Explaining Propagators for Edge-valued Decision Diagrams

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

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

Abstract. Propagators that combine reasoning about satisfiability and reasoning about the cost of a solution, such as weighted all-different, or global cardinality with costs, can be much more effective than reasoning separately about satisfiability and cost. The COST-MDD constraint is a generic propagator for reasoning about reachability in a multi-decision diagram with costs attached to edges (a generalization of COST-REGULAR). Previous work has demonstrated that adding nogood learning for MDD propagators substantially increases the size and complexity of problems that can be handled by state-of-the-art solvers. In this paper we show how to add explanation to the COST-MDD propagator. We demonstrate on scheduling benchmarks the advantages of a learning COST-MDD global propagator, over both decompositions of COST-MDD and MDD with a separate objective constraint using learning.

1 Introduction

Optimization constraints merge the checking of feasibility and optimization conditions into a single propagator. A propagator for an optimization constraint filters decisions for variables which cannot take part in a solution which is better than the best known solution. They also propagate the bounds on the cost variable to keep track of its lower bound, and hence allow fathoming of the search, when no better solution can be found. There is a significant body of work on optimization constraints including: *weighted alldifferent* [1] and *global cardinality with costs* [2]. In this paper we examine the COST-MDD optimization constraint which is a generalization of the COST-REGULAR [3] constraint.

Previous work has explored the use of *Boolean Decision Diagrams* (BDDs) [4, 5] and *Multi-valued Decision Diagrams* (MDDs) [6] for automatically constructing efficient global propagators. But these propagators do not handle costs. And adding a separate objective function constraint to encode the costs, leads to significantly weaker propagation.

COST-MDD is a generic constraint that can be used to encode many problems where the feasibility of a sequence of decisions is represented by an MDD, and the costs of the sequence of decisions is given by the sum of the weights on the

edges taken in this MDD. COST-REGULAR [3] is encoded as a particular form of COST-MDD where the set of states at each level is uniform, and the transition from one level to another is uniform. The WEIGHTED-GRAMMAR constraint [7] is a similar optimization constraint which permits a more concise encoding of some constraints than COST-MDD, but is less convenient to construct and manipulate.

In this paper we investigate how to incorporate COST-MDD global propagators into a lazy clause generation [8] based constraint solver. The principle challenge is to be able to explain propagations as concisely as possible, in order that the nogoods learnt are as reusable as possible. We give experimental evidence that explaining COST-MDD propagators outperform both decompositions of COST-MDD and previous MDD-based propagators.

2 Preliminaries

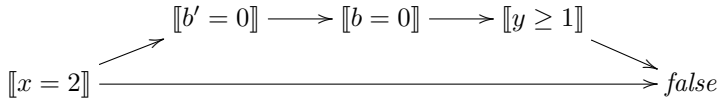
Constraint programming solves constraint satisfaction problems by interleaving propagation, which remove impossible values of variables from the domain, with search, which guesses values. All propagators are repeatedly executed until no change in domain is possible, then a new search decision is made. 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. [9].

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 $D(x) \subseteq D'(x), \forall x \in \mathcal{V}$. We let $\mathsf{lb}_D(x) = \min D(x)$ and $\mathsf{ub}_D(x) = \max D(x)$, and will omit the D subscript when D is clear from the context. 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 computes a *nogood* which records the reason for failure. The nogood prevents search making the same incorrect set of decisions later.

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 $\neg\,[\![x = d]\!]$, and $[\![x \geq d]\!]$ as shorthand for $\neg\,[\![x \leq d-1]\!]$. 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 $[\![b = 0]\!] \rightarrow [\![y \geq 1]\!]$ and the second constraint determines failure with explanation $[\![x = 2]\!] \wedge [\![y \geq 1]\!] \rightarrow \mathit{false}$. The graph of the implications is



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 Edge-valued Decision Diagrams

A *Multi-valued Decision Diagram (MDD)* encodes a propositional formula as a directed acyclic graph with a single terminal $\mathcal{T}$ representing *true* (the *false* terminal is typically omitted for MDDs). In an MDD G, each internal node $n = \mathit{node}(x_i, [(v_1, n_1), (v_2, n_2), \ldots, (v_k, n_k)])$ is labelled with a variable x_i, and outgoing edges consisting of a value v_j and destination node n_j. Each node represents the formula

$$\langle n \rangle \Leftrightarrow \bigvee_{j=1}^{k} (x = v_j \wedge \langle n_j \rangle)$$

where $\langle n \rangle$ is a Boolean representing the reachability of node n, and $\langle \mathcal{T} \rangle = \mathit{true}$. The MDD constraint enforces $\langle G.root \rangle = \mathit{true}$ where $G.root$ is the root of the MDD.

In this paper we restrict ourselves to *layered* MDDs. In a layered MDD G each node n is assigned to a layer k and all its child nodes must be at layer $k+1$. Each node at layer k is labelled with the *same* variable x_k, and the root node $G.root$ is at layer 1. This encodes an ordered MDD with no *long edges*, which typically propagate faster than MDDs with long edges [6]. Each assignment satisfying the constraint represented by G corresponds to a path from the root $G.root$ to the terminal $\mathcal{T}$. If, at the i-th layer, the path follows an edge with value v_j, the corresponding assignment has $x_i = v_j$.

