Explaining Propagators for s-DNNF Circuits

Graeme Gange[2] and Peter J. Stuckey[1,2]

[1]National ICT Australia, Victoria Laboratory
[2]Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{ggange,pjs}@csse.unimelb.edu.au

Abstract. Smooth decomposable negation normal form (s-DNNF) circuits are a compact form of representing many Boolean functions, that permit linear time satisfiability checking. Given a constraint defined by an s-DNNF circuit, we can create a propagator for the constraint by decomposing the circuit using a Tseitin transformation. But this introduces many additional Boolean variables, and hides the structure of the original s-DNNF. In this paper we show how we can build a propagator that works on the s-DNNF circuit directly, and can be integrated into a lazy-clause generation-based constraint solver. We show that the resulting propagator can efficiently solve problems where s-DNNF circuits are the natural representation of the constraints of the problem, outperforming the decomposition based approach.

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 weak propagation.

Previous work has explored the use of *Boolean Decision Diagrams* (BDDs) [1, 2] and *Multi-valued Decision Diagrams* (MDDs) [3] for automatically constructing efficient global propagators. However, in the absence of a sequential structure (such as is present in `regular` constraints [4]), (B/M)DDs can require exponential space to encode a function.

In such cases, it may be convenient to construct a propagator from a less restricted representation, but one which still permits efficient propagation and explanation. *Smooth, decomposable negation normal form* (s-DNNF) appears to be a suitable representation, as it allows for polynomial representation of a larger class of functions (most notably including context-free languages), while still permitting linear-time satisfiability checking. Given the recent development of *sentential decision diagrams* [5], which can be automatically constructed in a similar fashion to BDDs, it seems likely that s-DNNF representations will be increasingly convenient. In this paper we investigate how to construct propagators for `s-DNNF` circuits, and compare it with the only existing approach we are aware of for handling such circuits in constraint programming systems, decomposing the circuits using a form of Tseitin transformation [6].

2 Propagating DNNF

Constraint programming solves constraint satisfaction problems by interleaving propagation, which remove impossible values of variables from the domain, with search, which guesses values. All propagators are repeatedly executed until no change in domain is possible, then a new search decision is made. If propagation determines there is no solution then search undoes the last decision and replaces it with the opposite choice. If all variables are fixed then the system has found a solution to the problem. For more details see e.g. [7].

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 initial finite set of values or *domain* $D_{init}(x)$. The domain D keeps track of the current set of possible values $D(x)$ for a 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 $f(D) \sqsubseteq D$.

We make use of constraint programming with learning using the lazy clause generation [8] approach. Learning keeps track of what caused changes in domain to occur, and on failure records a *nogood* which records the reason for failure. The nogood prevents search making the same incorrect set of decisions.

In a lazy clause generation solver integer domains are also represented using Boolean variables. Each variable x with initial domain $D_{init}(x) = [l..u]$ 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 $D(x)$. We use $[\![x \neq d]\!]$ as shorthand for $\neg\, [\![x = d]\!]$. A lazy clause generation solver keeps the two representations of the domain in sync. 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.

Example 1. Consider a simple constraint satisfaction problem with constraints $b \leftrightarrow x + y \leq 2$, $x + y \leq 2$, $b' \leftrightarrow x \leq 1$, $b \rightarrow b'$, with initial domains $D_{init}(b) = D_{init}(b') = \{0, 1\}$, and $D_{init}(x) = D_{init}(y) = \{0, 1, 2\}$. There is no initial propagation. Setting $x = 2$ makes the third constraint propagate $D(b') = \{0\}$ with explanation $x = 2 \rightarrow b' = 0$, this makes the last constraint propagate $D(b) = \{0\}$ with explanation $b' = 0 \rightarrow b = 0$. The first constraint propagates that $D(y) = \{1, 2\}$ with explanation $b = 0 \rightarrow y \neq 0$ and the second constraint determines failure with explanation $x = 2 \wedge y \neq 0 \rightarrow \mathit{false}$. The graph of the implications is

$$\begin{tikzcd}
 & b' = 0 \arrow[r] & b = 0 \arrow[r] & y \neq 0 \arrow[dr] & \\
x = 2 \arrow[ur] \arrow[rrrr] & & & & \mathit{false}
\end{tikzcd}$$

Any cut separating the decision $x = 2$ from *false* gives a nogood. The simplest one is $x = 2 \rightarrow \mathit{false}$ or equivalently $x \neq 2$. □

2.1 Smooth Decomposable Negation Normal Form

A circuit in *Negation Normal Form (NNF)* is a propositional formula using connectives $\{\wedge, \vee, \neg\}$, such that $\neg$ is only applied to variables. While NNF normally defines functions over Boolean variables, it can be readily adapted to non-binary domains by permitting leaves of the form $[\![x_i = v_j]\!]$ for each value in $D(x_i)$. Hence a Boolean variable b is represented by leaves $[\![b = 0]\!]$ and $[\![b = 1]\!]$ corresponding directly to $\neg b$ and b. As we are concerned with constraints over finite-domain variables, we consider circuits in this class of *valued* NNF [9].

The rest of the presentation ignores bounds literals $[\![x_i \leq v_j]\!]$. We can extend the algorithms herein to directly support bounds literals $[\![x_i \leq v_j]\!]$ but it considerably complicates their presentation. They (and their negations) can of course be represented with disjunctive nodes e.g. $\left[\!\left[\vee_{v' \leq v_j} [\![x_i = v']\!]\right]\!\right]$.

