Rule-based constraint propagation: theory and applications
Brand, S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 8

Array Constraint Propagation

8.1 Introduction

Many problems can be modelled advantageously using look-up functionality: associate each item in a group of items with a unique identifier, or index, and make items directly accessible by their respective index. In mathematics, indices on variables are ubiquitous, and functions are used to uniquely map arguments to values. In programming languages, the corresponding construct is usually called array. In an imperative language such as C, we might define an array of integer variables by integer \( a[3] \), or an array of constants by \( a[] = \{5, 7, 9\} \); we can then access the element at position \( i \) by writing \( a[i] \).

In such languages, the condition for these look-up expressions to be valid is that the index is known when the expression is evaluated. It is in the spirit of constraint programming to relax this restriction. We view \( x \) and \( y \) as variables constrained by the equality \( x = a[y] \) which involves an array \( a \).

A corresponding binary constraint named element was originally developed within the CHIP system, one of the earliest constraint programming systems, [Dincbas et al., 1988]. element proved to be very useful in modelling; many problems (scheduling, resource allocation, etc.) formulated as CSPs make use of it, and most contemporary constraint programming systems provide it now. Sometimes it is generalised so as to allow the one-dimensional array to consist of variables instead of constants. Array constraints and element are examples of so-called 'global' constraints [Beldiceanu and Contejean, 1994, Beldiceanu, 2000a].

Another point motivating the study of array constraints lies in the on-going development in constraint programming research to lift the notion of a constrained variable from the conventional numeric or simple finite-domain variable to higher-structured objects, such as vectors, sets [Gervet, 1997], multisets [Walsh, 2003]. Arrays connect to the notion of a function variable [Hnich, 2003]. In this view, an array mapping the indices in \( I \) to variables ranging over \( A \) is itself a single variable whose domain is the set of functions from \( I \) to \( A \).
Chapter 8. Array Constraint Propagation

Figure 8.1: Crossing entries in a crossword puzzle

Let us demonstrate the use of arrays in modelling.

8.1.1. Example. We formulate the problem of crossword puzzle generation using array constraints [Beacham et al., 2001]. Given a list of words and an empty crossword puzzle grid, the task is to fill the horizontal and vertical entries in the grid with words of appropriate length such that crossing entries agree on the letter at the crossing position. Figure 8.1 shows such a crossing.

We view the entries as variables $w_i$. Their domain is the respective set of words of appropriate size, i.e., the domain of $w_1$ in the figure is the set of 5-letter words.

We use a constant two-dimensional array `letter` to associate words with their letters. The first index denotes the word, the second denotes the position of a letter in that word, e.g., `letter[sail,2] = a`. Every crossing of two entries contributes an array constraint. For example,

$$letter[w_1, 4] = letter[w_2, 2]$$

captures the crossing of Fig. 8.1.

Solely establishing GAC on the array constraints solves some instances of the crossword problem without any search; see [Hentenryck, 1989, p. 140], in which an equivalent special constraint for crossing entries is used.

8.1.2. Example. In Chapter 9, we discuss qualitative spatial reasoning using an array-based model. In this approach, we map tuples of objects to their spatial relation. For example, the relative orientation of point triples is represented as a three-dimensional array `OrRel` indexed by points $p_i$. The set of qualitative relations \{between, behind, in_front, left, \ldots\} is the domain of the array elements. We specify by `OrRel[a,b,c] = in_front` that the continuation of the directed line $\overrightarrow{ab}$ passes through the point $c$. The constraint

$$OrRel[\text{forward}_A, \text{defender}_B, \text{goal}_B] = \text{in_front},$$

$$\text{defender}_B \in \text{Team}_B$$

in a football context formalises the suboptimal situation for a forward player of team $A$ who is in possession of the ball that some player of team $B$ prevents a direct goal shot.
8.1. Introduction

We study here constraint propagation for array constraints. Arrays can be multidimensional and they can consist of variables, the indices in an array expression can be variables, and the array expression is equated with a variable. We consider propagation establishing generalised arc-consistency and bounds-consistency. We also discuss nested array expressions. Furthermore, we examine a method to transform a multidimensional array constraint into an equivalent one-dimensional array constraint and an auxiliary constraint. Such a transformation is acceptable if GAC is the local consistency to be established by constraint propagation. If BC is to be established, we argue that this is not the case; propagating the original array constraint is then preferable.