An *Edge-valued MDD (EVMDD)* G is a (layered) MDD with a weight attached to each edge. Hence nodes are of the form

$$n = \mathit{node}(x_i, [(v_1, w_1, n_1), (v_2, w_2, n_2), \ldots, (v_k, w_k n_k)]),$$

where w_j is the weight of the j^{th} outgoing edge. The cost of a solution $\theta = [x_1 = d_1, x_2 = d_2, \ldots, x_n = d_n]$ which defines a path from the root of G to $\mathcal{T}$ is given by the sum of the weights along the corresponding path in the EVMDD. Each

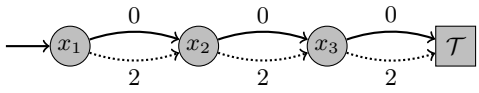


Fig. 1: A simple EVMDD with only paths of even cost.

node n enforces the constraint:

$$\langle\langle n \rangle\rangle = \begin{cases} 0 & n = \mathcal{T} \\ \min\{w_j + \langle\langle n_j \rangle\rangle \mid j = 1, .., k \wedge x_i = v_j\} & \text{otherwise} \end{cases}$$

where $\langle\langle n \rangle\rangle$ holds the cost of the minimal weight path from n to $\mathcal{T}$.

For convenience, we denote edges by 4-tuples $(n, x_i = v_j, w_j, n_j)$, representing the edge with source n (in layer i), destination n_j (in layer $i+1$) and weight w_j corresponding to the value v_j. We will refer to the components as $(e.begin, e.var = e.val, e.weight, e.end)$.

We use $s.out\_edges$ to refer to all the edges of the form $(s, \_, \_, \_)$, i.e. those leaving node s, and $d.in\_edges$ to refer to edges of the form $(\_, \_, \_, d)$, i.e. those entering node d. We use $G.edges(x_i, v_j)$ to record the set of edges of the form $(\_, x_i = v_j, \_, \_)$ in EVMDD G.

The COST-MDD constraint COST-MDD$(G, [x_1, \ldots, x_n], \bowtie, C)$ requires that $\langle\langle G.root \rangle\rangle \bowtie C$ where $\bowtie \in \{\leq, =, \geq\}$. Note that the constraint (except the $\geq$ incarnation) enforces satisfiability, i.e., that there is a path from $G.root$ to $\mathcal{T}$, since otherwise $\langle\langle G.root \rangle\rangle = \infty$. The COST-MDD constraint can represent COST-REGULAR as well as other constraints representable by automata with counters.

Our definition of EVMDDs differs from the standard treatment of edge-valued BDDs [10], apart from the extension from Boolean variables to finite-domain variables. We do not require the graph to be deterministic; a single node may have multiple edges annotated with the same value. Also, we do not require the edge weights to be normalized; normalization may reduce the size of the graph by inducing additional sharing, but does not affect propagation or explanation.

3 EVMDD Propagation

An incremental algorithm for propagating COST-REGULAR constraints was described in [3]. This algorithm essentially converts the COST-REGULAR constraint into a COST-MDD constraint where $\bowtie$ is $=$, then performs propagation on this transformed representation. This algorithm operates by incrementally maintaining the distance of the shortest $up[n]$ and longest $lup[n]$ path from the root to each node n, and the distance of the shortest $dn[n]$ and longest $ldn[n]$ path from each node n to $\mathcal{T}$. Given a constraint COST-MDD$(G, [x_1, \ldots, x_n], =, C)$, an edge e may be used to build a path from $G.root$ to $\mathcal{T}$ only if $up[e.start] + e.weight + dn[e.end] \leq \mathsf{ub}(C)$ and $lup[e.start] + e.weight + ldn[e.end] \geq \mathsf{lb}(C)$.

The description in [3] does not mention how changes to the bounds of C are handled. When the upper bound of C is reduced, the lengths of all shortest

paths remain the same; however, the domains of variables x_i may change, if the shortest path through $x_i = v_j$ is longer than the updated bound.

Example 2. Consider the EVMDD (EVBDD) G shown in Figure 1 where edges for value 0 are shown dotted, and edges for value 1 are shown full. The constraint COST-MDD$(G, [x_1, x_2, x_3], =, C)$ encodes the equation $2x_1 + 2x_2 + 2x_3 = C$. If we initially have $D(C) = [0..2]$, no values may be eliminated, as every edge can occur on a path of cost at most 2. However, if we reduce $\mathsf{ub}(C)$ to 1, we must eliminate $x_i = 1$ from the domain of each variable.

The authors claim that their propagation algorithm enforces domain consistency on the x variables in a COST-MDD constraint. This statement is not correct.

Example 3. Consider again the EVMDD G shown in Figure 1. The algorithm of [3] makes no propagation for the constraint COST-MDD$(G, [x_1, x_2, x_3], =, C)$ when $D(C) = \{3\}$. This is because every edge can take part in a path which is both longer (length 4) or shorter (length 2) than the bounds of C. But there is no support for any value of x_i since there is no path of length exactly 3. □

In fact even bounds propagation is NP-hard for COST-MDD where $\bowtie$ is $=$, using any applicable definition of bounds consistency [11].