We shall use vars to denote the set of variables involved in an NNF circuit, defined as:

$$\begin{aligned}
\mathsf{vars}([\![x_i = v_j]\!]) &= \{x_i\} \\
\mathsf{vars}([\![\bigvee N]\!]) &= \textstyle\bigcup_{n' \in N} \mathsf{vars}(n') \\
\mathsf{vars}([\![\bigwedge N]\!]) &= \textstyle\bigcup_{n' \in N} \mathsf{vars}(n')
\end{aligned}$$

It is difficult to analyse NNF circuits in the general case – even determining satisfiability is NP-hard. However, restricted subclasses of NNF, described in [10], permit more efficient analysis. In this paper, we are concerned with *decomposability* and *smoothness*.

Decomposability requires that for any node of the form $\phi = [\![\bigwedge N]\!]$, any two children n_i, n_j must satisfy $\mathsf{vars}(n_i) \cap \mathsf{vars}(n_j) = \emptyset$ – that is, children of a conjunction cannot have any shared dependencies. Similarly, *smoothness* requires that for any node $\phi = [\![\bigvee N]\!]$, any two children n_i, n_j must satisfy $\mathsf{vars}(n_i) = \mathsf{vars}(n_j)$.

Smooth Decomposable Negation Normal Form (`s-DNNF`) is the set of circuits of the form

$$\begin{aligned}
\phi \;\rightarrow\; & [\![x_i = v_j]\!] \\
\mid\; & [\![\bigvee N]\!] \ \textbf{iff}\ \forall_{n_i, n_j \in N, n_i \neq n_j}\ \mathsf{vars}(n_i) = \mathsf{vars}(n_j) \\
\mid\; & [\![\bigwedge N]\!] \ \textbf{iff}\ \forall_{n_i, n_j \in N, n_i \neq n_j}\ \mathsf{vars}(n_i) \cap \mathsf{vars}(n_j) = \emptyset
\end{aligned}$$

We represent a `s-DNNF` circuit as a graph G with literal leaves, and-nodes and or-nodes with children their subformulae. We assume $G.root$ is the root of the graph and $n.parents$ are the parent nodes of a node n.

Example 2. An `s-DNNF` for the constraint $b \leftrightarrow x + y \leq 2$ where $D_{init}(b) = \{0, 1\}$ and $D_{init}(x) = D_{init}(y) = \{0, 1, 2\}$ is shown in Figure 1. Ignore the different styles of edges for now. It is smooth, e.g. all of nodes 9,10,11,12,13 have $\mathsf{vars} =$

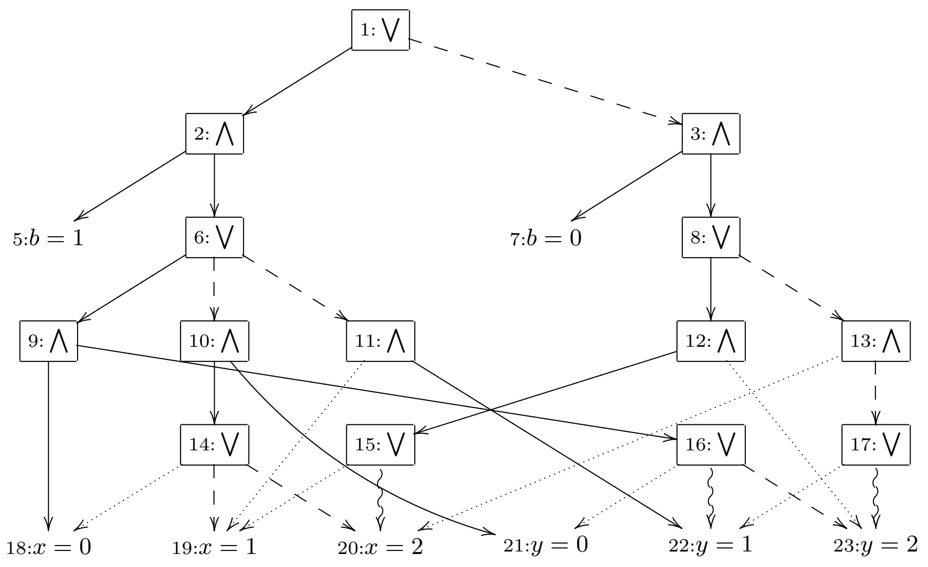

Fig. 1: An example `s-DNNF` graph for $b \leftrightarrow x + y \leq 2$

.

$\{x, y\}$, and it is decomposable, e.g. for each such node the left child has $\mathsf{vars} = \{x\}$ and the right child has $\mathsf{vars} = \{y\}$. □

2.2 DNNF Decomposition

Previous methods for working with these constraints (implicitly in [11] and explicitly in [12]) transformed the circuit into a set of clauses by introducing a new Boolean variable for each node.

For each node $n = [\![\mathbf{op}\ N]\!]$, we introduce a new Boolean variable $\langle n \rangle$. We then introduce the following clauses

$$[\![\bigvee N]\!] : \neg\langle n \rangle \vee \bigvee_{n_i \in N} \langle n_i \rangle \qquad [\![\bigwedge N]\!] : \bigwedge_{n_i \in N} \neg\langle n \rangle \vee \langle n_i \rangle$$

and set the variable $\langle G.root \rangle$ to true.

