Sequential Precede Chain for value symmetry elimination

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

[1] Faculty of Information Technology, Monash University, Melbourne, Australia
[2] Data61, CSIRO, Melbourne, Australia

Abstract. The main global constraint used for removing value symmetries is the VALUE-PRECEDE-CHAIN constraint which forces the first occurences of values in an ordered list to be appear in order. We introduce the SEQ-PRECEDE-CHAIN constraint for the restricted, but common, case where the values are $1, 2, \ldots, k$, and variables in the constraint do not take values higher than k. We construct an efficient domain consistent propagator for this constraint, and show how we can generate explanations for its propagation. This leads us to an efficient domain consistent decomposition. We show how we can map any VALUE-PRECEDE-CHAIN to use instead SEQ-PRECEDE-CHAIN. Experiments show that the new propagator and decomposition are better than existing approachs to propagating VALUE-PRECEDE-CHAIN.

1 Introduction

The VALUE-PRECEDE-CHAIN constraint introduced by Law and Lee [1] is one of the principal symmetry breaking constraints for removing value symmetries. The constraint VALUE-PRECEDE-CHAIN$([t_1, t_2, \ldots, t_k], X)$ holds if in the sequence of values taken by variables X, at least one occurence of t_i occurs earlier in the sequence than any occurrence of t_j for all $1 \leq i < j \leq k$.

Law and Lee also introduce the constraint VALUE-PRECEDE(t_1, t_2, X) which requires that an occurence of t_1 occurs before any occurence of t_2 in X. They describe a domain consistent propagator for VALUE-PRECEDE and implement VALUE-PRECEDE-CHAIN by decomposition into a series of VALUE-PRECEDE:

$$\text{VALUE-PRECEDE-CHAIN}([t_1, t_2, \ldots, t_k], X) = \bigwedge_{i=2}^{k} \text{VALUE-PRECEDE}(t_{i-1}, t_i, X)$$

Law and Lee [1] observe that adding redundant VALUE-PRECEDE constraints obtains stronger propagation. However, even with redundant constraints, this encoding is not domain consistent.

Example 1. Consider the constraint VALUE-PRECEDE-CHAIN$([1, \ldots, 5], X)$, where $X = [x_1, \ldots, x_9]$), with domains $[\{0, 1\}, \{0, 1, 5\}, \{0, 3\}, \{0, 1, 2, 4\}, \{0, 1, 3\}, \{1, 3\}, \{2, 3, 4, 5\}, \{4, 5\}, \{0, 1, 2, 3\}]$. The domains of the variables are illustrated in Figure 1. We can clearly reduce the upper bound of x_2 and x_3 to 1 since there is

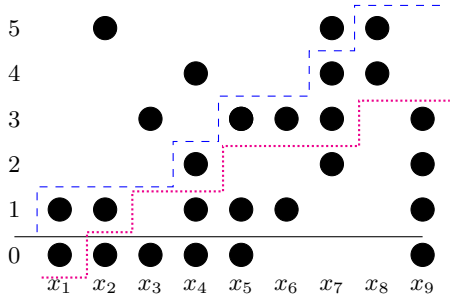


Fig. 1: A sequence of variables $[x_1, \ldots, x_9]$, with corresponding upper and lower frontiers.

no earlier occurence of 2. Similarly we can reduce the upper bound of x_4 to 2 since there is no earlier occurence of 2. Similarly we can reduce the upper bound of x_7 to 4 since there is no earlier occurence of 4. A decomposition into VALUE-PRECEDE constraints cannot obtain any further propagation: there are at least two possible occurrences each of 2 and 3 to the left of x_8, so neither can be assigned. And because all other variables may be lower, nothing else can propagate.

But we can also increase the lower bound of x_4 to 2 (fixing it to 2) since there is no other value 2 capable of supporting $x_8 = 4$. We know that at least one of x_5, x_6 and x_7 must take the value 3 to support this value, and hence x_4 cannot possibly take the value 1. □

It is observed in [2] that a domain consistent encoding may be obtained by reformulating VALUE-PRECEDE-CHAIN as a REGULAR [3] constraint, for which appropriate propagators [4–6] and decompositions [7, 8] exist. However, this reformulation is rather inefficient: the automaton, shown in Figure 2 has $k \times |\mathcal{D}(X)|$ edges, which are unfolded $|X|$ times in constructing the REGULAR constraint.

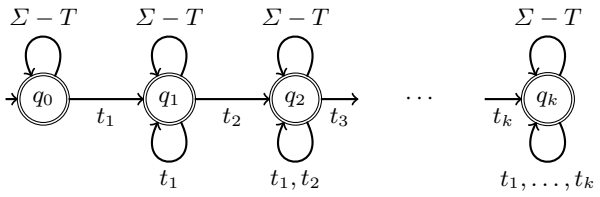


Fig. 2: Regular automata for VALUE-PRECEDE-CHAIN

In this paper we investigate a special case of VALUE-PRECEDE-CHAIN and show how it can be propagated efficiently, or implemented by a decomposition.

We then show how it can be used to encode any VALUE-PRECEDE-CHAIN constraint. The contributions of this paper are:

- an efficient domain consistent propagator for SEQ-PRECEDE-CHAIN which explains its propagations;
- an efficient decomposition of SEQ-PRECEDE-CHAIN;
- a proof that the decomposiion maintains domain consistency;
- an encoding of VALUE-PRECEDE-CHAIN using SEQ-PRECEDE-CHAIN which is domain consistent; and
- experiments showing that the propagator and decomposition are both highly competitive with existing approaches to VALUE-PRECEDE-CHAIN.

2 The common case: value-precede-chain($[1, 2, \ldots, k], X$)

The most frequent use of VALUE-PRECEDE-CHAIN is in cases where either all values in the domain are indistinguishable (i.e. colouring problems), or have exactly one distinguished value (i.e. concert hall scheduling, optional bin packing). We introduce a new global SEQ-PRECEDE-CHAIN(X) equivalent to VALUE-PRECEDE-CHAIN($[1, 2, \ldots, k], X$) where k is the maximum value appearing in the domain on any $x \in X$.