8.1.1 Arrays

An array is a representation of a total function. Given a Cartesian product \( I = I_1 \times \cdots \times I_n \) and a set \( \mathcal{A} \) for the function domain and range, respectively, an array \( a \) is a set of atomic mappings that satisfies

for every \( b \in I \) some \( e \in \mathcal{A} \) exists such that \( (b \mapsto e) \in a \).

We use conventional array notation and write

\[ a[b] = e \quad \text{if} \quad (b \mapsto e) \in a. \]

The length \( n \) of \( I = I_1 \times \cdots \times I_n \) is the dimensionality of the array. We assume that all \( I_i \) are finite.

8.1.2 Array Constraints

We use arrays in array expressions \( a[b] \) and simple array equations \( e = a[b] \), where \( b \in I, e \in \mathcal{A} \). We lift array equations to array constraints of the form

\[ x = a[y_1, \ldots, y_n] \]

by allowing variables instead of constants, as follows:

- result variable \( x \) with domain \( D_x = \mathcal{A} \),
- index variables \( y_1, \ldots, y_n = y \) with domains \( D_y = I_i \),
- array variables \( a[b] \) for \( b \in I \) with domains \( D_{a[b]} = \mathcal{A} \).

So such an array constraint is a constraint on the sequence of variables

\[ X = x, y_1, \ldots, y_n, \langle a[b] | b \in I \rangle. \quad (8.1) \]

It is of arity \( 1 + n + \prod_{i=1}^n |I_i| \), and therefore highly non-binary.

We assume from now on that all the variables in the sequence \( X \) are pair-wise different. We say that \( a \) is an array of constants if all \( \langle a[b] | b \in I \rangle \) are constants, otherwise we call it an array of variables.
8.2 Constraint Propagation

8.2.1 Propagation Rules for Generalised Arc-Consistency

Simple Array Constraints

We derive GAC-establishing constraint propagation rules for array constraints from the generic rule of Fact 2.2.3. So we are interested in all correct rules of the form

\[ C(x_1, \ldots, x_n) \rightarrow x_i \neq e \]

for an array constraint \( C \). Correctness follows from \( e \notin C[x_i] \).

The variables in an array constraint \( x = a[y_1, \ldots, y_n] \) split in three groups, see Statement (8.1). We examine the correctness condition separately for a representative of each group.

**Variable** \( x \). We require \( e \notin C[x] \), or equivalently \( e \notin a[y_1, \ldots, y_n] \). That is the case exactly if

\[ \forall b \in D_{y_1} \times \cdots \times D_{y_n}. e \in D_{a[b]} \]  

holds for the domains.

**Variable** \( y_k \). The correctness condition is \( b_k \notin C[y_k] \). We find in this case

\[ D_{y|k} = D_{y_1} \times \cdots \times D_{y_{k-1}} \times \{b_k\} \times D_{y_{k+1}} \times \cdots \times D_{y_n}, \]

\[ \forall b \in D_{y|k}. \exists e \in D_x \cap D_{a[b]} \]  

(8.3)

**Variable** \( a[b_1, \ldots, b_n] \). We need a circumstance in which \( e \notin C[a[b_1, \ldots, b_n]] \). That is only the case once the index is fixed to \((b_1, \ldots, b_n)\); then, all other variables \( a[b'_1, \ldots, b'_n] \) are unconstrained. We have thus

\[ \{(b_1, \ldots, b_n)\} = D_{y_1} \times \cdots \times D_{y_n} \land e \notin D_x. \]  

