Constraint Programming
– Filtering : Part 2 –

Christophe Lecoutre
lecoutre@cril.fr

CRIL-CNRS UMR 8188
Universite d'Artois
Lens, France

January 2021

Outline

1. Tables and MDDs
2. Specific Algorithms for Table Constraints
3. Compact Table
4. Local Consistencies

Outline

1. Tables and MDDs
2. Specific Algorithms for Table Constraints
3. Compact Table
4. Local Consistencies

Recall

CP is about:

1. modeling constrained combinatorial problems under the form of constraint networks (CSPs / COPs)
2. solving such problems by employing inference methods and search strategies

Classically, we use:

- backtrack search
- while maintaining AC (Arc Consistency) at each node

For enforcing AC, all constraints are sollicited in turn for filtering domains (principle called constraint propagation).

It is possible to:

- use a generic propagation scheme, like AC3
- or implement specialized filtering algorithms, one for `allDifferent`, one for `extension`, ...

Table Constraints

Classically, for constraints defined in extension, we use ordinary tables that contain ordinary tuples, as e.g., (a, b, a).

But, many recent developments concern:

- starred (or short) tables, containing the symbol *, as e.g., $(a, *, b)$
- smart tables, a form of hybridization between intensional and extensional constraints
- MDDs (Multi-Valued Decision Diagrams)

Remark.
These different forms are useful when modeling.

Remark.
We need filtering algorithms for both positive and negative forms (tables).

Table Constraints

Classically, for constraints defined in extension, we use ordinary tables that contain ordinary tuples, as e.g., (a, b, a).

But, many recent developments concern:

- starred (or short) tables, containing the symbol *, as e.g., $(a, *, b)$
- smart tables, a form of hybridization between intensional and extensional constraints
- MDDs (Multi-Valued Decision Diagrams)

Remark.
These different forms are useful when modeling.

Remark.
We need filtering algorithms for both positive and negative forms (tables).

Table Constraints

Classically, for constraints defined in extension, we use ordinary tables that contain ordinary tuples, as e.g., (a, b, a).

But, many recent developments concern:

- starred (or short) tables, containing the symbol *, as e.g., $(a, *, b)$
- smart tables, a form of hybridization between intensional and extensional constraints
- MDDs (Multi-Valued Decision Diagrams)

Remark.

These different forms are useful when modeling.

Remark.

We need filtering algorithms for both positive and negative forms (tables).

Starred Tables

Introduction of wildcard symbols (*) in tables (Nightingale *et al.*, 2013)

The constraint $x = y \vee y = z$ can be defined by:

x	y	z
a	a	$*$
b	b	$*$
c	c	$*$
$*$	a	a
$*$	b	b
$*$	c	c

The global constraint **element**$(I, \langle x_1, x_2, \ldots, x_m \rangle, R)$ can be defined by:

I	x_1	x_2	$\ldots$	x_r	R
1	a	$*$	$\ldots$	$*$	a
1	b	$*$	$\ldots$	$*$	b
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
2	$*$	a	$\ldots$	$*$	a
2	$*$	b	$\ldots$	$*$	b
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Starred Tables

Introduction of wildcard symbols (*) in tables (Nightingale *et al.*, 2013)

The constraint $x = y \vee y = z$ can be defined by:

x	y	z
a	a	$*$
b	b	$*$
c	c	$*$
$*$	a	a
$*$	b	b
$*$	c	c

The global constraint **element**$(I, \langle x_1, x_2, \ldots, x_m \rangle, R)$ can be defined by:

I	x_1	x_2	$\ldots$	x_r	R
1	a	$*$	$\ldots$	$*$	a
1	b	$*$	$\ldots$	$*$	b
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
2	$*$	a	$\ldots$	$*$	a
2	$*$	b	$\ldots$	$*$	b
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Starred Tables

Introduction of wildcard symbols (*) in tables (Nightingale *et al.*, 2013)

The constraint $x = y \vee y = z$ can be defined by:

x	y	z
a	a	$*$
b	b	$*$
c	c	$*$
$*$	a	a
$*$	b	b
$*$	c	c

The global constraint **element**($I, \langle x_1, x_2, \ldots, x_m \rangle, R$) can be defined by:

I	x_1	x_2	$\ldots$	x_r	R
1	a	$*$	$\ldots$	$*$	a
1	b	$*$	$\ldots$	$*$	b
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
2	$*$	a	$\ldots$	$*$	a
2	$*$	b	$\ldots$	$*$	b
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Smart Tables

Introduction of elementary constraints in tables (Mairy *et al.*, 2015)

The constraint $x = y \vee y = z$ can be defined by:

x	y	z
$= y$	$*$	$*$
$*$	$= z$	$*$

The global constraint **element(**$I, \langle x_1, x_2, \ldots, x_m \rangle, R$**)** can be defined by:

I	x_1	x_2	$\ldots$	x_m	R
1	$*$	$*$	$\ldots$	$*$	$= x_1$
2	$*$	$*$	$\ldots$	$*$	$= x_2$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
m	$*$	$*$	$\ldots$	$*$	$= x_m$

Smart Tables

Introduction of elementary constraints in tables (Mairy *et al.*, 2015)

The constraint $x = y \vee y = z$ can be defined by:

x	y	z
$= y$	$*$	$*$
$*$	$= z$	$*$

The global constraint **element(**$I, \langle x_1, x_2, \ldots, x_m \rangle, R$**)** can be defined by:

I	x_1	x_2	$\ldots$	x_m	R
1	$*$	$*$	$\ldots$	$*$	$= x_1$
2	$*$	$*$	$\ldots$	$*$	$= x_2$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
m	$*$	$*$	$\ldots$	$*$	$= x_m$

Smart Tables

Introduction of elementary constraints in tables (Mairy *et al.*, 2015)

The constraint $x = y \vee y = z$ can be defined by:

x	y	z
$= y$	$*$	$*$
$*$	$= z$	$*$

The global constraint **element(**$I, \langle x_1, x_2, \ldots, x_m \rangle, R$**)** can be defined by:

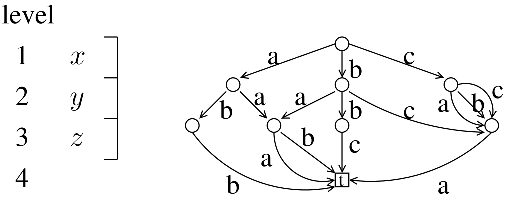
I	x_1	x_2	$\ldots$	x_m	R
1	$*$	$*$	$\ldots$	$*$	$= x_1$
2	$*$	$*$	$\ldots$	$*$	$= x_2$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
m	$*$	$*$	$\ldots$	$*$	$= x_m$

Tables vs MDDs

Multi-valued Decision Diagrams allow us to share prefixes and suffixes.

$\langle x, y, z \rangle \in T$

T
a a a
a a b
a b b
b a a
b a b
b b c
b c a
c a a



Bin Packing

We are given:

- a pool of similar bins (with a specified capacity)
- a set of items, each of them with a specified weight

The problem is:

- to put all items in the available bins
- while minimizing the number of necessary bins

Data for BinPacking

Data are stored in a JSON file:

```
{
  "nBins":40,
  "binCapacity":100,
  "itemWeights":[30,31,31,32,34,35,35,40,40,40,41,41,...]
}
```

The PyCSP^3 model given in the next slide requires two auxiliary functions (not shown here):

- `max_items_per_bin()`
- `occ_of_weights()`

Remark.

The operator + defined on dictionaries is a PyCSP^3 extension.

Model for BinPacking

```
from pycsp3 import *

nBins, capacity, weights = data
nItems = len(weights)
maxPerBin = max_items_per_bin()

# x[i][j] is the weight of the jth object put in the ith bin.
x = VarArray(size=[nBins, maxPerBin], dom={0, *weights})

satisfy(
  # not exceeding the capacity of each bin
  [Sum(x[i]) <= capacity for i in range(nBins)],

  # items are stored decreasingly according to their weights
  [Decreasing(x[i]) for i in range(nBins)],

  # ensuring that each item is stored in a bin
  Cardinality(x, occurrences={0: nBins * maxPerBin - nItems}
                  + {wgt: occ for (wgt, occ) in occ_of_weights()})
)

maximize(
    # maximizing the number of unused bins
    Sum(x[i][0] == 0 for i in range(nBins))
)
```

Using Tables or MDDs

Can a pair of constraints defined on similar scopes (from a given *i*):

```
Sum(x[i]) <= capacity
Decreasing(x[i])
```

be translated into :

- a table constraint
- or an MDD constraint

?