Theorem 1. *Domain propagation, bounds($\mathcal{Z}$) or bounds($\mathcal{D}$) consistent propagation for* COST-MDD$(G, [x_1, \ldots, x_n], =, C)$ *is NP-hard*

Proof. We map SUBSETSUM to COST-MDD *propagation. Given a set* $S = \{s_1, \ldots, s_m\}$ *of numbers and target* T *we build an EVBDD with* m *0-1 variables* $x_1, \ldots, x_m$ *and* m *nodes* $n_1, \ldots, n_m$ $(n_{m+1} = \mathcal{T})$ *with* $2m$ *edges* $(n_i, x_i = 0, 0, n_{i+1})$ *and* $(n_i, x_i = 1, s_i, n_{i+1})$. *Enforcing domain (or equivalently in this case bounds($\mathcal{Z}$) or bounds($\mathcal{D}$)) consistency on* COST-MDD$(G, [x_1, \ldots, x_n], =, C)$ *with* $D(C) = \{T\}$ *generates a false domain unless the SUBSETSUM holds.* □

In this paper we restrict consideration to the COST-MDD constraint of the form COST-MDD$(G, [x_1, \ldots, x_n], \leq, C)$. This is the critical form of the constraint when we are trying to minimize costs. Treatment of COST-MDD$(G, [x_1, \ldots, x_n], \geq, C)$ is identical by negating each edge weight and the cost variable; the treatment of COST-MDD$(G, [x_1, \ldots, x_n], =, C)$ in [3] is effectively combining propagators for each of COST-MDD$(G, [x_1, \ldots, x_n], \leq, C)$ and COST-MDD$(G, [x_1, \ldots, x_n], \geq, C)$.

We give a non-incremental propagation algorithm for the constraint COST-MDD$(G, [x_1, \ldots, x_n], \leq, C)$ in Figure 2.[3] evmdd_prop first records the shortest path (given the current domain D) from each node n to $\mathcal{T}$ in $dn[n]$ using mark_paths. It returns the shortest path from $G.root$ to $\mathcal{T}$. It then visits using infer all the edges reachable from $G.root$ that appear on paths of length less than $\mathsf{ub}(C)$. Initially the negation of all edge labels are placed in *inferences*. When an edge that appears on a path of length less than or equal to $\mathsf{ub}(C)$ is

[3] This is not novel with respect to [3] but they don't formally define their algorithm.

discovered, the negation of its label is removed from *inferences*. The algorithm returns the a lower bound of C (which may not be new) and any new inferences on x_i variables.

Example 4. Consider the propagation that occurs with the EVMDD of Figure 1 with $C \leq 2$ when we set $x_1 \neq 1$ ($x_1 = 0$) and $x_2 \neq 0$ ($x_2 = 1$). mark_paths sets $dn[\mathcal{T}] = 0$, $dn[x_3] = 0$ (using the variable name for the node name), $dn[x_2] = 2$ and $dn[x_1] = 2$ and returns 2. infer initially starts with $\mathit{inferences} = \{[\![x_1 \neq 0]\!], [\![x_1 \neq 1]\!], [\![x_3 \neq 0]\!], [\![x_3 \neq 1]\!]\}$. It sets $up[x_1] = 0$ then removes $[\![x_1 \neq 0]\!]$ from *inferences* setting $up[x_2] = 0$. It then removes $[\![x_2 \neq 2]\!]$ from *inferences* setting $up[x_3] = 2$. It removes $[\![x_3 \neq 0]\!]$ from *inferences*, but then when examining the full edge from x_3 the distance test fails. Hence it returns $\{[\![x_3 \neq 1]\!]\}$. The final inferences are $\{[\![C \geq 2]\!], [\![x_3 \neq 1]\!]\}$.

Proposition 1. *evmdd_prop maintains domain consistency for* COST-MDD$(G, [x_1, \ldots, x_n], \leq, C)$.

Proof. After evmdd_prop finishes if $v_j \in D(x_i)$ then there is an edge $(s, x_i = v_j, w, d)$ in G where $up[s] + w + dn[d] \leq \mathsf{ub}(C)$. Hence there is a path of edges from $G.root$ to s of length $up[s]$ and a path of edges from d to $\mathcal{T}$ of length $dn[d]$. If we set each variable to the value given on this path and $C = \mathsf{ub}(C)$ we have constructed a solution supporting $x_i = v_j$. Similarly, given $l = \mathsf{lb}(C)$ then after evmdd_prop finishes there is a path from $G.root$ to $\mathcal{T}$ of length l. If we set each variable to the value given on this path, and C to any value $d \in D(C)$ domain we have constructed a solution supporting $C = d$. □

It is straightforward to make the above algorithm incremental in changes in x variables. A removed edge $e = (s, x = v, w, d)$ forces the recalculation of $dn[s]$ which may propagates upward, and $up[s]$ which may propagate downwards. If a change reaches $G.root$ or $\mathcal{T}$ then the lower bound on C may change. When the upper bound of C changes, we simply scan the edges for each value until we find one that is still feasible (infeasible edges are not checked on later calls).