(8.4)

We now instantiate the generic GAC-establishing rule for each variable type and obtain the following three rules:

\[ x = a[y_1, \ldots, y_n] \rightarrow x \neq e \] if (8.2), \( (arr_{gac_x}) \)

\[ x = a[y_1, \ldots, y_n] \rightarrow y_k \neq b_k \] if (8.3), \( (arr_{gac_y}) \)

\[ x = a[y_1, \ldots, y_n] \rightarrow a[b_1, \ldots, b_n] \neq e \] if (8.4), \( (arr_{gac_a}) \)

8.2.1. Theorem. The rules \( (arr_{gac_x}), (arr_{gac_y}), (arr_{gac_a}) \) establish generalised arc-consistency on the array constraint \( x = a[y_1, \ldots, y_n] \).
8.2. Constraint Propagation

**Proof.** Fact 2.2.3, and the preceding derivations of the respective correctness conditions.

**Pair-wise Variable Difference Requirement**

It is indeed necessary to restrict variables to occur just once. Consider the array 
\[\text{xor} = \{(0,0)\mapsto0, (0,1)\mapsto1, (1,0)\mapsto1, (1,1)\mapsto0\}\]
and the constraint 
\[x = \text{xor}[y, y]\]
with \(x \in \{0\}, y \in \{0, 1\}\). It is inconsistent but stable under the rules \((\text{arr\_gac}_x), (\text{arr\_gac}_y), (\text{arr\_gac}_a)\).

**Compound Array Constraints**

We have only admitted array constraints in the simple form \(x = a[y_1, \ldots, y_n]\) so far. It can sometimes be easier, however, to use several arrays in one constraint, such as in 
\[a_1[y_1, \ldots, y_n] = a_2[z_1, \ldots, z_m]\]
or in the nested expression 
\[x = a_3[a_1[y_1, \ldots, y_p], a_2[z_1, \ldots, z_q]]\].

Establishing GAC on such array constraints is generally hard if variables are used in multiple places. If variables occur just once then the compound expressions can simply be decomposed, using fresh auxiliary variables. Lemma 2.1.7 states that GAC on the constraints of the decomposition corresponds to GAC on the compound constraint.

So, for example, the constraint \(\text{letter}[w_1, 4] = \text{letter}[w_2, 2]\) from the crossword example 8.1.1 can be decomposed into the two constraints \(\text{letter}[w_1, 4] = \ell_{1,2}\) and \(\text{letter}[w_2, 2] = \ell_{1,2}\) without affecting propagation.

**Domain Reduction vs. Constraint Transformation**

As instances of the generic GAC-establishing rule in Fact 2.2.3, the rules 
\((\text{arr\_gac}_x), (\text{arr\_gac}_y), (\text{arr\_gac}_a)\)
are domain reduction rules by type. In presence of GAC-establishing constraint propagation rules or algorithms for basic constant/variable equality constraints \(v_1 = v_2\), we can replace the domain reduction rule \((\text{arr\_gac}_a)\) by a constraint propagation rule that does not reduce domains but imposes the entailed equality constraint. Such equality constraints are generally provided in constraint logic programming systems, which implemented them through unification extended with domain intersection.

So we extract just
\[
\{(b_1, \ldots, b_n)\} = D_{y_1} \times \ldots \times D_{y_n}
\]
from correctness condition (8.4), and state the rule
\[x = a[y_1, \ldots, y_n] \rightarrow x = a[b_1, \ldots, b_n] \quad \text{if (8.5).} \quad (\text{arr\_gac}_{a=})
\]
Propagation of the new equality constraint \(x = a[b_1, \ldots, b_n]\) reduces then the domains of the variables \(x\) and \(a[b_1, \ldots, b_n]\) in the same way as \((\text{arr\_gac}_x), (\text{arr\_gac}_a)\).
8.2.2. NOTE. In presence of constraint propagation mechanisms for variable equality constraints, the rules \((\text{arr-gac}_x), (\text{arr-gac}_y), (\text{arr-gac}_{a_1})\) establish GAC on the array constraint \(x = a[y_1, \ldots, y_n]\).