For the domain consistent encoding, we also introduce for $n \in N \setminus \{G.root\}$:

$$\bigvee_{n_i \in n.parents} \langle n_i \rangle \vee \neg\langle n \rangle$$

Example 3. Consider the graph shown in Figure 1. The clauses generated by the decomposition of this constraint are shown in Table 1. $\mathtt{decomp_{tt}}$ gives the clauses generated by the basic encoding. $\mathtt{decomp_{dc}}$ gives the additional clauses produced by the domain-consistent encoding. □

	$\texttt{decomp}_{\texttt{tt}}$	$\texttt{decomp}_{\texttt{dc}}$
	$\{\{\langle 1\rangle\}\}$	
1:	$\{\{\neg\langle 1\rangle, \langle 2\rangle, \langle 3\rangle\}\}$	$\emptyset$
2:	$\{\{\neg\langle 2\rangle, [\![b = 1]\!]\}, \{\neg\langle 2\rangle, \langle 6\rangle\}\}$	$\{\{\neg\langle 2\rangle, \langle 1\rangle\}\}$
3:	$\{\{\neg\langle 3\rangle, [\![b = 0]\!]\}, \{\neg\langle 3\rangle, \langle 8\rangle\}\}$	$\{\{\neg\langle 3\rangle, \langle 1\rangle\}\}$
5:	$\emptyset$	$\{\{\neg [\![b = 1]\!], \langle 2\rangle\}\}$
6:	$\{\{\neg\langle 6\rangle, \langle 9\rangle, \langle 10\rangle, \langle 11\rangle\}\}$	$\{\{\neg\langle 6\rangle, \langle 2\rangle\}\}$
7:	$\emptyset$	$\{\{\neg [\![b = 0]\!], \langle 3\rangle\}\}$
8:	$\{\{\neg\langle 6\rangle, \langle 9\rangle, \langle 10\rangle, \langle 11\rangle\}\}$	$\{\{\neg\langle 6\rangle, \langle 2\rangle\}\}$
	...	...
18:	$\emptyset$	$\{\{\neg [\![x = 0]\!], \langle 9\rangle, \langle 14\rangle\}\}$
	...	...

Table 1: Clauses produced by the decomposition of the graph in Fig. 1

3 DNNF Propagation

3.1 Propagation from the root

Consider an `s-DNNF` circuit G over variables X. propagate(G, X, D) enforces domain consistency over G with domain D. It consists of three stages. First, it determines which nodes may be both true and reachable from $G.root$ under the current partial assignment. If the root is not possibly true the propagator fails, there are no solutions. Second it collects in *supported* which literals $[\![x_i = v_j]\!]$ participate in solutions by being part of nodes that are true and reachable. Third, it propagates that any unsupported literals must be false.

```
Algorithm 1: propagate(G, X, D)
cache = $\emptyset$;
reachable = prop_mark($G.root$, cache, $D$);
if $\neg$reachable then return false;
supported = prop_collect($G.root$, cache);
for $x_i \in X$ do
    for $v_j \in D(x_i)$ do
        if $[\![x_i = v_j]\!] \notin$ supported then enqueue($[\![x_i \neq v_j]\!]$);
return true;
```

Marking the reachable nodes (prop_mark) simply traverses the `s-DNNF` circuit marking which nodes are reachable, and storing in a cache whether they may be true (*alive*) given the current domain D. Each node is visited at most once.

Collecting the literals (prop_collect) that appear in solutions simply traverses the `s-DNNF` circuit restricted to the nodes which are reachable and true, and returns all literals encountered. Each true node is visited at most once.

As each node and outgoing edge is expanded at most once on each pass, propagate runs in $O(|G|)$ time.

```
prop_mark($node, cache, D$)
    if $(node, S) \in cache$ then
        return $S$;
    case $node$ of
        $[\![x_i = v_j]\!]$ :
            $alive = (v_j \in D(x_i))$;
        $[\![\bigwedge N]\!]$ : $alive = \bigwedge_{n' \in N}$ prop_mark($n', cache$);
        $[\![\bigvee N]\!]$ : $alive = \bigvee_{n' \in N}$ prop_mark($n', cache$);
    $cache = cache \cup \{(node, alive)\}$;
    return $alive$;
```

```
prop_collect($node, cache$)
    if $(node, \mathsf{true}) \in cache$ then
        $cache = cache \setminus \{(node, \mathsf{true})\}$;
        case $node$ of
            $[\![x_i = v_j]\!]$ :
                $supported = \{[\![x_i = v_j]\!]\}$;
            $[\![\bigwedge N]\!]$ : $supported = \bigcup_{n' \in N}$ prop_collect($n', cache$);
            $[\![\bigvee N]\!]$ : $supported = \bigcup_{n' \in N}$ prop_collect($n', cache$);
        return $supported$;
    else return $\emptyset$;
```

Example 4. Imagine we are propagating the `s-DNNF` shown in Figure 1 when $D(b) = \{0\}$ and $D(x) = \{2\}$. The marking stage marks nodes $\{5, 2, 18, 9, 19, 11\}$ as dead and the rest alive. The collection visits nodes 1, 3, 7, 8, 12, 13, 15, 17, 20, 22, 23 and collects $b = 0$, $x = 2$, $y = 1$ and $y = 2$. Propagation determines that $y \neq 0$. □

3.2 Incremental propagation

Propagation from the root can be expensive, it must traverse the entire `s-DNNF` circuit each time the propagator executes. In many cases very little will have changed since the last time the propagator was executed. We can instead keep track of which nodes in the `s-DNNF` circuit are reachable and possibly true by just examining the part of the circuit which may have changed starting from leaves which have changed.