4 Explaining EVMDD 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. In order to be incorporated in a nogood learning solver, the EVMDD propagator must be able to explain its inferences.

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 EVMDD

```
evmdd_prop(G, [x_1, ..., x_n], C, D)
    ĉ := mark_paths(G, D)
    L := infer(G, [x_1, ..., x_n], D, ub(C))
    return {[[C ≥ ĉ]]} ∪ L

mark_paths(G, D)
    for(n ∈ G.nodes) dn[n] := ∞
    dn[T] := 0; queue := {T}
    while(queue ≠ ∅)
        nqueue := {} % Record nodes of interest on the next level.
        for(node in queue)
            for(e in node.in_edges)
                if(e.val ∈ D(e.var))
                    dn[e.begin] := min(dn[s.begin], e.weight + dn[node])
                    nqueue ∪={e.begin}
        queue := nqueue
    return dn[G.root]

infer(G, [x_1, ..., x_n], D, u)
    inferences := {[[x_i ≠ v_j]] | 1 ≤ i ≤ n, v_j ∈ D(x_i)}
    for(n ∈ G.nodes) up[n] := ∞
    up[G.root] := 0; queue := {G.root}
    while(queue ≠ ∅)
        nqueue := {} % Record nodes of interest on the next level.
        for(node in queue)
            for(e ∈ node.out_edges)
                if(e.val ∈ D(e.var))
                    if(up[node] + e.weight + dn[e.end] ≤ u)
                        inferences := inferences − {[[e.var ≠ e.val]]}
                        up[e.end] := min(up[e.end], e.weight + up[node])
                        nqueue ∪={e.end}
        queue := nqueue
    return inferences
```

Fig. 2: Algorithm for inferring newly propagated literals.

unsatisfiable. A correct explanation is (the negation of) all the values for other variables which are currently false. 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) $\bigwedge_{l \in expln} l \rightarrow [\![x \neq v]\!]$ since removing any literal l' from the *expln* would mean $\text{COST-MDD}(G, [x_1, \ldots, x_n], \leq, C) \wedge \bigwedge_{l \in expln - \{l'\}} l \wedge [\![x = v]\!]$ is satisfiable. Constructing a smallest minimal explanation for an EVMDD is NP-hard just as for BDDs [12].

We adapt the minimal MDD explanation algorithm used in [6] to COST-MDD constraints. The propagator conflicts when the shortest path from $G.root$ to $\mathcal{T}$ (under the current domain) is longer than $\mathsf{ub}(C)$. To construct a minimal

```
evmdd_explain(G, C, D, [[x ≠ v]])
    D' := D with D(x) replaced by D'(x) = {v}
    ĉ := mark_paths(G, D')
    if(ĉ < ∞ or choice) u := ub(C) + 1
    else u := ∞
    return [[x ≠ v]] ← collect_expln(G, C, x, v, u)

evmdd_explain_lb(G, C, D, [[C ≥ l]])
    mark_paths(G, D) % unnecessary if just run evmdd_prop
    return [[C ≥ l]] ← collect_expln(G, C, ⊥, ⊥, l) − {[[C ≤ l − 1]]}

collect_expln(G, C, x, v, u)
    queue := {G.root}; up[G.root] := 0; s := ∞
    while(queue ≠ ∅)
        for(node in queue)
            for(e ∈ node.out_edges)
                up[e.end] := ∞
                if(e.var ≠ x and up[node] + e.weight + dn[e.end] < u)
                    explanation ∪= [[e.var ≠ e.val]]
                else s := min(s, up[node] + e.weight + dn[e.end])
        nqueue := {} % Record nodes of interest on the next level.
        for(node in queue)
            for(e ∈ node.out_edges)
                if ((e.var = x and e.val = v)
                        or (e.var ≠ x and [[e.var ≠ e.val]] ∉ explanation))
                    nqueue ∪={e.end}
                    up[e.end] := min(up[e.end], up[node] + e.weight)
        queue := nqueue
    return explanation ∪ [[C ≤ s − 1]]
```

Fig. 3: Algorithms for computing a minimal explanation.

explanation, we begin with the set of values that have been removed from variable domains, and progressively restore any values which would not re-introduce a path of length $\leq \mathsf{ub}(C)$.

The minimal explanation algorithm is illustrated in Figure 3. To explain $[\![x \neq v]\!]$ under current domain D, we first create the domain D' where $D'(x) = \{v\}$ and otherwise D' agrees with D. With this domain the constraint is unsatisfiable. We use mark_paths to compute the shortest path from each node n to $\mathcal{T}$ and store this in $dn[n]$. It returns the shortest path $\hat{c}$ from root to $\mathcal{T}$. If $\hat{c}$ is finite, or we choose to (by setting global *choice* true) we use an upper bound of C in the explanation, by setting $u \neq \infty$. collect_expln traverses the EVMDD from the root, building an explanation of literals which if not true would cause a path of length $< u$ to be created in the EVMDD. The algorithm examines all reachable nodes on a level (initially just the root) and if adding an edge would create a path shorter than u then the (negation of) the label on the edge is added to the explanation, if not then we update s which records the shortest path found from

root to $\mathcal{T}$ with length $\geq u$. The algorithm then adds all the nodes of the next level which are still reachable, and updates the shortest path from the root to each such node n storing this in $up[n]$. This continues while there are still some reachable nodes. At the end the algorithm returns the collected explanation, plus the relaxed upper bound literal $[\![C \leq s-1]\!]$, which ensures that none of the paths found from root to $\mathcal{T}$ can be traversed.