Moreover, observe that, as long as the domains \(D_x, D_{a[y_1, \ldots, y_n]}\) are non-empty, also the rule \((\text{arr-gac}_y)\) is redundant (more precisely: it has no correct instances). So, once \((\text{arr-gac}_{a_1})\) has fired, the original constraint \(x = a[y_1, \ldots, y_n]\) and all its propagation rules can be eliminated from the constraint solver.

8.2.2 Propagation Rules for Bounds-Consistency

Generalised arc-consistency is a strong but often also a computationally expensive local consistency. Depending on the problem, it can be more efficient to propagate less. Bounds-consistency (Def. 2.1.8) is a good candidate for a weaker local consistency notion. Recall that it checks and modifies only the domain bounds of variables. The significant implication for the representation of domains is that domains that are intervals remain intervals, which reduces the space complexity substantially.

For array constraints we obtain propagation rules for bounds-consistency from the rules establishing GAC, see Fact 2.2.3. We restrict correctness condition to domain bounds, i.e.,

\[(\text{arr-gac}_x)\text{ in which } e \in \{\min(D_x), \max(D_x)\}\]

\[(\text{arr-gac}_y)\text{ in which } b_k \in \{\min(D_{y_k}), \max(D_{y_k})\}\]

\[(\text{arr-gac}_a)\text{ in which } e \in \{\min(D_{a[y_1, \ldots, y_n]}), \max(D_{a[y_1, \ldots, y_n]})\}\]

8.2.3. THEOREM. The rules \((\text{arr-bc}_x), (\text{arr-bc}_y), (\text{arr-bc}_a)\) establish bounds-consistency on the array constraint \(x = a[y_1, \ldots, y_n]\). \(\square\)

8.2.3 From Rules to Algorithms

A naive iteration algorithm of the propagation rules establishing BC or GAC is computationally expensive, due to a repetitive access to the same variable domains in the process of verifying the correctness conditions (8.2), (8.3), (8.4).

In Figure 8.2, we give propagation algorithms for BC and GAC which implement the rule iteration process. The principle is to start with sets of values \(e\) that are candidates for removal in a body \(x_i \neq e\) of an array constraint propagation rule. The algorithm core loop \texttt{array_prop} deletes all those values for which the corresponding propagation rule is incorrect. Subsequently, the remaining values can correctly be removed from the respective variable domains.

We presume that basic equality constraints are provided by the underlying constraint programming platform, and pose an equality constraint as soon as correct by condition (8.5), instead of reducing domains; see Section 8.2.1.
8.2. Constraint Propagation

\framebox{array_gac}: array constraint \iff equivalent GAC-reduced constraint
\[ (XU, YU_{1..n}) = \text{array-prop}(D_x, D_{y_1}, \ldots, D_{y_n}) \]
\[ D_x := D_x - XU \]
\[ D_{y_i} := D_{y_i} - YU_i, \text{ for all } i \in [1..n] \]
if \( \{(b_1, \ldots, b_n)\} = D_{y_1} \times \cdots \times D_{y_n} \) then constrain \( x = a[b_1, \ldots, b_n] \)