Given this new simplified form of the constraint we can build a global domain consistent propagator that is more efficient. Let $\mathcal{D}$ be the current domain of the variables, thought of as a set of unary constraints. Let $\mathcal{D}_x$ be the restriction of $\mathcal{D}$ to constraints only mentioning variables x. We use notation $\mathcal{D}(x)$ to refer to the current domain of x, that is the set of all values it can take $\mathcal{D}(x) = \{v \mid x = v \rightarrow \mathcal{D}_x\}$, and lb and ub to refer to the least and greatest value x can take in the current domain, $lb(x) = \min \mathcal{D}(x)$ and $ub(x) = \max \mathcal{D}(x)$.

2.1 Propagation

Propagation proceeds in two passes: a forward pass to find the upper frontier up, tightening upper bounds as it proceeds. Then it walks backwards to collect the lower frontier low, setting the value of any variable where the frontiers coincide.

The algorithm shown in Figure 3 is fairly straightforward. In a forward pass the algorithm maintains up the largest value seen in a chain of the form $1, \ldots, up$. If the upper bound of a variable is more than one greater than up we reduce it to $up + 1$. If the new upper bound is $up + 1$ we increment up and record in $first[up]$ the first position where the new bound is met. We similarly keep track of the highest lower bound seen so far low, and store in $last[low]$ the last position where the increasing sequence can take a value below low. Note that the sequence $[ub(x_1), \ldots, ub(x_n)]$ (after propagation) records the lexicographically greatest solution to the constraint.

The backwards pass pulls up low bounds. If low is first possible at position i then i must take that value, so we propagate that. If low is in the domain of $x[i]$ and this is before $last[low]$ we then reduce the lower bound. We also store

```
seq-precede-chain(x)
  up := 0
  low := 0
  last[0] := −∞ ▷ last[i] = ∞ for i > 0
  for(i ∈ 1..n) ▷ impose upper bounds
    if(ub(x[i]) > up + 1)
      propagate x[i] ≤ up + 1
    if(ub(x[i]) = up + 1)
      up := up + 1
      first[up] := i
    if(low < lb(x_i))
      last[lb(x_i)] := i
      low := lb(x_i)
  for(i ∈ n..1) ▷ impose lower bounds
    least[i] := lb(x_i)
    if(first[low] = i)
      propagate x[i] ≥ low
    if(i ≤ last[low] ∧ low ∈ dom(x[i]))
      least[i] := low
      last[low] := i
      low := low − 1
```

Fig. 3: Propagation algorithm for SEQ-PRECEDE-CHAIN.

the new *last* position information. The code to create $least[i]$ generates the lexicographically least solution to the constraint, $[least[1], \ldots, least[n]]$. This is not required for propagation, but useful for proofs.

Example 2. Consider the the problem of Example 1 using instead the constraint SEQ-PRECEDE-CHAIN(X). The propagator sets $first[1] = 1$ propagates $x_2 \leq 1$, $x_3 \leq 1$ and $x_4 \leq 2$, sets $first[2] = 4$, $first[3] = 5$ and propagates $x_7 \leq 4$ and sets $first[4] = 7$ and $last[1] = 7$, then sets $first[5] = 8$, and $last[4] = 8$. The blue line in Figure 1 shows the progress of up (drawn above its value). In the backwards pass low takes values $4, 4, 3, 3, 3$ setting $last[3] = 7$. Then we detect that $first[2] = 4$ and propagate $x_4 \geq 2$, and set $last[2] = 4$, then low takes values 2, 2, 1, 0, setting $last[1] = 2$ and $last[0] = 1$. The red line in Figure 1 shows the progress of low (drawn underneath its value). The sequence $least$ is calculated as [0,1,0,2,0,1,3,4,0]. □

Theorem 1. *The propagator for* SEQ-PRECEDE-CHAIN(X) *is correct.*

Proof. Given a solution $\theta = [v_1, \ldots, v_n]$ of SEQ-PRECEDE-CHAIN(X) we show that the propagator never removes it. Suppose the upper bound pruning removes earliest value v_i, then up at iteration $i - 1$ has value less than $v_i - 1$, but then there can be not $j < i$ with value $v_j = v_i - 1$ earlier in θ, otherwise up would have reached this level. Contradiction.

Suppose the lower bound pruning removes latest value v_i, then $v_i < u$ where $first[u] = i$, and low at iteration $i+1$ has value $u+1$. Since $first[u] = i$ then clearly $v_j < u, 1 \leq j < i$. Let k be the first position after i where $v_k = low$ in the k^{th} iteration. Then the sequence of low in between only steps down to $u+1$ hence there is no position in θ taking value u before a value of $u+1$. Contradiction. □

Theorem 2. *The propagator for* SEQ-PRECEDE-CHAIN(X) *enforces domain consistency.*

Proof. Given a value $v_i \in dom(x_i)$ after the propagation of SEQ-PRECEDE-CHAIN(X) we show that there exists a solution $[v_1, \ldots, v_n]$ where $v_j \in \mathcal{D}(x_j), 1 \leq j \leq n$ supporting this value. If $v_i = ub(x_i)$ then we can take the sequence of $[ub(x_1), \ldots, ub(x_n)]$. Suppose $first[ub(x_i)] \neq i$ then an alternate solution is clearly $[ub(x_1), \ldots, ub(x_{i-1}), v_i, ub(x_{i+1}), \ldots, ub(x_n)]$.