Since the procedures mark_paths and collect_expln perform one and two breadth-first traversals of the graph, respectively, the explanation requires $O(|G|)$ time.

Proposition 2. *evmdd_explain*$(G, C, D, [\![x \neq v]\!]$ *returns a correct minimal explanation for* $[\![x \neq v]\!]$.

Proof. (Sketch) The algorithm implicitly maintains the invariant that there is no path in G through an edge labelled $x = v$ of length less than or equal to $\mathsf{lb}(C)$ *which does not make use of an edge in DE. Initially DE is the set of edges e where $e.var \neq x$ and $e.val \notin D(e.var)$. The base case holds using the correctness of* evmdd_prop. *During* collect_expl *we remove processed edges from this implicit set DE, except those kept in explanation. Whenever we remove an edge from DE the shortest path through the edge that uses an edge labeled $x = v$ and none of the edges in DE is $>$* $\mathsf{ub}(C)$. *This demonstrates the correctness of the algorithm, since the explanation literals force that $[\![x \neq v]\!]$ holds since there is no feasible path through any edges labelled $x = v$.*

For minimality we can reason that if we remove any literal from the explanation, then we would have added a path that was too short passing through an edge labelled $x = v$. The minimality of the bound constraint $[\![C \leq s-1]\!]$ follows since if we relax it we will allow a path of length s through $x = v$. □

Explaining a new lower bound l for C is similarly defined by evmdd_explain_lb. We compute $dn[n]$ for each reachable node using mark_paths with the current domain D, then choose a set of literals to ensure no shorter paths are allowed. In this case collect_expln will always return $[\![C \leq l-1]\!]$ in the explanation which we can safely omit. Explaining failure of the whole constraint is identical to explaining why $C \geq \infty$.

Example 5. Consider the constraint defined by the EVMDD shown in Figure 4, which encodes a simple scheduling constraint requiring shifts to be of even length. Assume the solver first propagates $C \leq 2$, then fixes $[\![x_1 \neq 1]\!]$ and $[\![x_2 \neq 0]\!]$. The only satisfying assignment is then $[x_1, x_2, x_3, x_4] = [0, 1, 1, 0]$.

If we are asked to explain the inference $x_4 \neq 1$, we first compute the shortest paths from each node to $\mathcal{T}$ through $x_4 = 1$, using mark_paths. This is shown in Figure 4(b). Notice that the cost at the root node is ∞. This indicates that, even without a cost bound, there is no feasible path through $x_4 = 1$. We have the choice of either omitting the cost bound (obtaining an explanation not dependent on C) or including it and possibly obtaining a smaller explanation.

Whether or not we include a bound on C, we proceed by sweeping down level-by-level from the r_1. Assuming we include the bound $[\![C \leq 2]\!]$, so $u = 3$, we first check if any of the outgoing edges would introduce a path of length

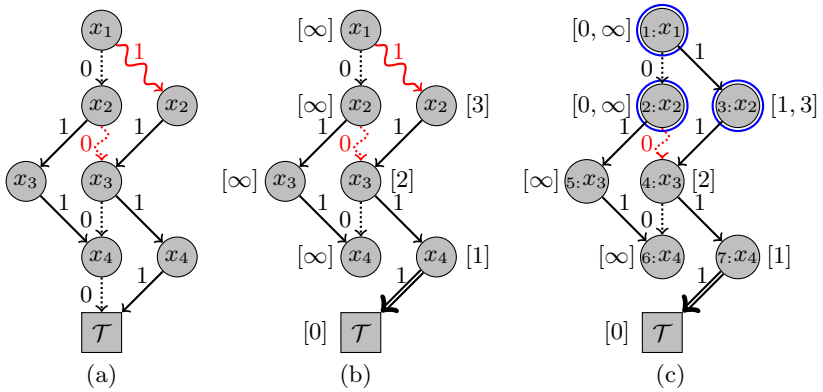


Fig. 4: (a) An EVMDD which requires shifts to be assigned in blocks of two. (b) We compute the shortest path from each node to $\mathcal{T}$. (c) Enqueued nodes are shown circled in blue, and have been annotated with the shortest path from n_1 under the current assignment.

less than 3. We find that the edge from n_1 to n_3 can safely be restored, since $up[n_1] + 1 + dn[n_3] = 4 \geq u = 3$. We update $s = 4$. As no edges introduce a feasible path, we update up for both n_2 and n_3, and add them to the queue for the next level.