Answer: Yes

Example.

Instance `BinPacking-sw100-00`

- 18 tables with $2,747,755$ tuples
- 18 MDDs with $1,554$ nodes

By the way, what is the interest?

Using Tables or MDDs

Can a pair of constraints defined on similar scopes (from a given *i*):

```
Sum(x[i]) <= capacity
Decreasing(x[i])
```

be translated into :

- a table constraint
- or an MDD constraint

?

Answer: Yes

Example.

Instance `BinPacking-sw100-00`

- 18 tables with $2,747,755$ tuples
- 18 MDDs with $1,554$ nodes

By the way, what is the interest?

Using Tables or MDDs

Can a pair of constraints defined on similar scopes (from a given i):

```
Sum(x[i]) <= capacity
Decreasing(x[i])
```

be translated into :

- a table constraint
- or an MDD constraint

?

Answer: Yes

Example.

Instance `BinPacking-sw100-00`

- 18 tables with $2,747,755$ tuples
- 18 MDDs with $1,554$ nodes

By the way, what is the interest?

Outline

1. Tables and MDDs
2. Specific Algorithms for Table Constraints
3. Compact Table
4. Local Consistencies

Recall: Table Constraints

Often, a constraint `extension` is called a table constraint, especially when it is non-binary.

A table constraint is then simply a constraint defined in extension. And is is said to be:

- positive if allowed tuples are given
- negative if forbidden tuples are given

Remark.
We turn to specific algorithms for efficiency reasons.

Recall: Table Constraints

Often, a constraint `extension` is called a table constraint, especially when it is non-binary.

A table constraint is then simply a constraint defined in extension. And is is said to be:

- positive if allowed tuples are given
- negative if forbidden tuples are given

Remark.

We turn to specific algorithms for efficiency reasons.

Recall: Table Constraints

Often, a constraint `extension` is called a table constraint, especially when it is non-binary.

A table constraint is then simply a constraint defined in extension. And is is said to be:

- positive if allowed tuples are given
- negative if forbidden tuples are given

Remark.

We turn to specific algorithms for efficiency reasons.

Algorithms for Table Constraints

Many schemes/algorithms proposed in the literature:

- AC-valid (Bessiere & Régin, 1997)
- AC-allowed (Bessiere & Régin, 1997)
- AC-valid+allowed (Lecoutre & Szymanek, 2006)
- NextIn Indexing (Lhomme & Régin, 2005)
- NextDiff Indexing (Gent *et al.*, 2007)
- Tries (Gent *et al.*, 2007)
- Compressed Tables (Katsirelos & Walsh, 2007)
- MDDs (Cheng & Yap, 2010)
- STR1 (Ullmann, 2007)
- STR2 (Lecoutre, 2008)
- STR3 (Lecoutre *et al.*, 2012)
- AC5-TCOpt (Mairy *et al.*, 2012)
- AC4R and MDD4R (Perez & Régin, 2014)
- CT (Demeulenaere *et al.*, 2016)

Classical Schemes

Basic Schemes:

- AC-allowed: iterating over the list of allowed tuples
- AC-valid: iterating over the list of valid tuples
- AC-valid+allowed: visiting both lists

There exist r-ary positive table constraints such that, for some current domains of variables,

- applying AC-allowed is $O(2^{r-1})$.
- applying AC-valid is $O(2^{r-1})$.
- applying AC-valid+allowed is $O(r^2)$

Classical Schemes

Basic Schemes:

- AC-allowed: iterating over the list of allowed tuples
- AC-valid: iterating over the list of valid tuples
- AC-valid+allowed: visiting both lists

There exist r-ary positive table constraints such that, for some current domains of variables,

- applying AC-allowed is $O(2^{r-1})$.
- applying AC-valid is $O(2^{r-1})$.
- applying AC-valid+allowed is $O(r^2)$