It remains to consider where $first[ub(x_i)] = i$ and $v_i < ub(x_i)$. We show that the sequence $\theta = [ub(x_1), \ldots, ub(x_{i-1}), v_i, least[i+1], \ldots least[n]]$ is a solution of the constraint. First note that $[least[1], \ldots, least[n]]$ is a solution of the constraint, since we only step down by at most one, and $least[j] \in dom(x_j), 1 \leq j \leq n$. Now $least[i] \neq ub(x_i)$ otherwise we would have propagated that $x_i \geq last[i]$ and then $v_i \notin dom(x_i)$. The sequence θ is a solution since $least[i] \leq v_i < ub(x_i)$ is neither too large for the predecessor sequence nor too large for the successor sequence by definition. □

2.2 Incrementality

Unfortunately, successive runs of the propagator are frequently quite wasteful. The propagator will perform a full forward/backward pass, even if neither frontier has changed. We can do better by identifying the circumstances where a frontier may change: (1) the upper-bound of a variable on the upper frontier is decreased, (2) the lower-bound of a variable is increased *above* the lower frontier, or (3) k is removed from the domain of $last[k]$. These changes are also *directional*: if the upper-bound of x_i changes, only variables right of x_i may be affected. Similarly, if the lower frontier is shifted, the change can only cascade to the left. Thus, if we start repairing the frontiers as the result of a change, we can stop whenever the repaired frontier coincides with the existing one.

However, *first* and *last* are not quite enough for an incremental propagator: when $\mathcal{D}(x_i)$ changes, we only want to execute the propagator if one of the frontiers has been affected. However while *first* and *last* tell us for a given value which variable supports it, we need to know for a given variable whether it supports a frontier step.

We thus maintain two persistent arrays, *first_val* and *last_val*, maintaining the invariant $first[k] \leq n \rightarrow first\_val[first[k]] = k$ (similarly for *last* and *last_val*). That is, $first\_val[x_i] = k$ whenever x_i supports the k^{th} step of the upper frontier (but is unconstrained if x_i is not a support). This allows us to quickly determine if changes to $\mathcal{D}(x_i)$ have invalidated the frontier (by testing if $last\_val[x_i] \notin$

$\mathcal{D}(x_i) \wedge last[last\_val[x_i]] = x_i$), while only updating cells along the frontier. The incremental propagator is shown in Figure 4 where wakeup looks at domain changes since last execution and repair_upper and repair_lower exactly reflect the corresponding stage of the basic seq-precede-chain propagator.

```
wakeup(changes)
    for(x_i ∈ changes)
        k := first_val[x_i]
        if(first[k] = x_i ∧ ub(x_i) < k)
            if(repair_upper(k) = FAIL)
                return FAIL
    for(x_i ∈ changes)
        k := last_val[x_i]
        if(last[k] = x_i ∧ k ∉ D(x_i))
            repair_lower(k)
    return OKAY

repair_lower(k)
    i := last[k]
    while(k > 0)
        if(k ∈ D(x_i))
            last[k] := i
            last_val[i] := k
            if(first[k] = i)
                propagate ⟨x_i ≥ k⟩
            k := k − 1
            if(last[k] < i)
                return
        i := i − 1
```

```
repair_upper(k)
    i := first[k]
    lim := last[k]
    while(i < lim)
        if(ub(x[i]) > k)
            propagate ⟨x[i] ≤ k⟩
        if(k ∈ D(x[i]))
            first[k] := i
            first_val[i] := k
            k := k + 1
            if(i < first[k])
                return OKAY
        i := i + 1
    if(i < N)
        ▷ First occurrence of k is too late.
        return FAIL
```

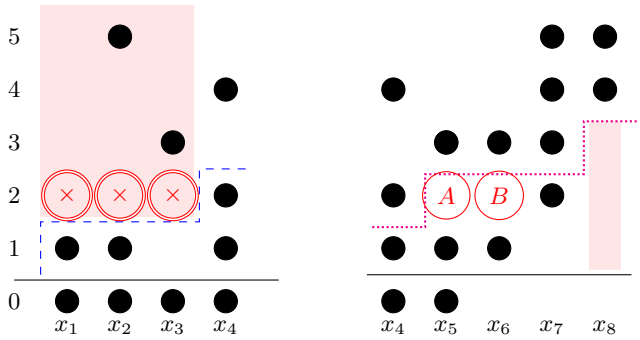
Fig. 4: Incremental propagation algorithms for SEQ-PRECEDE-CHAIN

2.3 Explanation

For integration into lazy clause generation [9], every propagator must be able to *explain* itself: for each inference a it make, the propagator p must be able to produce a set of antecedents E such that $\mathcal{D} \Rightarrow E$, and $c \wedge E \Rightarrow a$.

The SEQ-PRECEDE-CHAIN propagator performs two kinds of propagation: on the forward pass, it tightens upper bounds to the reachable frontier. On the backward pass, it fixes values which we discover *must* be on the frontier.

Explaining $x_i \leq k$ is straightforward: if x_i is greater than j, some variable to the left of x_i must take the value k. Thus $\bigwedge_{j=1}^{i-1} x_j \neq k \rightarrow x_i \leq k$. However, if there are no occurrences of k, we know there can also be no occurrences of values $k' > k$. Thus, we can use the more general explanation $\bigwedge_{j=1}^{i-1} x_j < k \rightarrow x_i \leq k$.



(a) Upper bound explanation (b) Lower bound explanation

Fig. 5: Example explanations for upper- and lower-bound pruning.

Example 3. Recall the upper-bound propagation from Example 1. The explanation for $x_4 \leq 2$ is illustrated in Figure 5(a), and is simply: $\langle x_1 \leq 1\rangle \wedge \langle x_2 \leq 1\rangle \wedge \langle x_3 \leq 1\rangle \rightarrow \langle x_4 \leq 2\rangle$. □

The reasoning for lower-bound propagation is slightly more intricate. Lower bounds are updated when a lower bound pulls the frontier upwards, forcing the upper and lower frontiers to coincide at a variable. The explanation then consists of 3 parts: why the frontier cannot be relaxed to the left, why the lower bound was lifted, and why there is no slack in the frontier.

Example 4. Now consider the lower-bound propagation from Example 1. We must explain why the lower and upper frontiers coincide at $x_4 = 2$. The reason for the upper frontier is given in Example 3.