At the second level, we discover that restoring the edge from n_2 to n_4 would introduce a feasible path, as $up[n_2]+0+dn[n_4] = 2 < u = 3$. The literal $[\![x_2 \neq 0]\!]$ must then be added to the explanation. Since n_4 is still reachable via n_3, both n_4 and n_5 are added to the queue for the next level; however, $up[n_4]$ is only updated by the edge from n_3, and not from n_2. This process continues until no further nodes remain. At the end $s = 4$ so we didn't need the bound on paths to be $\mathsf{ub}(C)$ it could have been looser. Hence we add $[\![C \leq 3]\!]$ to the explanation. The explanation returned is $[\![x_4 \neq 1]\!] \leftarrow [\![C \leq 3]\!] \wedge [\![x_2 \neq 0]\!]$. This is a minimal explanation.

If we omit the cost bound, then we cannot restore the edge from n_1 to n_3; so we construct the alternate explanation $[\![x_4 \neq 1]\!] \leftarrow [\![x_1 \neq 1]\!] \wedge [\![x_2 \neq 0]\!]$ which is also minimal. Note we omit the redundant literal $[\![C \leq \infty - 1]\!]$ created by collect_expl. □

4.2 Incremental Explanation

Example 6. Unfortunately, on large EVMDDs, constructing a minimal explanation can be expensive since explaining each inference may involve exploring the entire EVMDD. For these cases, we present a greedy algorithm for constructing valid, but not necessarily minimal, explanations in an incremental manner, often only examining a small part of the EVMDD.

We adapt the incremental MDD algorithm of [6] to COST-MDD. As in the MDD case, we explain $[\![x \neq v]\!]$ beginning from the set of edges corresponding to

```
mdd_inc_explain(G, x, v, u)
    for(n ∈ G.nodes) upe[n] := ∞
    for(n ∈ G.nodes) dne[n] := ∞
    kfa := {} % edges killed from above
    kfb := {} % edges killed from below
    for(e in G.edges(x, v))
        % Split possible supports
        Assign p_up, p_dn subject to:
            p_up + e.weight + p_dn ≥ u ∧ p_up ≤ up[e.begin] ∧ p_dn ≤ dn[e.end]
        if(p_up > up_0[e.begin])
            kfa ∪={edge}
            upe[e.begin] := max(upe[e.begin], p_up)
        if(p_dn > dn_0[e.end])
            kfb ∪={edge}
            dne[e.end] := max(dne[e.end], p_dn)
    % Explain all those killed from below
    return explain_down(kfb)
    % And all those killed from above
            ∪ explain_up(kfa)
```

Fig. 5: Top-level wrapper for incremental explanation.

$x = v$. For all such edges $e = (s, x = v, w, d)$, we know that $up[s] + w + dn[d] > \mathsf{ub}(C)$. If we have $up[s] + w + dn[d] = \mathsf{ub}(C) + 1$, then there is no flexibility in the bounds; we must select an explanation which ensures the shortest path from $G.root$ to s has cost $up[s]$, and the shortest path from e to $\mathcal{T}$ has cost $dn[d]$. We record the amount of cost that needs to be explained on all paths to s; this is denoted by $upe[s]$. We then sweep upwards, level-by-level, collecting an explanation which guarantees this minimum cost. At each level, we maintain the set of edges which need to be explained. If for some edge we have $up[s] + w < upe[d]$, then $[\![x \neq v]\!]$ must be added to the current explanation; otherwise, a feasible path would be introduced. We perform an initial pass over the edges at the current level to determine which values must be included in the explanation; during the second pass, we update upe for the source node of each edge that hasn't been excluded, and enqueue the set of incoming edges to be processed at the next level. If at any point we have $upe[s]$ is no greater than $up_0[s]$ (the shortest path to s under the initial variable domains), then we don't need to enqueue the incoming edges, as an empty explanation is sufficient.

If $up[s] + w + dn[d] > \mathsf{ub}(C) + 1$, then we can potentially relax the generated explanation. Obviously, the amount by which we relax $up[s]$ affects the amount of slack available to $dn[d]$. To relax the bounds as far as possible, we would initially allocate as much slack as possible to $up[s]$, and collect the corresponding explanation. Before performing the downward pass, we would then propagate the newly reduced path lengths back to the current layer, to determine how much slack remains for the explanation of d.

```
explain_down(kfb)
    reason = {}
    % Traverse the MDD downwards, breadth first
    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(e.val ∉ D(e.var) and
                e.weight + dn[e.end] < dne[e.begin])
                % 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 and e.weight + dn0[e.end] < dne[e.begin])
                % If e is not explained already collect its outgoing edges
                next ∪= e.end.out_edges
                dne[e.end] := max(dne[e.end], dne[e.begin] − e.weight
        % Continue with the next layer of edges.
        kfb = next
    return reason
```

Fig. 6: Pseudo-code for incremental explanation of EVMDDs. explain_up acts in exactly the same fashion as *explain_down*, but in the opposite direction.