\framebox{array_bc}: array constraint \iff equivalent BC-reduced constraint
let \( \text{bds}(D) = \{\min(D), \max(D)\} \)
\[ XS := \emptyset \] // supported values
\[ YS_i := \emptyset, \text{ for all } i \in [1..n] \]
repeat
\[ XT := \text{bds}(D_x) \setminus XS \] // values to be tested
\[ YT_i := \text{bds}(D_{y_i}) \setminus YS_i, \text{ for all } i \in [1..n] \]
\[ (XU, YU_{1..n}) = \text{array-prop}(XT, YT_{1..n}) \] // unsupported values
\[ XS := XS \cup (XT - XU) \]
\[ YS_i := YS_i \cup (YT_i - YU_i), \text{ for all } i \in [1..n] \]
\[ D_x := D_x - XU \]
\[ D_{y_i} := D_{y_i} - YU_i, \text{ for all } i \in [1..n] \]
until \( XU = \emptyset \) and \( YU_i = \emptyset, \text{ for all } i \in [1..n] \)
if \( \{(b_1, \ldots, b_n)\} = D_{y_1} \times \cdots \times D_{y_n} \) then constrain \( x = a[b_1, \ldots, b_n] \)

\framebox{array_prop}: domain values \( XU, YU_{1..n} \iff \) unsupported domain values
\[ B := D_{y_1} \times \cdots \times D_{y_n} \]
while \( B \neq \emptyset \) and \( YU_k \neq \emptyset \) for some \( k \in [1..n] \) do
choose \( (b_1, \ldots, b_n) \in B \) such that \( b_k \in YU_k \) for some \( k \in [1..n] \)
remove \( (b_1, \ldots, b_n) \) from \( B \)
if \( D_x \cap a[b_1, \ldots, b_n] \neq \emptyset \) then \( YU_i := YU_i \setminus \{b_i\}, \text{ for all } i \in [1..n] \)
\[ XU := XU \setminus a[b_1, \ldots, b_n] \]
end
while \( B \neq \emptyset \) and \( XU \neq \emptyset \) do
choose and remove \( (b_1, \ldots, b_n) \) from \( B \)
\[ XU := XU \setminus a[b_1, \ldots, b_n] \]
end
return \( (XU, YU_{1..n}) \)

Figure 8.2: Propagation for array constraints
8.2.4. NOTE. Algorithm \texttt{array\_gac} establishes GAC, and algorithm \texttt{array\_bc} establishes BC, on the array constraint $x = a[b_1, \ldots, b_n]$. 

Let us examine the working of the GAC-enforcing propagation.

8.2.5. EXAMPLE. Consider $x \in \{B, C, D\}$ and $y_1 \in \{1, 2\}$, $y_2 \in \{1, 2, 3\}$ in the constraint $x = a[y_1, y_2]$, and let $a$ be defined as the array of constants

\[
\begin{array}{c|ccc}
  a[y_1, y_2] & 1 & 2 & 3 \\
  \hline
  1 & A & B & C \\
  2 & D & E & F \\
\end{array}
\]

The constraint $x = a[y_1, y_2]$ is GAC, which \texttt{array\_gac} verifies by calling \texttt{array\_prop} with $XU = \{B, C, D\}$ and $YU = \{(1, 2), (1, 2, 3)\}$.

Initially, the set of indices $B$ is $\{1, 2\} \times \{1, 2, 3\}$. We iterate through $B$ (choose statement) from lexicographically small to large indices.

1. $D_{a[1,1]} = \{A\}$ is evaluated, but no changes to $XU$, $YU$ result.
2. $D_{a[1,2]} = \{B\}$ follows. We have $XU = \{C, D\}$ and $YU = (\{2\}, \{1, 3\})$.
3. $D_{a[1,3]} = \{C\}$ is read, which results in $XU = \{D\}$ and $YU = (\{2\}, \{1\})$.
4. Only $(2, 1)$ remains in $B$, so $D_{a[2,1]} = \{D\}$ is looked up. $XU = \emptyset$ and $YU = (\emptyset, \emptyset)$ remain.

Only one incomplete run is needed; the indices $(2, 2), (2, 3)$ permissible by the domains of $y_1, y_2$ are skipped.

Observe that an alternative iteration strategy with less steps exists in this case. Suppose $(2, 1)$ had been chosen first, then only $(1, 2)$ and $(1, 3)$ could be chosen next, and $(1, 1), (2, 2), (2, 3)$ had been skipped. 