inc_propagate($changes, G, X$) propagates the `s-DNNF` circuit G over variables X given change to domains $changes$ which are literals $[\![x_i = v_j]\!]$ that have become false since the last propagation. The algorithms maintains for each node whether it is dead: unreachable or false with the current domain, and for each node which parents rely on it to keep them possibly true ($node.watched\_parents$) and for each node which children rely on this node to keep them reachable ($node.watched\_children$). In the first phase the algorithm visits a set of nodes kbQ which are "killed from below", i.e. became false because of leaf information. And nodes are killed from below if one of their children becomes killed, while or-nodes are killed from below if all their children are killed. The first phase also records killed and-nodes in kaQ ("killed from above") since we have to mark their other children as possibly unreachable. If the root is killed from below the propagator returns failure.

The second phase visits nodes in kaQ and determines if they kill child nodes since they are the last alive parent, in which case the child node is added to kaQ. A killed literal node ensures that we propagate the negation of the literal.

During propagator construction, $watched\_parents$ and $watched\_children$ are initialised to $\emptyset$. For each node n, we then pick one parent p and add n to $p.watched\_children$ – so p is now supporting n from above. For or-nodes, we then pick one child c, and add n to $c.watched\_parents$ – since n is satisfiable so long as any child is alive, it must be satisfiable so long as c is not killed. In the case of an and node, however, we must add n to $watched\_parents$ of *each* of its children, as n must be killed if any children die.

When a node is killed, in the worst case we must check all adjacent nodes n' to determine if there are remaining watches – this happens at most once per edge, so $O(|G|)$ times down a branch of the search tree. With a suitable implementation of watches, this only scans potential watches once down a branch. Since each node is killed at most once, and each edge is checked as a watch at most twice (once supporting above, once below), inc_propagate runs in $O(|G|)$ down a branch.

Example 5. Imagine we are propagating the `s-DNNF` graph of Figure 1. The policy for initial watches is illustrated in Figure 1, where edges for initially watched parents are solid or dotted, and edges for initially watched children are solid or dashed.

Suppose we set $D(x) = \{2\}$. The changes are $[\![x = 0]\!]$ and $[\![x = 1]\!]$. Initially $kbQ = \{18, 19\}$. Then 9 is added to kbQ and kaQ with killing child 18, and similarly 11 is added to kbQ and kaQ with killing child 19. Because 6 is a watched parent of 9, it is examined and the watched parents of 10 is set to 6. In the second phase examining node 9 we set 16 as dead and add it to kaQ. Examining 11 we look at its child 22 and set node 16s watched children to include 22. Examining node 16 we set 10s watched children to include 21, and 17s watched children to include 22. No propagation occurs.

Now suppose we set $D(b) = \{0\}$. The changes are $[\![b = 1]\!]$. Initially $kbQ = \{5\}$ and this causes 2 to be added to kbQ and kaQ and the killing child set to 5. Examining 2 causes the watched parent of 3 to be set to 1. In the second phase examining 2 causes 6 to be added to kaQ, which causes 10 to be added to kaQ, which causes 14 and 21 to be added to kaQ. Examining 14 adds 20 to the watched children on 15. Examining 21 we propagate that $y \neq 0$. □

4 Explaining DNNF 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 [13, 2] and MDD-based [3] constraint

```
inc_propagate($changes, G, X$)
  $kbQ = changes$;
  $kaQ = \emptyset$;
  // Handle nodes that were killed due to dead children.
  for $node \in kbQ$ do
      for $parent \in node.watched\_parents$ do
          case $parent$ of
              $[\![\bigwedge N]\!]$ :
                  if dead[$parent$] then continue;
                  $dead[parent]$ = true;
                  $parent.killing\_child = node$;      // For greedy explanation.
                  $kbQ = kbQ \cup \{parent\}$;
                  $kaQ = kaQ \cup \{parent\}$;          // Handle other children.
              $[\![\bigvee N]\!]$ :
                  if $\exists\ n' \in N$ s.t. $\neg$dead[$n'$] then
                      // A support still remains -- update the watches.
                      $node.watched\_parents = node.watched\_parents \setminus \{parent\}$;
                      $n'.watched\_parents = n'.watched\_parents \cup \{parent\}$;
                  else
                      // No supports -- kill the node.
                      $dead[parent]$ = true;
                      $parent.killed\_above$ = false;
                      $kbQ = kbQ \cup \{parent\}$;
  if $G.root \in kbQ$ then return $false$;
  // Downward pass
  for $node \in kaQ$ do
      case $node$ of
          $[\![x_i = v_j]\!]$ :
              enqueue($[\![x_i \neq v_j]\!]$);
              continue;
      for $child \in node.watched\_children$ do
          if $\exists\ n' \in child.parents$ s.t. $\neg$dead[$n'$] then
              $node.watched\_children = node.watched\_children \setminus \{child\}$;
              $n'.watched\_children = children \cup \{child\}$;
          else
              $dead[child]$ = true;
              $kaQ = kaQ \cup \{child\}$;
              $child.killed\_above$ = true;
  return $true$;
```

solvers. In order to be incorporated in a nogood learning solver, the s-DNNF propagator 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 as needed for nogood construction. For more details on conflict generation we refer the reader to [8].