For the lower frontier, we have to explain why the lower frontier could not be lower. Hence we need to prevent any step downs which did not occur from the tight lower bound, until the propagation. For the explanation of the example we need the tight bound $\langle x_8 \geq 4\rangle$ and the prevention of step downs $\langle x_6 \neq 2\rangle$ and $\langle x_5 \neq 2\rangle$. Note that position x_7 where there is a step down is not relevant. The full explanation is $\langle x_1 \leq 1\rangle \wedge \langle x_2 \leq 1\rangle \wedge \langle x_3 \leq 1\rangle \wedge \langle x_5 \neq 2\rangle \wedge \langle x_6 \neq 2\rangle \wedge \langle x_8 \geq 4\rangle \rightarrow \langle x_4 \geq 2\rangle$. □

In general the procedure can be formalized using the equations below.

$$explain(\langle x_i \leq k\rangle) = \bigwedge_{j=0}^{i-1} \langle x_i < k\rangle$$

$$explain(\langle x_i \geq k\rangle) = ex_L(x_{i+1}, k) \wedge \bigwedge_{j=0}^{i-1} \langle x_i < k\rangle$$

$$\textbf{where } ex_L(x_i,k) = \begin{cases} \langle x_i \geq k \rangle & \textbf{if } \mathsf{lb}(x_i) \geq k \\ ex_L(x_{i+1}, k+1) & \textbf{if } k \in \mathcal{D}(x_i) \\ \langle x_i \neq k \rangle \wedge ex_L(x_{i+1}, k) & \textbf{otherwise} \end{cases}$$

3 Domain consistent decomposition for seq-precede-chain

Given the nature of the propagation for the SEQ-PRECEDE-CHAIN(X) constraint we devised a simple decomposition, introducing intermediate variables $H_0, \ldots, H_n$:

$$\text{SEQ-PRECEDE-CHAIN}([x_1, \ldots, x_n]) = \exists H_0, H_1, \ldots, H_n. \; \begin{array}{l} H_0 = 0 \\ \wedge \bigwedge_{i=1}^{n} \; H_i \leq H_{i-1} + 1 \; \wedge \\ \qquad H_i = \max(x_i, H_{i-1}) \end{array}$$

The intuition is that H_i stores the *highwater mark* u of the increasing subsequence $1, \ldots, u$ seen in the x values from positions 1 to i. A MiniZinc definition (slightly different to avoid using H_0) is given below.

```
predicate seq_precede_chain(array[int] of var int: X) =
      let { int: l = lb_array(X); % least possible value
            int: u = ub_array(X); % greatest possible value
            int: f = min(index_set(X));
            array[index_set(X)] of var l..u: H; } in
      H[f] <= 1 /\ H[f] = max(X[f], 0) /\
      forall(i in index_set(X) diff {f} )
            (H[i] <= H[i-1] + 1 /\ H[i] = max(X[i], H[i-1]));
```

Example 5. Consider the problem of Example 1. Passing forward across the constraints sets $H_0 = 0$, $\mathcal{D}(H_1) = \mathcal{D}(H_2) = \mathcal{D}(H_3) = \{0, 1\}$, propagating $x_2 \leq 1$ and $x_3 \leq 1$, $\mathcal{D}(H_4) = \{0, 1, 2\}$, propagating $x_4 \leq 2$, $\mathcal{D}(H_5) = \mathcal{D}(H_6) = \{0, 1, 2, 3\}$, $\mathcal{D}(H_7) = \{0, 1, 2, 3, 4\}$ propagating $x_7 \leq 4$. and $dom(H_8) = \{4, 5\}$. Now passing backwards over the constraints sets $\mathcal{D}(H_7) = \mathcal{D}(H_6) = \{3, 4\}$, then $\mathcal{D}(H_5) = \{3\}$ and $\mathcal{D}(H_4) = \{2\}$ which propagates $x_4 \geq 2$ and finally $\mathcal{D}H_3 = \{1\}$. The decomposition mimics the global where $dom(H_i) = \{low..up\}$ for the values reached after the i^{th} iteration. □

We now show that this decomposition is correct and maintains domain consistency if the max propagator and $\leq$ propagators are domain consistent, and we do not branch to introduce holes in domains of H variables.

Theorem 3. *The decomposition for* SEQ-PRECEDE-CHAIN(X) *is correct.*

Proof. Given a solution $\theta = [v_1, \ldots, v_n]$ of SEQ-PRECEDE-CHAIN(X). We show that the decomposition never removes it. Now ignoring the constraints $H_i \leq H_{i-1} + 1$ we can inductively show that the max constraints enforce $H_i = \max_{j \in 1..i}(v_j)$. The max constraints are satisfied. Suppose the inequalities were violated then the maximum value before $i - 1$ is $H_{i-1} < v_i - 1$ and hence θ is not a solution. □

Lemma 1. *The decomposition maintains $\mathcal{D}(H_i)$ as range domains, assuming these variables are not branched on to introduce holes.*

Proof. The proof is by induction. Assuming all domains of H_i are range domains, we show that no propagation can cause a hole to be created. First $H_i \leq H_{i-1}+1$ can never create holes. Consider $H_i = \max(x_i, H_{i-1})$ this may create a hole in the domain of H_i if there is hole in $v \notin \mathcal{D}(x_i)$. Now either $v < lb(H_{i-1})$ in which case $lb(H_i) = lb(H_{i-1})$ and there is no hole, or $v \geq lb(H_{i-1})$ in which case either $v \leq ub(H_{i-1})$ and then no hole is punched since $v \in \mathcal{D}(H_{i-1})$. The remaining case is $v > ub(H_{i-1})$ then $ub(x_i) > v$ and $ub(H_i) = ub(x_i)$ but this would violate the constraint $H_i \leq H_{i-1} + 1$ so it would reduce the upper bounds to $ub(H_{i-1})$ and no hole is created. □

Lemma 2. *The constraints $H_i \leq H_{i-1} \wedge H_i = \max(x_i, H_{i-1})$ maintain domain consistency of the conjunction assuming $\mathcal{D}(H_i)$ and $\mathcal{D}(H_{i-1})$ are range domains.*

Proof. We first show that an arbitrary $v \in \mathcal{D}(x_i)$ can be extended to a solution. If $v = ub(H_i)$ then we have solution (H_{i-1}, x_i, H_i) of $(v-1, v, v)$, and $v - 1 = ub(H_{i-1})$ by propagation of the inequality. If $\exists v' \in \mathcal{D}(H_i) \cap \mathcal{D}(H_{i-1}), v' \geq v$ we have solution (v', v, v'). Otherwise we have $\forall v' \in \mathcal{D}(H_i) \cap \mathcal{D}(H_{i-1}), v' < v$. Since they are range domains this means that $ub(H_{i-1}) < v$ and thus $ub(H_i) \leq v$, which implies $v = ub(H_i)$.