For GAC, the correctness-checking procedure \texttt{array\_prop} iterates through all possible indices $b_1, \ldots, b_n$ in the domains of $y_1, \ldots, y_n$ in the worst case. This situation occurs, for instance, with the constraint and initial domains of Example 8.2.5 except for $x \notin \{F\}$, and if \texttt{array\_prop} iterates through $B$ from small to large indices.

In the best case, the number of iteration steps is the size of the largest domain $D_{y_i}$. Take Example 8.2.5, but with $x \in \{A, E, F\}$. The algorithm iterates through $(1, 1), (2, 2), (2, 3)$ in three steps, corresponding to $|D_{y_2}| = 3$.

8.2.6. NOTE. The number of iteration steps in \texttt{array\_prop} has an upper bound of $O(d^n)$ and a lower bound of $O(d)$, where $d$ is the size of the largest input set of values.
8.3. Decomposing Multidimensional Array Constraints

In the case of array_gac, the input sets are the complete variable domains. In the case of array_bc only the currently unchecked domain bounds are examined by one call to array_prop. If subsequently some bounds are reduced, the process needs to be repeated for the new bounds. Generally, the cost of checking the domain bounds by array_bc will be lower than the cost of checking every domain element by array_gac.

It is useful to remark that the set \( B \) in the procedure array_prop need not be represented extensionally with the resulting high space cost. Instead, a compact iterator/pointer can be maintained that marks the lexicographically next tuple.

8.3 Decoding Multidimensional Array Constraints

Interestingly, a constraint language with one-dimensional array constraints and integer arithmetic constraints is already expressive enough to support multidimensional arrays. We discuss now a method to translate a multidimensional array constraint into a one-dimensional array constraint and an additional constraint. The interest of such a translation lies in the greater simplicity of a propagation algorithm for only one-dimensional array constraints.

We show that decomposing is an acceptable technique when the desired result of propagation is GAC. We argue that this is not the case, however, when we wish to enforce only BC.

8.3.1 Reducing the Array Dimensionality

Let the \( n \)-dimensional array \( a \) represent a total function from the Cartesian product \( \mathcal{I}_1 \times \cdots \times \mathcal{I}_n = \mathcal{I} \) to the set \( \mathcal{A} \). Assume that every component set is a (finite) integer interval, so

\[ \mathcal{I}_i = [0..(m_i - 1)] \quad \text{for all} \ i \in [1..n]. \]

We can do so for our purposes without substantial loss of generality, as any finite set can be mapped to such an interval. We define a mapping \( f \) from \( \mathcal{I} \) to the interval \([0..(\prod_{i=0}^{n} m_i - 1)]\), by

\[ b = f(b_1, \ldots, b_n) = \sum_{i=1}^{n} \left( b_i \cdot \prod_{j=0}^{i-1} m_i \right) \tag{8.6} \]

(we define \( m_0 = 1 \) for convenience).

8.3.1. Example. We map car number plates labels to numbers. Let us assume that a number plate consists of a sequence of six symbols: two letters, two digits,
and again two letters, e.g., RB-18-GH. A letter is taken from the Latin alphabet of 26 letters; we translate it implicitly to a number in the interval \([0..25]\). A number plate \(p \in [0..25]^2 \times [0..9]^2 \times [0..25]^2\) with \(p = (p_1, \ldots, p_6)\) can thus be mapped to a number between 0 and \(26^4 \cdot 10^2 - 1\) by

\[
\overline{p} = f(p) = p_1 + 26p_2 + 26^2p_3 + 10 \cdot 26^2p_4 + 10^2 \cdot 26^2p_5 + 10^2 \cdot 26^3p_6.
\]

This means \(f(R, B, 1, 8, G, H) = 12763599\).