4.1 Minimal Explanation

The explanation algorithm is similar in concept to that used for BDDs and MDDs. To explain $[\![x \neq v]\!]$ we assume $[\![x = v]\!]$ and hence make the `s-DNNF` unsatisfiable. A correct explanation is (the negation of) all the values for other variables which are currently false ($varQ$). We then progressively remove assignments (unfix literals) from this explanation while ensuring the constraint as a whole remains unsatisfiable. We are guaranteed to create *a minimal explanation* (but not the smallest minimal explanation) $\bigwedge_{l \in expln} \neg l \rightarrow [\![x \neq v]\!]$ since removing any literal l' from the $expln$ would mean $G \wedge \bigwedge_{l \in expln - \{l'\}} \neg l \wedge x = v$ is satisfiable. Constructing a smallest minimal explanation for a `s-DNNF` is NP-hard just as for BDDs [14].

Unlike (B/M)DDs, `s-DNNF` circuits do not have a global variable ordering that can be exploited. As such, we must update the reachability information as we progressively unfix leaf nodes. A node n is considered *binding* if n becoming satisfiable would make the root r satisfiable. $locks[n]$ denotes the number of dead children holding n dead. And nodes $[\![\bigwedge N]\!]$ start with $|N|$ locks while other nodes have 1. If n is *binding* and $locks[n] = 1$, then making any children satisfiable will render r satisfiable.

The explain algorithm initialises locks and then unlocks all nodes which are true for variables other than in the explained literal $[\![x = v]\!]$, and unlocks the explained literal. This represents the state of the current domain D except that we set $D(x) = \{v\}$. All nodes which may be true with the explained literal true will have 0 locks. The algorithm then marks the root as binding using set_binding. If the locks on the node are 1, then set_binding marks any locked children as also binding. The algorithm then examines each literal in $varQ$. If the literal is binding then clearly setting it to true will allow the root to become true, hence it must remain in the explanation. If not it can be removed from the explanation. We unfix the literal or equivalently unlock the node. We chain unlocking up as nodes reach zero locks, we unlock their parent nodes. Any node with just one lock which is binding, then makes its locked children binding.

The procedure init_locks processes the graph in $O(|G|)$ time. The body of set_binding is run at most once per node, so costs $O(|G|)$ over all nodes. Similarly, unlock may be called from each incoming edge, but the body of the loop is run only once for each node. Since each component runs in $O(|G|)$ overall, the algorithm explain is also $O(|G|)$.

Example 6. To create a minimal explanation for the propagation of $y \neq 0$ of Example 4 we initialize the locks using init_locks which sets the locks to 2 for each and node, and 1 for each other node. We unlock the literals which are in the current domain, for variables other that y, that is $b = 0$ and $x = 2$. Unlocking $b = 0$ reduces the locks on 7 to 0, and hence unlocks 3, reducing its locks to 1. Unlocking $x = 2$ reduces the locks on 13 to 1, and 14 and 15 to 0. Unlocking

```
explain($\neg [\![x = v]\!], G, X, D$)
    init_locks(G) ;
    for $x_i \in X \setminus \{x\}, v_j \in D(x_i)$ do unlock($[\![x_i = v_j]\!]$);
    unlock($[\![x = v]\!]$);
    set_binding($G.root$);
    $expln = \emptyset$;
    $varQ = \{[\![x_i = v_j]\!] \mid x_i \in X \setminus \{x\}, v_j \notin D(x_i)\}$;
    for $[\![x_i = v_j]\!] \in varQ$ do
        if $binding[[\![x_i = v_j]\!]]$ then $expln = expln \cup \{[\![x_i = v_j]\!]\}$;
        else unlock($node$);
    return $expln$;
```

```
init_locks(G)
    for $node \in G$ do
        case $node$ of
            $[\![\bigwedge N]\!]$ :
                $locks[node] = |N|$;
            $[\![\bigvee N]\!]$ :
                $locks[node] = 1$;
            $[\![x_i = v_j]\!]$ :
                $locks[node] = 1$;
        $binding[node] =$ false;
```

```
unlock(node)
    if $locks[node] = 0$ then return;
    $locks[node]$ −= 1;
    if $locks[node] = 0$ then
        for $parent \in node.parents$ do
            unlock($parent$);
    else if $locks[node] == 1 \wedge binding[node]$
    then
        for
        $n' \in node.children$ s.t. $locks[n'] > 0$
        do set_binding($n'$);
```

14 and 15 reduces the locks on 10 and 12 to 1. We then unlock the propagated literal $y = 0$. This reduces the locks on 10 and 16 to 0. Unlocking 16 reduces the locks on 9 to 1. Unlocking 10 causes 6 to unlock which reduces the locks on 2 to 1. We now set the root as binding. Since it has 1 lock we set its children 2 and 3 as binding. Since node 2 has one lock, binding it sets the child 5 as binding, but not 6 (since it has zero locks). Binding 3 has no further effect. Finally traversing $varQ = \{[\![b = 1]\!], [\![x = 0]\!], [\![x = 1]\!]\}$ adds $[\![b = 1]\!]$ to the explanation since it is binding. Since $x = 0$ is not binding it is unlocked, which unlocks 9. Since $x = 1$ is not binding it is unlocked, which sets the locks of 11 to 1 but has no further effect. The explanation is $b \neq 1 \rightarrow y \neq 0$ is minimal. □

4.2 Greedy Explanation

Unfortunately, on large circuits, constructing a minimal explanation can be expensive. For these cases, we present a greedy algorithm for constructing valid, but not necessarily minimal, explanations.