Next we show that an arbitrary value $v \in \mathcal{D}(H_{i-1})$ can be extended to a solution. If $v' = lb(x_i) \leq v$ then (v, v', v) is a solution since the max constraint ensures $v \in \mathcal{D}(H_i)$. Otherwise if $v' = v + 1$ then (v, v', v') is a solution since the max constraint ensures $v' \in \mathcal{D}(H_i)$. Otherwise $v' > v + 1$, but then $lb(H_i) = v'$ and the inequality would have removed v from the $\mathcal{D}(H_{i-1})$.

Finally we show that an arbitrary value $v \in \mathcal{D}(H_i)$ can be extended to a solution. Now $v' = lb(x_i) \leq v$ hence either $v' < v$ in which case (v, v', v) is a solution since the max constraint ensures $v \in \mathcal{D}(H_{i-1})$. Or $v' = v$ and either $v \in \mathcal{D}(H_{i-1})$ leading to solution (v, v, v) or $v > ub(H_{i-1})$ meaning $v = ub(H_i)$ and hence $(v - 1, v', v)$ is a solution. □

Theorem 4. *The decomposition for* SEQ-PRECEDE-CHAIN*(X) enforces domain consistency, assuming no hole punching branching on H variables.*

Proof. Lemma 1 shows that the decomposition will always have range domains for H_i if holes are not punched by branching. Lemma 2 shows that the constraint $c(H_{i-1}, x_i, H_i) = H_i \leq H_{i-1} + 1 \wedge H_i = \max(x_i, H_{i-1})$ is domain consistent assuming range domains for H. Since the decomposition treating $c(H_{i-1}, x_i, H_i)$ as a single constraint is Berge acyclic, the decomposition enforces domain consistency. □

Note that the decomposition $c(H_{i-1}, x_i, H_i)$ does *not* maintain domain consistency if we permit holes in the H domains.

Example 6. Consider $c(H_{i-1}, x_i, H_i)$, with variable domains $\mathcal{D}(H_{i-1}) = \{1, 4\}$, $\mathcal{D}(x_i) = [1, 5]$, $\mathcal{D}(H_i) = \{1, 3, 5\}$.

The value $H_i = 3$ is infeasible, as neither 2 nor 3 are in the domain of H_{i-1}. However $H_i \leq H_{i-1} + 1$ is satisfied by assignment $(4, \_, 3)$, and $H_i = \max(x_i, H_{i-1})$ is satisfied by $(1, 3, 3)$.

The value $x = 3$ is also infeasible, as $x > 2$ would require H_{i-1} to take value 4, which forces H_i to be 5. But $H_{i-1} = 4 \wedge H_i = 5$ forces $x_i = 5$, which contradicts our x_i assignment. However $H_i \leq H_{i-1} + 1$ is independent of x, and $H_i = \max(x_i, H_{i-1})$ is satisfied by $(1, 3, 3)$. □

The advantage of the decomposition over the global is that it is incremental by its nature. For a learning solver there is another advantage, explanations for propagation and failure can use the H variables, this allows the summary of lots of previous behaviour in a way that is reusable. For example given the lower bound propagation in Example 1 is explained by

$$\langle H_3 \leq 1 \rangle \wedge \langle H_4 \geq 2 \rangle \rightarrow \langle x_4 \geq 2 \rangle .$$

There could be many reasons why $H_4 \geq 2$ rather than the particular history shown in Figure 5(b).

4 Mapping value-precede-chain to seq-precede-chain

While the case for SEQ-PRECEDE-CHAIN may seem somewhat restricted we can use it to model arbitrary VALUE-PRECEDE-CHAIN constraints by using a mapping to the case that SEQ-PRECEDE-CHAIN supports.

A MiniZinc decomposition for the encoding is given below. Because the mapping using ELEMENT is domain consistent (including the view used to encode $X[i] - l + 1$), the resulting decomposition is also domain consistent, since the constraint structure is a tree [10].

```
predicate value_precede_chain(array[int] of int: T, array[int] of var int: X)
  = if min(index_set(T)) = 1 /\ forall(i in index_set(T))(T[i] = i)
       /\ max(T) = ub_array(X) then seq_precede_chain(X)
    else
    let { int: l = lb_array(X);
          int: u = ub_array(X);
          array[1..u-l+1] of int: p
              = [ sum(i in index_set(T) where T[i] = j)(i)
                | j in l..u ];
          array[index_set(X)] of var 0..length(T): Y;
    } in
          forall(i in index_set(X))
                (element(X[i] - l + 1, p, Y[i])) /\
          seq_precede_chain(Y)
    endif;
```

Example 7. Consider the constraint VALUE-PRECEDE-CHAIN$([2, -2, 1, -1], X)$ where X variables range over values $-3..3$, then we use element constraints of the form ELEMENT$(x_i{+}4, [0,2,4,0,3,1,0], y_i)$ to map X to Y, together with SEQ-PRECEDE-CHAIN(Y) to encode this constraint. □

5 Experimental Results

We implemented the new propagator and decomposition in CHUFFED [11], a lazy clause generation CP solver. For comparison, we also tested several alternate encodings of VALUE-PRECEDE-CHAIN:

dec Decomposition into $k-1$ VALUE-PRECEDE constraints, enforced with the propagator described in [1].
ddec As `dec`, but the VALUE-PRECEDE constraints encoded using the standard `MiniZinc` decomposition. [3]
reg As a finite-state automaton of Figure 2, enforced using an incremental MDD-based propagator [6].
chain The new propagator given in Section 2.
seq The new decomposition given in Section 3.