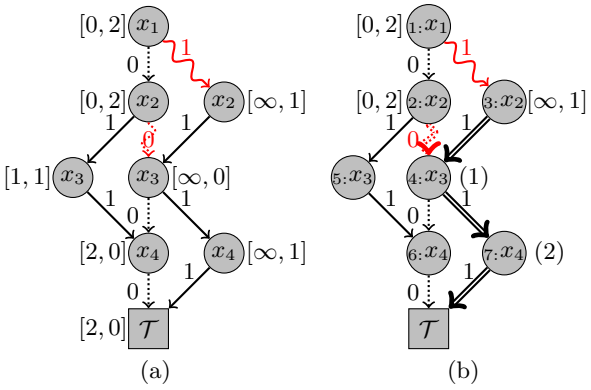


Fig. 7: The EVMDD from Example 5. (a) Values of $[up, dn]$ for each node. (b) Edges enqueued while explaining $[\![x_4 \neq 1]\!]$.

Instead, we determine *a priori* how the slack is allocated in the explanation. If either $up[s]$ or $dn[d]$ is ∞, then we build the explanation in only that direction (if both, we arbitrarily explain upwards). Otherwise, we explain as much as possible in the upward pass, and allocate all possible slack to the downwards pass. Alternative strategies for relaxing the bounds is interesting future work.

Consider again the case described in Example 5. During incremental propagation, we maintain up and dn for each node. These are shown in Figure 7(a). To explain $[\![x_4 \neq 1]\!]$, we need to eliminate some set of values which ensures that $up[n_7] + 1 + dn[\mathcal{T}] \geq 3$.

Under the current assignment, $up[n_7] = \infty$. However, as our current cost bound is 2, we only need to ensure $up[n_7] + 1 \geq 3$. We set $upe[n_7] = 2$, the amount of cost that must be guaranteed from above, and add n_7 to the queue. Expanding n_7, we find that it has only one parent, which is the edge from n_4 with weight 1. This edge cannot be eliminated, so we set $upe[n_4] = upe[n_7] - 1 = 1$, and enqueue n_4.

n_4 has 2 incoming edges, so we first check both edges to determine if any values must be added to the explanation. Examining the edge from n_2 to n_4, we have $upe[n_4] - 0 = 1 > up[n_2]$. This indicates that, if the edge from n_2 to n_4 were restored, a path of length 2 would be introduced. $[\![x_2 \neq 0]\!]$ is therefore added to the explanation. The edge from n_3 to n_4 is safe, as $up[n_3] = \infty \geq upe[n_3] - 1$.

We then make a second pass through the edges, to determine which new nodes must be enqueued. As $[\![x_2 \neq 0]\!]$ is in the explanation, we don't need to expand the node from n_2 to n_4. The edge from n_3 to n_4 is traversable, so we update $upe[n_3]$. However, since $upe[n_3] = 0$, and the base cost to reach n_3 is 1 (that is, $upe[n_3] \leq up_0[n_3]$), we don't need to enqueue n_3, since the cost to n_3 will always be at least 0. Since we have no nodes enqueued, the upwards pass is finished. Since there are no nodes which must be propagated downwards, this yields the final explanation $[\![x_4 \neq 1]\!] \leftarrow [\![C \leq 2]\!] \wedge [\![x_2 \neq 0]\!]$.

Observe that this is not minimal, since the explanation is still valid if we replace $[\![C \leq 2]\!]$ with $[\![C \leq 3]\!]$. □

5 Experimental Results

Experiments were conducted on a 3.00GHz Core2 Duo with 4 Gb of RAM running Ubuntu GNU/Linux 10.04. The propagators were implemented in `chuffed`, a state-of-the-art lazy-clause generation [8] based constraint solver. All experiments were run with a 10 minute time limit. For the minimal explanation algorithm, we always selected to use upper bounds in the explanation if possible.

We evaluate the COST-MDD constraints on a standard set of shift scheduling benchmarks. For the experiments, `dec` denotes propagation using a decomposition of COST-MDD like that of [13] but introducing a cost variable per layer of the EVMDD and summing them to compute cost, and $\texttt{dec}_{mdd}$ uses the domain-consistent Boolean decomposition described in [14] (or equivalently in [6]) and a separate cost constraint. `mdd` denotes using a separate MDD propagator [6] and cost constraint, `ev-mdd` denotes COST-MDD using incremental propagation and

minimal explanations, `ev-mdd`$_I$ denotes COST-MDD using incremental propagation and greedy explanations. We also tried a domain consistent decomposition of COST-MDD based on [7] but it failed to solve any of the shown instances, and is omitted.

5.1 Shift Scheduling

Shift scheduling, a problem introduced in [3], 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).
- 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:

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

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

We convert the `grammar` constraint into a Boolean formula, as described in [13]; however, we convert the formula directly into an MDD, rather than a `s-DNNF` circuit; the MDD and cost-MDD propagators, as well as the decompositions, are all constructed from this MDD. This process is similar to the reformulation described in [15]. 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].

The model using MDD is substantially better than the COST-MDD decomposition, and also superior to the MDD decomposition. It already improves upon the best published CP/SAT models for these problems[4] in [15]. The results for `ev-mdd` show that modelling the problem using COST-MDD is substantially better than separately modelling cost and an MDD constraint. Incremental greedy explanation can improve on minimal explanations, but the results demonstrate

[4] The best results for these problems use dynamic programming as a column generator in a branch-and-price solution [17].