This algorithm is shown as greedy_explain. It relies on additional information recorded during execution of inc_prop to record the cause of a node's death, and operates by following the chain of these actions to construct an explanation.

```
set_binding(node)
  if $binding[node]$ then return;
  $binding[node]$ = true;
  case $node$ of
    $[\![\mathbf{op}\ N]\!]$ :
      if $locks[node] == 1$ then
        for $n' \in N$ s.t. $locks[n'] > 0$ do set_binding($n'$);
```

```
greedy_explain(($x \neq v$), G, X)
  $explQ = [\![x = v]\!].parents$;
  $expln = \emptyset$;
  for $node \in explQ$ do
    if $node.killed\_above$ then $explQ = explQ \cup node.parents$;
    else
      case $node$ of
        $[\![x = v]\!]$ : $expln = expln \cup \{[\![x = v]\!]\}$;
        $[\![\bigwedge N]\!]$ : $explQ = explQ \cup \{node.killing\_child\}$;
        $[\![\bigvee N]\!]$ : $explQ = explQ \cup N$;
  return $expln$
```

node.killed_above indicates whether the node was killed by death of parents – if true, we add the node's parents to the set of nodes to be explained; otherwise, we add one (in the case of conjunction) or all (for disjunction) children to the explanation queue. If a node n is a conjunction that was killed due to the death of a child, $n.killing\_child$ indicates the child that killed node n – upon explanation, we add this node to the explanation queue.

In the worst case, this still takes $O(|G|)$ time per execution (if it needs to explore the entire graph to construct the explanation), but even in this case it only needs to make a single pass over the graph. In practice, however, it only needs to explore a small section of the graph to construct a correct explanation.

Example 7. Explaining the propagation $y \neq 0$ of Example 5 proceeds as follows. Initially $explQ = \{10, 16\}$. Since 10 was killed from above we add 6 to $explQ$, similarly 16 adds 9. Examining 6 we add 2 since it was killed from above. Examining 9 we add $x = 0$ to $expln$ as the killing child. Examining 2 we add $b = 1$ to $expln$ as the killing child. The explanation is $b \neq 1 \wedge x \neq 0 \rightarrow y \neq 0$. This is clearly not minimal. □.

Whether minimal or greedy explanation is preferable varies depending on the circuit. On small circuits, the cost of minimal explanation is sufficiently cheap that the reduction in search outweighs the explanation cost – on larger graphs, the cost of explanation dominates.

4.3 Explanation Weakening

Explanations derived from `s-DNNF` circuits can often be very large. This causes overhead in storage and propagation. It can be worthwhile to weaken the explanation in order to make it shorter. This also can help direct propagation down the same paths and hence give more reusable nogoods. Conversely the weaker nogood may be less reusable since it is not as strong.

We can shorten an explanation $\wedge L \rightarrow l$ as follows. Suppose there are at least two literals $\{[\![x_i \neq v]\!], [\![x_i \neq v']\!]\} \subseteq L$. Suppose also that at the time of explanation $D(x_i) = \{v''\}$ (where clearly $v'' \neq v$ and $v'' \neq v'$). We can replace all literals about x_i in L by the literal $[\![x_i = v'']\!]$. This shortens the explanation, but weakens it.

For greedy explanation, we perform weakening as a postprocess. However for minimal explanation, weakening as a postprocess can result in explanations that are far from minimal. Hence we need to adjust the explanation algorithm so that for a variable x_i, we first count the number of nodes $[\![x_i = v_j]\!]$ that are binding. If in the current state $D(x_i) = \{v'\}$ and there are at least 2 binding nodes we add $[\![x_i = v']\!]$ to the explanation and progress to x_{i+1}; otherwise, we process the nodes as usual.

5 Experimental Results

Experiments were conducted on a 3.00GHz Core2 Duo with 2 Gb of RAM running Ubuntu GNU/Linux 8.10. The propagators were implemented in `chuffed`, a state-of-the-art lazy-clause generation [8] based constraint solver. All experiments were run with a 1 hour time limit.

We consider two problems that involve grammar constraints that can be expressed using `s-DNNF` circuits. For the experiments, `decomp` denotes propagation using the domain consistent decomposition described in Section 2.2 (which was slightly better than the simpler decomposition), `full` denotes propagation from the root and minimal explanations, `ip` denotes incremental propagation and minimal explanations, `+g` denotes greedy explanations and `+w` denotes explanation weakening.

Note that while `full` and `ip` generate the same inferences, the order of propagation differs, which causes different explanations to be generated and search to diverge.

5.1 Shift Scheduling

Shift scheduling, a problem introduced in [15], allocates n workers to shifts such that (a) each of k activities has a minimum number of workers scheduled at any given time, and (b) the overall cost of the schedule is minimised, without violating any of the additional constraints:

- An employee must work on a task (A_i) for at least one hour, and cannot switch tasks without a break (b).

Table 2: Comparison of different methods on shift scheduling problems.