The modified version of `chuffed`, MiniZinc models and data files are available at `https://github.com/gkgange/chuffed/releases/tag/data-2018-cp`.

5.1 Concert Hall

The *concert hall scheduling problem* [12] considers a set H of identical halls, and a set C of concerts each having a start time s_c, end time e_c and profit p_c. Each concert may either be allocated to a hall $h \in H$, or not scheduled. As the halls are interchangeable, we break symmetries by imposing a SEQ-PRECEDE-CHAIN constraint over the set of assignments. We also add dominance-breaking constraints, to prefer shorter, more profitable concerts. Branch first on the set of concerts to include (including concerts ordered by $\frac{p_c}{e_c - s_c + 1}$), then assign these concerts in input order to the first available hall.

Figure 6 reports performance of the five SEQ-PRECEDE-CHAIN implementations on concert hall scheduling problems, with and without lazy clause generation. It is clear that dec, decomposition into VALUE-PRECEDE propagators, is at all not competitive. Without learning, chain and seq are clearly superior. With learning (where we omit the uncompetitive dec, those results being far outside the current chart bounds), the gap closes dramatically, with ddec nearly catching the new methods, and reg slightly behind.

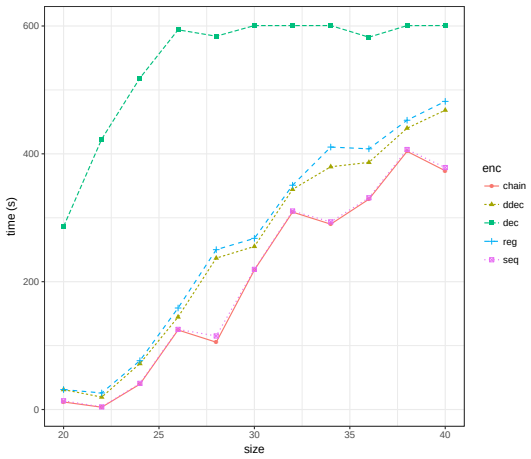
A direct comparison of seq and chain is given in Figure 7. Interestingly, although `chuffed`'s propagator for $\text{Z} = \text{MAX}(\text{X}, \text{Y})$ is only bounds((Z)) consistent, with learning disabled the propagator and decomposition always explored the same number of nodes. The incremental propagator was up to 30% faster than the decomposition, but typically closer to 10%.

Enabling learning introduces a good deal of variance, as the changes in nogoods can result in dramatic changes in search. Nevertheless, we see similar results: the incremental propagator is typically slightly faster than the decomposition, both of which consistently outperform existing encodings.

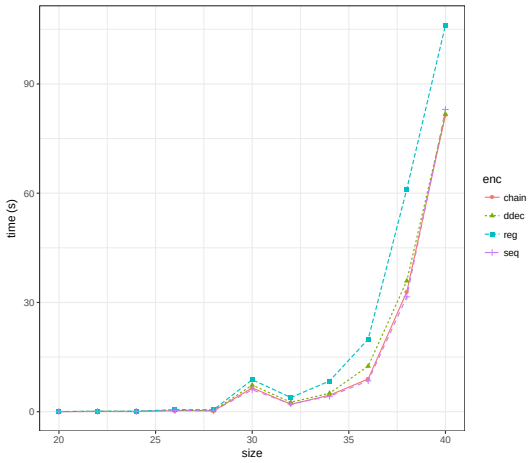
5.2 Capacitated Concert Hall

To evaluate these techniques on domains with multiple sets of indistinguishable values, we extend the concert hall problem with *capacities*: each offer now has a minimum size requirement, and each hall has a maximum capacity (and the 'not scheduled' hall can fit any concert). We then add the constraint:

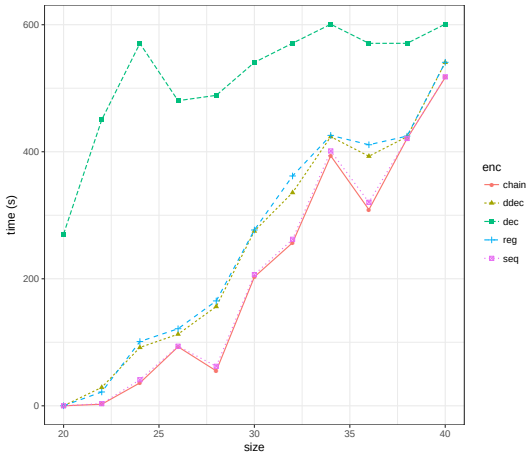
[3] That is, $\exists\, b_1, \ldots, b_n.\ ((b_1 \leftrightarrow x_1 = s) \wedge \bigwedge_{i=2}^{n} b_i \leftrightarrow (b_{i-1} \vee \langle x_{i-1} = s \rangle) \wedge \langle x_i = t \rangle \rightarrow b_{i-1}$. Here b_i records whether there is an occurrence of s no later than x_i.