We associate the multidimensional array \(a\) with a new one-dimensional array \(\overline{a}\) by

\[
\overline{a} = \{ f(b) \mapsto e \mid (b \mapsto e) \in a \},
\]

which means that

\[
\overline{a}[f(b_1, \ldots, b_n)] = a[b_1, \ldots, b_n].
\]

Informally speaking, we ‘linearise’ \(a\).

### 8.3.2 Decomposition

We deal now with array constraints (possibly on a multidimensional array) by replacing them by a new array constraint on a one-dimensional array, and an appropriate linear constraint derived from (8.6), linking the respective array indices.

#### 8.3.2. Example

Consider the two-dimensional array \(a\) defined by

<table>
<thead>
<tr>
<th>(a[y_1, y_2])</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

We set up a new one-dimensional array \(\overline{a}\) as follows:

<table>
<thead>
<tr>
<th>(\overline{a}[y])</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1</td>
<td>16</td>
<td>2</td>
<td>17</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

We can then replace the constraint

\(x = a[y_1, y_2]\)

by the two constraints

\(x = \overline{a}[\overline{y}]\) and \(\overline{y} = y_1 + 2y_2\).

where \(\overline{y}\) is a new variable.
8.3. Decomposing Multidimensional Array Constraints

8.3.3 Propagation

Generalised Arc-consistency

Enforcing GAC on the two constraints of the decomposition is equivalent to enforcing GAC on the original array constraint. This is by Lemma 2.1.7, and since the two decomposition constraints share only the new auxiliary index variable. The cost of propagation is, however, not reduced by decomposing array constraints.

8.3.3. FACT. The complexity of establishing GAC on a linear arithmetic equality constraint in $n$ variables is in $O(d^n)$, where $d$ is the size of the largest variable domain.

So we have the same worst-case cost for propagation via the decomposition constraints as for propagation by the array_gac algorithm.

Bounds-Consistency

If we choose bounds-consistency as the desired local consistency notion, we observe that enforcing it on the decomposition is strictly weaker than bounds-consistency on the original array constraint. The problem occurs due to the loss of information exchange between the two constraints, if only bounds-consistency is enforced.

8.3.4. EXAMPLE. Reconsider the two-dimensional array $a$ from Example 8.3.2 and its linearised peer $\bar{a}$. Consider the variables

\[ x \in [1..3], \quad y_1 \in [0..1], \quad y_2 \in [0..2]. \]

The constraint $x = a[y_1, y_2]$ is clearly not bounds-consistent: for that, we must reduce the domain of $y_1$ to the singleton interval $[1..1]$ since $y_1 = 0$ does not occur in any solution. In contrast, the two decomposition constraints

\[ x = \bar{a}[\bar{y}] \quad \text{and} \quad \bar{y} = y_1 + 2y_2 \quad \text{with} \quad \bar{y} \in [1..5] \]

are bounds-consistent.

In conclusion, if we wish to enforce bounds-consistency on a multidimensional array constraint, we should choose the array_bc algorithm instead of decomposing the constraint.
8.4 Implementation

We implemented the algorithms \texttt{array\_gac} and \texttt{array\_bc}, see Fig. 8.2, in the constraint programming system ECL\textsuperscript{PS}\textsuperscript{e} [Wallace et al., 1997], using its finite domain constraints library. The propagation algorithm is provided in a library together with several other array-related functions, consisting of about 600 lines of source code in total.

A specific side effect of the array propagation algorithm can be exploited in an ECL\textsuperscript{PS}\textsuperscript{e} implementations. ECL\textsuperscript{PS}\textsuperscript{e} controls the execution order of constraint propagation algorithms based on changes to the constrained variables, such as a domain reduction. Propagation algorithms ‘watching’ a variable are scheduled to execute once this variable has changed.

The \texttt{array\_prop} procedure allows to extract useful variables to watch, namely the variables \(a[b_1, \ldots, b_n]\) for which \(D_x \cap D_{a[b_1, \ldots, b_n]} \neq \emptyset\). The domains of these watched variables provide support for domain values of other variables. Hence, changes to the watched variables require a repeated propagation round.