Inst.	`decomp`		`full`		`ip`		`ip+w`		`ip+g`		`ip+gw`	
	time	fails	time	fails	time	fails	time	fails	time	fails	time	fails
1,2,4	9.58	21284	17.38	28603	6.89	18041	9.05	26123	**2.59**	7827	6.70	14834
1,3,6	41.28	73445	96.47	99494	44.11	96801	56.32	103588	**39.77**	115166	80.01	128179
1,4,6	18.70	23250	7.41	9331	3.08	6054	2.74	5758	**1.27**	4234	4.04	9406
1,5,5	5.14	17179	3.26	4871	2.25	8820	3.20	15253	1.72	9939	**1.16**	5875
1,6,6	2.11	3960	1.39	1275	**0.88**	2551	1.12	3293	1.46	5806	0.97	3428
1,7,8	84.48	124226	159.16	273478	50.68	99574	**27.78**	85722	90.92	262880	106.09	250338
1,8,3	1.44	5872	5.37	8888	2.74	6083	2.53	5974	**0.47**	1599	1.02	3216
1,10,9	270.98	373982	1886.15	2389076	309.33	682210	**75.39**	158492	790.55	1802971	170.42	415286
2,1,5	0.37	1217	0.50	653	0.24	221	0.50	1405	**0.19**	710	0.22	624
2,3,6	240.14	162671	136.88	94966	195.79	181709	158.07	153738	**83.65**	159623	87.43	89192
2,5,4	95.90	160104	70.44	72447	36.50	74236	**21.28**	39374	87.26	186018	206.94	360892
2,6,5	99.20	130621	154.47	127314	116.23	163864	123.29	199502	214.24	380586	**64.26**	87175
2,8,5	58.67	136001	253.70	294527	63.53	118504	**38.83**	87444	116.11	221235	113.11	168101
2,9,3	13.61	37792	31.62	41817	**13.21**	28161	14.71	29910	32.67	74192	14.81	23530
2,10,8	590.73	507418	325.27	224429	**97.09**	133974	110.78	159988	162.03	224753	293.49	389813
Geom.	25.21	45445.09	35.46	40927.05	16.12	30816.80	**14.77**	32380.06	16.61	44284.41	17.79	36937.70

– A part-time employee (P) must work between 3 and 5.75 hours, plus a 15 minute break.
– A full-time employee (F) must work between 6 and 8 hours, plus 1 hour for lunch (L), and 15 minute breaks before and after.
– An employee can only be rostered while the business is open.

These constraints can be formulated as a `grammar` constraint as follows:

$$S \rightarrow RP^{[13,24]}R \mid RF^{[30,38]}R$$

$$\begin{array}{ll} F \rightarrow PLP & P \rightarrow WbW \\ W \rightarrow A_i^{[4,\ldots]} & A_i \rightarrow a_i A_i \mid a_i \\ L \rightarrow llll & R \rightarrow rR \mid r \end{array}$$

This `grammar` constraint can be converted into `s-DNNF` as described in [11]. Note that some of the productions for P, F and A_i are annotated with restricted intervals – while this is no longer strictly context-free, it can be integrated into the graph construction with no additional cost.

The coverage constraints and objective function are implemented using the monotone BDD decomposition described in [16].

Table 2 compares our propagation algorithms versus the domain consistent decomposition [12] on the shift scheduling examples of [11]. Instances $(2, 2, 10)$ and $(2, 4, 11)$ are omitted, as no solvers proved the optimum within the time limit. Generally any of the direct propagation approaches require less search than a decomposition based approach. This is slightly surprising since the decomposition has a richer language to learn nogoods on. But it accords with earlier results for BDD propagation, the Tseitin literals tend to confuse activity based search making it less effective. The non-incremental propagator `full` is too expensive, but once we have incremental propagation (`ip`) all methods beat the decomposition. Clearly incremental explanation is not so vital to the execution time as incremental propagation, which makes sense since we only explain on demand, so it is much less frequent than propagation. Both weakening and greedy explanations increase the search space, but only weakening pays off in terms of execution time.

5.2 Forklift Scheduling

As noted in [17], the shift scheduling problem can be more naturally (and efficiently) represented as a DFA. However, for other grammar constraints, the corresponding DFA can (unsurprisingly) be exponential in size relative to the arity.

In order to evaluate these methods on grammars which do not admit a tractable regular encoding, we present the *forklift scheduling problem*

A forklift scheduling problem is a tuple (N, I, C), where N is the number of stations, I is a set of items and C is a cost for each action. Each item $(i, source, dest) \in I$ must be moved from station *source* to station *dest*. These objects must be moved using a forklift. The possible actions are:

$move_j$ Move the forklift to station j.
$load_i$ Shift item i from the current station onto the forklift tray.
$unload_i$ Unload item i from the top of the forklift tray at the current station.
idle Do nothing.

Items may be loaded and unloaded at any number of intermediate stations, however they must be unloaded in a LIFO order.

The LIFO behaviour of the forklift can be modelled with the grammar:

$$\begin{aligned} S &\rightarrow W \mid WI \\ W &\rightarrow WW \\ &\quad\mid move_j \\ &\quad\mid load_i \; W \; unload_i \\ I &\rightarrow idle \; I \mid idle \end{aligned}$$

Note that this grammar does not prevent item i from being loaded multiple times, or enforce that the item must be moved from *source* to *dest*. To enforce these constraints, we define a DFA for item $(i, source, dest)$ with 3 states for each station:

$q_{k,O}$ Item at station k, forklift at another station.
$q_{k,U}$ Forklift and item both at station k, but not loaded.
$q_{k,L}$ Item on forklift, both at station k.