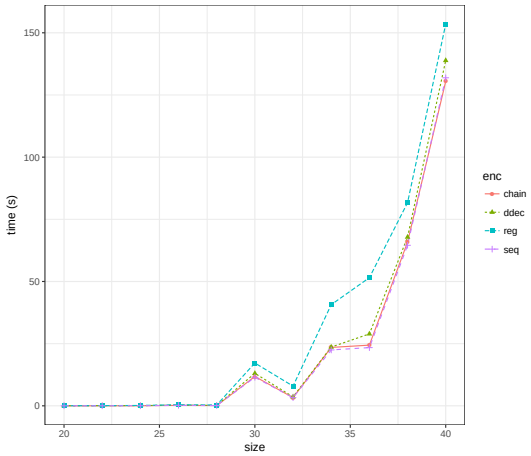
$|H| = 10$, without learning



$|H| = 10$, with learning



$|H| = 14$, without learning

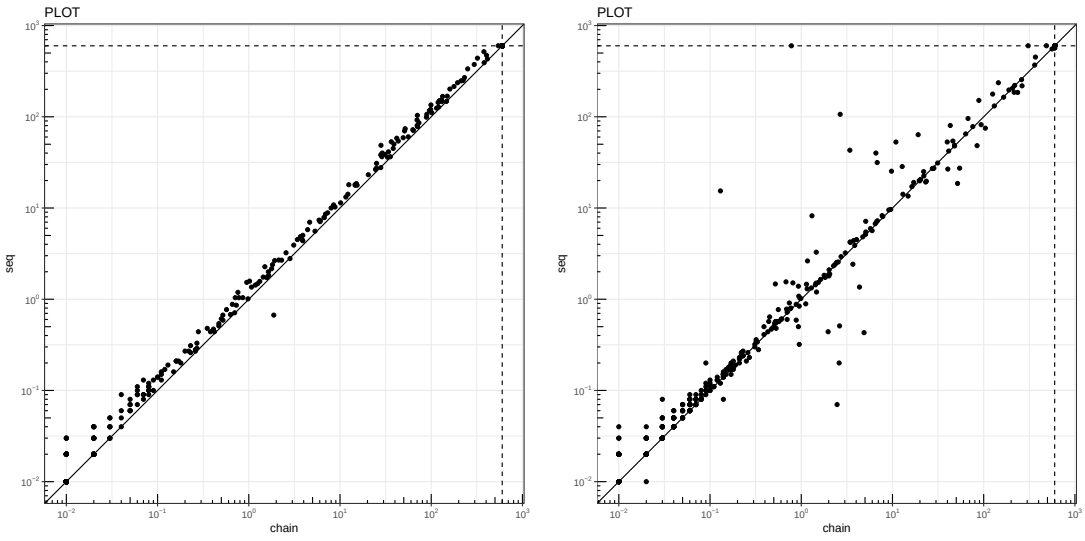


$|H| = 14$, with learning

Fig. 6: Comparison of SEQ-PRECEDE-CHAIN implementations, for different sizes of H. We omit DEC from plots for learning, being totally non-competitive.

```
constraint forall (o in Offers) (size[o] <= capacity[assign[o]] );
```

While the halls are no longer indistinguishable, they may still be partitioned into equivalence classes based on the set of concerts which they may hold. We break this symmetry by adding a VALUE-PRECEDE-CHAIN constraint for each



(a) without learning (b) with learning

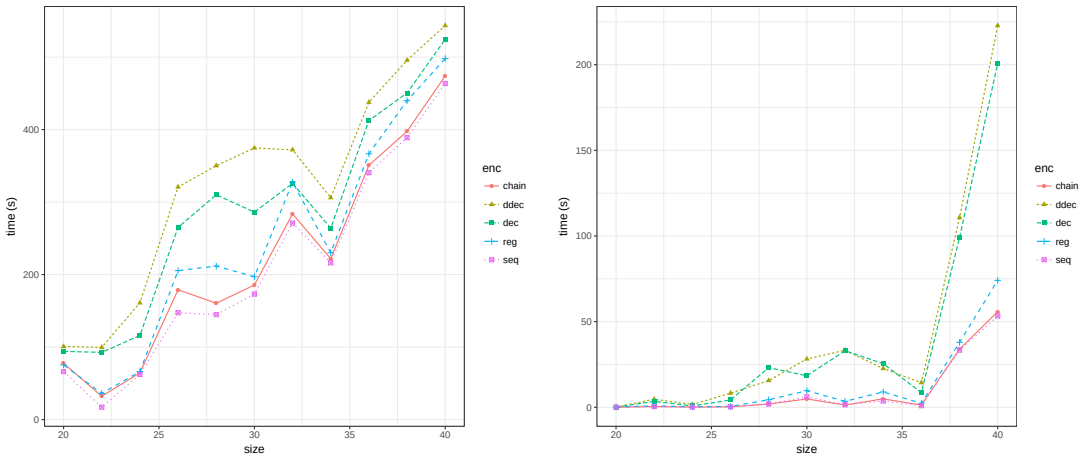
Fig. 7: Comparison of incremental SEQ-PRECEDE-CHAIN propagator versus the sequential decomposition, on concert hall scheduling instances (a) without and (b) with learning.

equivalence class of halls. When using seq and chain these are mapped to a SEQ-PRECEDE-CHAIN constraints using the decomposition of Section 4.

We extended each instance used in the previous section with a capacity and size for each hall and offers. We generate capacities in a similar manner as the other parameters described in [1]: we generate each equivalence-class of halls by first selecting the class size uniformly in $[1, r]$, then choose the capacity uniformly in the range $[200, 1000]$. This is repeated until the m hall capacities have been generated. We generate concert sizes using the same procedure. [4] Search follows the same strategy as for the previous problem: include concerts by profitability, then assign to the first hall in input order.

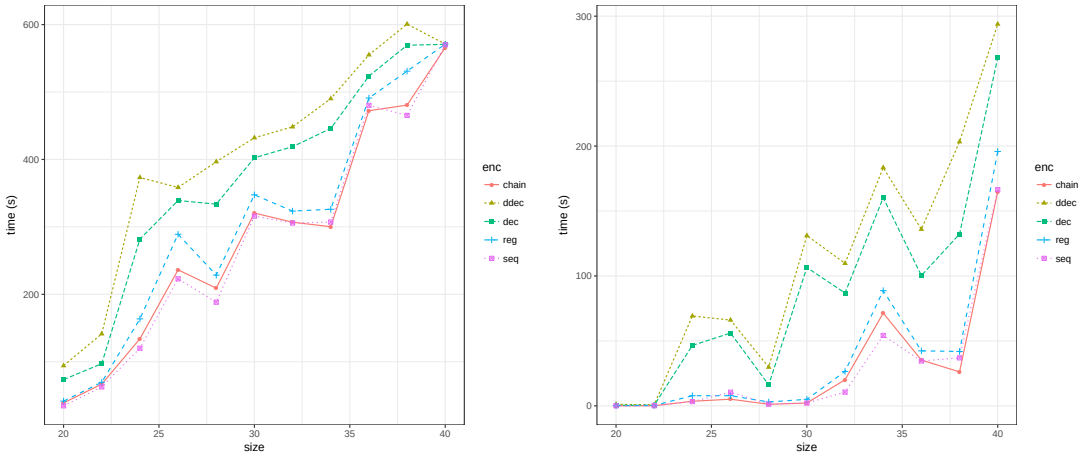
Figure 8 illustrates the performance of each encoding on the capacitated concert hall instances. On these problems, the differences between encodings are more pronounced – interestingly, here DDEC performs worse than DEC, which remains uncompetitive; CHAIN and SEQ again outperform REG. On these problems, we did observe differences in backtrack counts, but SEQ nevertheless appears to be the more robust approach (except on instances where both methods complete in < 0.1s, where initialisation time dominates).