Array constraints and the implementation of propagation algorithms are reused in the Chapters 9 and 10 on qualitative reasoning.

8.5 Final Remarks

Related Work

The established precursor of array constraints is the \texttt{element} constraint of CHIP [Dincbas et al., 1988], now available in many constraint programming languages. It is the one-dimensional specialisation, and usually requires the array to be constant.

Algorithms for propagation in the one-dimensional case have been published, for example, [Carlson et al., 1994] describes an AKL(FD) implementation of \texttt{element} using indexicals [Codognet and Diaz, 1996], in which the array can consist of variables. I am not aware of a published algorithm for the multidimensional case.

Array constraints in the constraint programming language OPL [Hentenryck et al., 1999] can be multidimensional and use arrays of variables. It is unclear what form of constraint propagation takes place, but in [Hentenryck et al., 1999, p. 100] it is stated that the reduction for an index variable in a multidimensional array constraint depends on its position. We found experimentally in the OPL implementation available to us that the propagation is weaker than GAC (and BC). For all three cases treated by the rules \((\texttt{arr\_gac}_x)\), \((\texttt{arr\_gac}_y)\), \((\texttt{arr\_gac}_a)\), we could construct simple examples using small 2-dimensional arrays in which reduction of domains is possible but not performed by OPL Studio 3 [ILOG, 2000], see Fig. 8.3.
enum Dz { i, j };  
enum Dy { k, l, m };  
enum Da { p, q, r };  

Da a[Dz, Dy] = # i: #[k:p, l:q, m:r]#,  
    j: #[k:p, l:q, m:r]# ];  
var Da x;  
var Dz z, u;  
var Dy y, v;  
solve { v <> 1;  // OPL Studio          GAC  
a[u, v] = x;  // x in { p, q, r } { p, r }  
            //  
a[z, y] = q;  // y in { k, l, m } { 1 }  
};

enum Dy { i, j, k };  
enum Da { p, q, r };  

var Da a[Dy];  
var Da x;  
var Dy y;  
solve { y = j;  
    x <> q;  // OPL Studio          GAC  
    x = a[y];  // a[j] in { p, q, r } { p, r }  
};

Figure 8.3: OPL programs exhibiting weak propagation in ILOG’s OPL Studio
In [Beldiceanu, 2000b] a constraint called \textit{case} is proposed that subsumes multidimensional array constraints with constant arrays. No algorithm is given. In [Hooker et al., 2000] on combining operations research techniques and constraint satisfaction methods, a continuous relaxation of \textit{element} using a cutting-planes approach is studied. The \textit{element} constraint there corresponds to a one-dimensional array of variables with continuous domains.

\textbf{Conclusions}

We studied here constraint propagation for array constraints. There is ample evidence suggesting that arrays are useful for modelling constraint satisfaction problems. Indices on objects are ubiquitous in mathematics. Arrays with multiple dimensions have long been present in programming languages. The \textit{element} constraint is supported by many constraint systems.

Practical experience shows that the most advantageous notion of local consistency depends on the considered problem. Sometimes a weaker notion such as bounds-consistency may suffice, perhaps just applied in the early stages of search and later replaced by full generalised arc-consistency. We derived constraint propagation rules to achieve generalised arc-consistency and bounds-consistency, and we gave algorithms implementing the rules.

We also examined the option of decomposing a multidimensional array constraint into one with just a one-dimensional array and a linear constraint. We argued that when we wish to establish GAC on array constraints, the composed and the decomposed behave similarly with respect to runtime, while this is not the case when we require only BC. We showed that decomposing a multidimensional array constraint results in a loss of information when just BC is enforced on the sub-constraints of the decomposition. In this case, it is more appropriate to use a BC algorithm, such as the one we propose, on the original non-decomposed array constraint.