With start state $q_{source,O}$ and accept states $\{q_{dest,O}, q_{dest,U}\}$. We define the transition function as follows (where $\perp$ represents an error state):

δ	$move_k$	$move_j, j \neq k$	$load_i$	$load_j, j \neq i$	$unload_i$	$unload_j, j \neq i$
$q_{k,O}$	$q_{k,U}$	$q_{k,O}$	$\perp$	$q_{k,O}$	$\perp$	$q_{k,O}$
$q_{k,U}$	$q_{k,U}$	$q_{k,O}$	$q_{k,L}$	$q_{k,U}$	$\perp$	$q_{k,U}$
$q_{k,L}$	$q_{k,L}$	$q_{j,L}$	$\perp$	$q_{k,L}$	$q_{k,U}$	$q_{k,L}$

Table 3: Comparison of different methods on forklift scheduling problems.

Inst.	`decomp`		`full`		`ip`		`ip+w`		`ip+g`		`ip+gw`	
	time	fails	time	fails	time	fails	time	fails	time	fails	time	fails
3-4-14	**0.58**	4962	2.00	4966	1.52	5912	1.30	3820	1.00	6069	0.80	4392
3-5-16	10.98	42421	46.19	53789	35.40	45486	15.32	28641	22.72	42023	**9.19**	30219
3-6-18	318.55	492147	687.69	611773	380.09	458177	223.06	289221	275.31	454268	**124.10**	279207
4-5-17	36.60	83241	142.77	146131	77.52	99027	43.94	72511	60.75	112160	**20.42**	53643
4-6-18	358.47	587458	704.20	643074	379.09	437797	251.67	331946	410.26	719219	**124.39**	283560
4-7-20	—	—	—	—	—	—	3535.74	3640783	—	—	**1858.79**	3057492
5-6-20	1821.55	2514119	—	—	—	—	1922.73	1894107	2521.49	3374187	**1220.28**	1893025
Geom.	—	—	—	—	—	—	118.80	176102.11	—	—	**65.65**	164520.95

A `regular` constraint is used to encode the DFA for each item.

Experiments with forklift sequencing use randomly generated instances with cost 1 for $load_j$ and $unload_j$, and cost 3 for $move_j$. The instance n-i-v has n stations and i items, with a planning horizon of v. The instances are available at `ww2.cs.mu.oz.au/∼ggange/forklift`.

The results for forklift scheduling are shown in Table 3. They differ somewhat for those for shift scheduling. Here the `full` propagator has no search advantage over the decomposition and is always worse, presumably because the interaction with the DFA side constraints is more complex, which gives more scope for the decomposition to use its intermediate literals in learning. Incremental propagation `ip` is similar in performance to the decomposition. It requires substantially less search than `full` presumably because the order of propagation is more closely tied to the structure of `s-DNNF` circuit, and this creates more reusable nogoods. For forklift scheduling weakening both dramatically reduces search and time, and greedy explanation has a synergistic effect with weakening. The best version `ip+gw` is significantly better than the decomposition approach.

6 Conclusion

In this paper we have defined an s-DNNF propagator with explanation. We define non-incremental and incremental propagation algorithms for s-DNNF circuits, as well as minimal and greedy approaches to explaining the propagations. The incremental propagation algorithm is significantly better than non-incremental approach on our example problems. Greedy explanation usually improves on non-incremental explanation, and weakening explanations to make them shorter is usually worthwhile. The resulting system provides state-of-the-art solutions to problems encoded using grammar constraints.

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. 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. Volume 5202 of LNCS. (2008) 509–523
2. Gange, G., Stuckey, P., Lagoon, V.: Fast set bounds propagation using a BDD-SAT hybrid. Journal of Artificial Intelligence Research **38** (2010) 307–338
3. Gange, G., Stuckey, P.J., Szymanek, R.: MDD propagators with explanation. Constraints **16**(4) (2011) 407–429
4. Pesant, G.: A regular language membership constraint for finite sequences of variables. In Wallace, M., ed.: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming. Volume 3258 of LNCS., Springer-Verlag (2004) 482–495
5. Darwiche, A.: Sdd: A new canonical representation of propositional knowledge bases. In: IJCAI. (2011) 819–826
6. Tseitin, G.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic **Part 2** (1968) 115–125
7. Schulte, C., Stuckey, P.: Efficient constraint propagation engines. ACM Transactions on Programming Languages and Systems **31**(1) (2008) Article No. 2
8. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3) (2009) 357–391
9. Fargier, H., Marquis, P.: On valued negation normal form formulas. In: IJCAI. (2007) 360–365
10. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research **17** (2002) 229–264
11. Quimper, C., Walsh, T.: Global grammar constraints. In: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming. Volume 4204 of LNCS. (2006) 751–755
12. Jung, J.C., P., B., Katsirelos, G., Walsh, T.: Two encodings of DNNF theories. ECAI Workshop on Inference Methods Based on Graphical Structures of Knowledge (2008)
13. 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. Volume 3819 of LNCS. (2006) 103–117
14. Subbarayan, S.: Efficent reasoning for nogoods in constraint solvers with BDDs. In: Proceedings of Tenth International Symposium on Practical Aspects of Declarative Languages. Volume 4902 of LNCS. (2008) 53–57
15. Demassey, S., Pesant, G., Rousseau, L.M.: A cost-regular based hybrid column generation approach. Constraints **11**(4) (2006) 315–333
16. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: BDDs for pseudo-boolean constraints - revisited. In: SAT. (2011) 61–75
17. Katsirelos, G., Narodytska, N., Walsh, T.: Reformulating global grammar constraints. Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (2009) 132–147