[4] Note that the size equivalences are generated independently of other instance parameters, so may break existing dominance relationships.



(a) `|H| = 10`, without learning

(b) `|H| = 10`, with learning



(a) `|H| = 14`, without learning

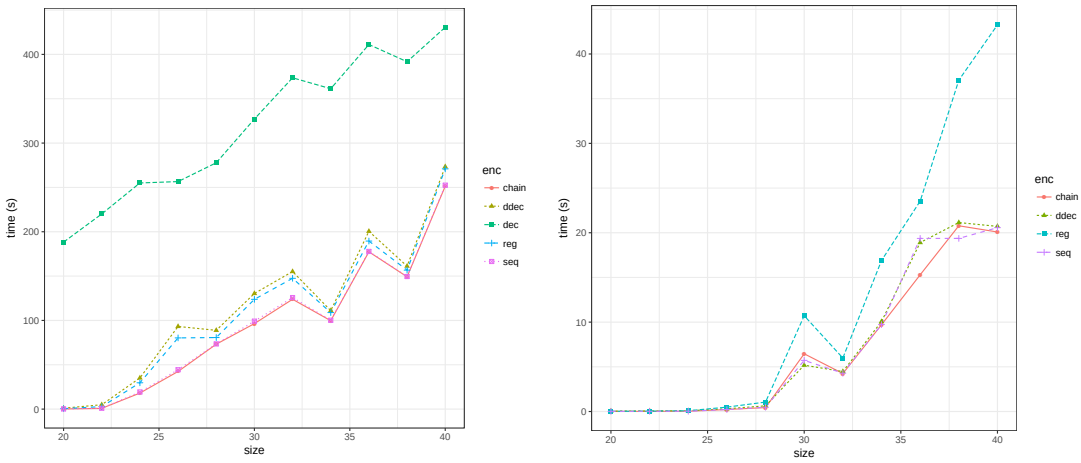
(b) `|H| = 14`, with learning

Fig. 8: Mean time to solve capacitated concert hall scheduling instances for 10 halls (above) and 14 (below) for each VALUE-PRECEDE-CHAIN implementation.

5.3 Graph Colouring

We also evaluated the VALUE-PRECEDE-CHAIN implementations on the graph colouring problems from [12]. In these problems, vertices are partitioned into equivalence classes. Each class is either complete or empty; and if there is an

edge between any members of two equivalence classes, there are edges between all pairs of members of the two classes. For search, we simply assign the minimum colour to vertices in input order. The results are shown in Figure 9. Without



(a) without learning

(b) with learning

Fig. 9: Mean time required to solve graph colouring instances, using each VALUE-PRECEDE-CHAIN implementation. Results for DEC are again omitted in (b).

learning, most of the solving time is spent in disequality reasoning, so differences between encodings are less pronounced. Even so, the new encodings are consistently faster. With learning enabled, we see the performance of REG degrade rapidly as the instance sizes increase.

The main lesson we take from these experiments is that SEQ is the fastest, most robust encoding of VALUE-PRECEDE-CHAIN for both learning and non-learning solvers. The domain-consistent propagator CHAIN has similar performance, but does not outperform SEQ enough to justify its implementation.

6 Conclusion

We define SEQ-PRECEDE-CHAIN to encode the most common usages of VALUE-PRECEDE-CHAIN. We define a new efficient global propagator for this constraint, and how to explain its propagations. This led us to an efficient decomposition, which actually creates more reusable explanations, and is competitive in propagation speed with the global because it is naturally incremental. We propose that the standard decomposition for VALUE-PRECEDE-CHAIN in the MiniZinc library make use of our decomposition.

References

1. Law, Y.C., Lee, J.H.: Global constraints for integer and set value precedence. In Wallace, M., ed.: Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings. Volume 3258 of Lecture Notes in Computer Science., Springer (2004) 362–376
2. Beldiceanu, N., Carlsson, M., Régin, J., Demassey, S.: Global constraint catalogue: `int_value_precede_chain`. `http://sofdem.github.io/gccat/gccat/Cint_value_precede_chain.html` Accessed Apr 2018.
3. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming. Volume 3258. (2004) 482–495
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. (2008) 509–523
5. Perez, G., Régin, J.: Improving GAC-4 for table and MDD constraints. In O'Sullivan, B., ed.: Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings. Volume 8656 of Lecture Notes in Computer Science., Springer (2014) 606–621
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.: Reformulating global grammar constraints. In van Hoeve, W.J., Hooker, J.N., eds.: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27-31, 2009, Proceedings. Volume 5547 of Lecture Notes in Computer Science., Springer (2009) 132–147
8. Abío, I., Gange, G., Mayer-Eichberger, V., Stuckey, P.J.: On CNF encodings of decision diagrams. In: Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings. (2016) 1–17
9. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3) (2009) 357–391
10. Freuder, E.C.: A sufficient condition for backtrack-free search. J. ACM **29**(1) (January 1982) 24–32
11. Chu, G.: Improving Combinatorial Optimization. PhD thesis, Department of Computing and Information Systems, University of Melbourne (2011)
12. Law, Y.C., Lee, J.H.: Symmetry breaking constraints for value symmetries in constraint satisfaction. Constraints **11**(2–3) (2006) 221–267