72.96	45.26	789K
6x10	288.91	1838K	179.21	112.14	1840K
$\sum$	437.69	2874K	275.59	171.38	2877K

With learning					
Size	base	ip	fails	ipe	fails
3x20	0.51	0.41	4K	**0.33**	4K
4x15	13.87	10.38	117K	**9.65**	129K
5x12	52.02	41.03	442K	**36.14**	462K
6x10	128.40	100.56	1088K	**89.15**	1136K
$\sum$	194.80	152.38	1651K	**135.27**	1730K

Table 10: Time to find all solutions for pentominoes, with Boolean model and symmetries removed.

ing MDDs are small enough that the learning engine drives the solver immediately to a solution, irrespective of which explanation algorithm is used; without learning, however, the solver takes longer than 10 minutes.

6 Conclusion

In this paper we have defined an MDD propagation with explanation. We introduce an incremental propagation algorithm for MDDs using watches, and incremental approach to explaining propagation for MDD constraints. The incremental propagation algorithm is significantly better than approaches starting from the root, at least on the kind of MDDs with large arity and low width appearing in the problems we study. Incremental explanation often improves on non-incremental explanation particularly when using activity based search where the non-minimality of the resulting explanations is not so critical. The resulting system provides the state-of-the-art solution to nonogram puzzles.

Acknowledgments

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

1. Andersen, H., Hadzic, T., Hooker, J., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, *LNCS*, vol. 4741, pp. 118–132. Springer-Verlag (2007)
2. Beldiceanu, N., Carlsson, M., Rampon, J.: Global Constraint Catalog 2nd Edition. SICS Technical Report T2010:07, Swedish Institute of Computer Science (2010)
3. Bessiere, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: SLIDE: A useful special case of the CARDPATH constraint. In: 18th European Conference on Artificial Intelligence, pp. 475–479 (2008)
4. Brand, S., Narodytska, N., Quimper, C., Stuckey, P., Walsh, T.: Encodings of the SEQUENCE constraint. In: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, *LNCS*, vol. 4741, pp. 210–224 (2007)
5. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
6. Cheng, K., Yap, R.: Maintaining generalized arc consistency on ad hoc r-ary constraints. In: 14th International Conference on Principles and Process of Constraint Programming, *LNCS*, vol. 5202, pp. 509–523 (2008)
7. van Dongen, M., Lecoutre, C., Roussel, O.: Third International CSP Solver Competition. [Online, accessed Sept 2010]. http://www.cril.univ-artois.fr/CPAI08/
8. Gange, G., Lagoon, V., Stuckey, P.: Fast set bounds propagation using BDDs. In: 18th European Conference on Artificial Intelligence, pp. 505–509 (2008)
9. Hawkins, P., Lagoon, V., Stuckey, P.: Solving set constraint satisfaction problems using ROBDDs. Journal of Artificial Intelligence Research **24**, 109–156 (2005)
10. Hawkins, P., Stuckey, P.: A hybrid BDD and SAT finite domain constraint solver. In: Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages, *LNCS*, vol. 3819, pp. 103–117 (2006)
11. Hoda, S., van Hoeve, W., Hooker, J.: A systematic approach to MDD-based constraint programming. In: Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming, *LNCS*, vol. 6308, pp. 266–280 (2010)
12. van Hoeve, W., Pesant, G., Rousseau, L., Sabharwal, A.: Revisiting the sequence constraint. In: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming, *LNCS*, vol. 4204, pp. 620–634 (2006)
13. Lagerkvist, M.: Techniques for efficient constraint propagation. Ph.D. thesis, Kungliga Teniska; Stockholm (2008)
14. Lagerkvist, M., Pesant, G.: Modeling irregular shape placement problems with regular constraints. In: First Workshop on Bin Packing and Placement Constraints BPCC'08 (2008)
15. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: 38th Design Automation Conference, pp. 530–535 (2001)
16. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3), 357–391 (2009)
17. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: M. Wallace (ed.) Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, *LNCS*, vol. 3258, pp. 482–495. Springer-Verlag (2004)
18. Quimper, C., Walsh, T.: Global grammar constraints. In: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming, *LNCS*, vol. 4204, pp. 751–755 (2006)
19. Quimper, C., Walsh, T.: Decomposing global grammar constraints. In: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, *LNCS*, vol. 4741, pp. 590–604 (2007)

20. Srinivasan, A., Ham, T., Malik, S., Brayton, R.: Algorithms for discrete function manipulation. In: Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on, pp. 92–95 (1990). DOI 10.1109/ICCAD.1990.129849
21. Subbarayan, S.: Efficent reasoning for nogoods in constraint solvers with BDDs. In: Proceedings of Tenth International Symposium on Practical Aspects of Declarative Languages, *LNCS*, vol. 4902, pp. 53–57 (2008)
22. Vanhoucke, M., Maenhout, B.: The nurse scheduling problem (NSP). [Online, accessed Oct 2010]. `http://www.projectmanagement.ugent.be/nsp.php`
23. Wolter, J.: Comparison of solvers on the n-dom puzzles. [Online, accessed Sept 2010]. `http://webpbn.com/survey/dom.html`
24. Wolter, J.: Survey of paint-by-numbers puzzle solvers. [Online, accessed Sept 2010]. `http://webpbn.com/survey/`
25. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, pp. 279–285 (2001)