Table 1: Comparison of different methods on shift scheduling problems.

Inst.	dec		$\texttt{dec}_{mdd}$		mdd		ev-mdd		$\texttt{ev-mdd}_I$	
	time	fails	time	fails	time	fails	time	fails	time	fails
1,2,4	—	—	14.51	39700	3.49	21888	0.20	607	**0.17**	635
1,3,6	—	—	11.25	40675	19.00	76348	**0.87**	4045	0.91	4156
1,4,6	36.48	86762	2.62	7582	0.69	3518	**0.11**	350	0.27	1077
1,5,5	5.64	32817	0.41	1585	0.52	3955	0.07	239	**0.06**	238
1,6,6	7.32	35064	0.40	1412	0.21	1161	**0.08**	249	0.11	413
1,7,8	27.58	77757	4.03	13149	2.43	12046	**0.73**	3838	0.83	4279
1,8,3	67.74	126779	0.85	5002	0.39	3606	**0.06**	219	0.07	262
1,10,9	321.44	441884	17.55	44222	19.77	68688	**1.23**	5046	1.31	7419
2,1,5	1.29	12520	0.14	691	0.24	1490	0.02	78	**0.01**	45
2,2,10	—	—	—	—	131.29	286747	**43.62**	99583	49.05	100958
2,3,6	—	—	188.77	187760	144.99	289568	**2.39**	6443	5.94	13695
2,4,11	—	—	—	—	391.59	918438	**42.38**	111567	92.89	220568
2,5,4	—	—	25.85	59635	12.18	50340	0.65	1545	**0.48**	1541
2,6,5	—	—	83.78	104911	30.27	80046	**6.18**	12100	7.63	16074
2,8,5	—	—	90.28	153331	34.69	110917	**4.99**	15507	10.02	26565
2,9,3	—	—	6.10	20472	9.17	42105	0.86	1898	**0.47**	1593
2,10,8	—	—	349.88	303227	95.61	168720	**8.85**	26331	17.22	37356
Total	—	—	—	—	896.53	2139581	**113.29**	289645	187.44	436874
Mean	—	—	—	—	52.74	125857.71	**6.66**	17037.94	11.03	25698.47
Geom.	—	—	—	—	7.92	31465.82	**0.86**	2667.65	1.03	3428.35

that minimal explanations are preferable. This contrasts with results for explaining MDD [6] where greedy incremental explanations were almost always superior. This may be because the presence of path costs in EVMDDs means that decisions higher in the graph have a greater impact on explanations further down (whereas for MDDs, the explanation only changes if a node is rendered completely unreachable).

6 Conclusion

In this paper we have defined how to explain the propagation of an EVMDD. Interestingly we have a trade-off between using cost bounds or literals on x to explain the same propagation. We define non-incremental minimal and incremental non-minimal explanation algorithms for EVMDDs. Using EVMDD with explanation to define a COST-MDD constraint, we are able to substantially improve on other modelling approaches for solving problems with COST-MDD with explanation.

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. Focacci, F., Lodi, A., Milano, M.: Cost-based domain filtering. In: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming. Volume 1713 of Lecture Notes in Computer Science., Springer (1999) 189–203
2. Régin, J.C.: Arc consistency for global cardinality constraints with costs. In: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming. Volume 1713 of Lecture Notes in Computer Science., Springer (1999) 390–404
3. Demassey, S., Pesant, G., Rousseau, L.M.: A cost-regular based hybrid column generation approach. Constraints **11**(4) (2006) 315–333
4. 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
5. Gange, G., Stuckey, P., Lagoon, V.: Fast set bounds propagation using a BDD-SAT hybrid. Journal of Artificial Intelligence Research **38** (2010) 307–338
6. Gange, G., Stuckey, P.J., Szymanek, R.: MDD propagators with explanation. Constraints **16**(4) (2011) 407–429
7. Katsirelos, G., Narodytska, N., Walsh, T.: The weighted grammar constraint. Annals OR **184**(1) (2011) 179–207
8. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3) (2009) 357–391
9. Schulte, C., Stuckey, P.: Efficient constraint propagation engines. ACM Transactions on Programming Languages and Systems **31**(1) (2008) Article No. 2
10. Vrudhula, S.B., Pedram, M., Lai, Y.T.: Edge valued binary decision diagrams. In: Representations of Discrete Functions. Springer (1996) 109–132
11. Choi, C., Harvey, W., Lee, J., Stuckey, P.: Finite domain bounds consistency revisited. In: Proceedings of the Australian Conference on Artificial Intelligence 2006. Volume 4304 of LNCS., Springer-Verlag (2006) 49–58
12. 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
13. 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
14. 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)
15. 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
16. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: BDDs for pseudo-boolean constraints - revisited. In: SAT. (2011) 61–75
17. Côté, M.C., Gendron, B., Rousseau, L.M.: Grammar-based integer programming models for multiactivity shift scheduling. Management Science **57**(1) (2011) 151–163