Simple Tabular Reduction (STR)

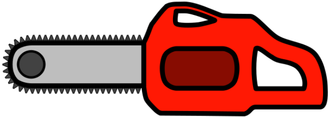
The previous schemes proceed **gradually**: a support is sought for each value in turn: (x, a), (x, b), (x, c), $\ldots$

Other (more recent) schemes proceed **globally**: AC is enforced by traversing (once) the structure of the constraint. For example :

- STR
- MDDc

Constraint filtering/propagation aims at pruning the search space. STR (Simple Tabular Reduction) prunes both:

- the tables
- and the domains



Simple Tabular Reduction (STR)

The previous schemes proceed **gradually**: a support is sought for each value in turn: (x, a), (x, b), (x, c), ...

Other (more recent) schemes proceed **globally**: AC is enforced by traversing (once) the structure of the constraint. For example :

- STR
- MDDc

Constraint filtering/propagation aims at pruning the search space. STR (Simple Tabular Reduction) prunes both:

- the tables
- and the domains

Simple Tabular Reduction

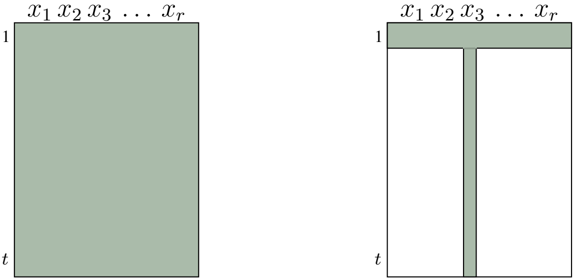
Simple Tabular Reduction (STR)

- principle: dynamically maintaining tables (only keeping supports)
- efficiency obtained by using a sparse set data structure

Versions of STR:

- STR(1) (Ullmann, 2007)
- STR2 (Lecoutre, 2008)
- STR3 (Lecoutre *et al.*, 2012)

Complexity:



(a) STR1 (b) STR2

Data Structures

For each constraint c, we just need a few structures:

- $table[c]$ the current table containing the current supports of c. It can be advantageously implemented by a sparse set (shown later).
- for each variable x, $gacValues[x]$ is the set containing the values in the domain of x that are (generalized) arc-consistent on c.

Algorithm 1: STR(c: Constraint)

```
foreach variable $x \in scp(c)$ do
    $gacValues[x] \leftarrow \emptyset$
foreach tuple $\tau \in table[c]$ do
    if $isValid(c, \tau)$ then
        foreach variable $x \in scp(c)$ do
            if $\tau[x] \notin gacValues[x]$ then
                add $\tau[x]$ to $gacValues[x]$
    else
        $removeTuple(c, \tau)$
// domains are now updated
foreach variable $x \in scp(c)$ do
    $dom(x) \leftarrow$ gacValues[$x$]
```

Illustration of STR

$table[c_{xyz}]$

$x\ y\ z$

(a,a,c)
(a,b,a)
(a,c,b)
(b,a,a)
(b,b,c)
(c,a,b)
(c,c,c)

Illustration of STR

$table[c_{xyz}]$

$x\ y\ z$

(a,a,c)
(a,b,a)
(a,c,b)
(b,a,a)
(b,b,c)
(c,a,b)
(c,c,c)

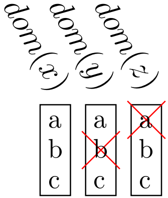
$dom(x)$	$dom(y)$	$dom(z)$
a	a	a
b	b	b
c	c	c

Illustration of STR

$table[c_{xyz}]$

$x\ y\ z$

(a,a,c)
(a,b,a)
(a,c,b)
(b,a,a)
(b,b,c)
(c,a,b)
(c,c,c)



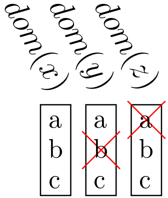
$$gacValues[x] = \{\}$$
$$gacValues[y] = \{\}$$
$$gacValues[z] = \{\}$$

Illustration of STR

$table[c_{xyz}]$

$x \; y \; z$

(a,a,c) ✓
(a,b,a)
(a,c,b)
(b,a,a)
(b,b,c)
(c,a,b)
(c,c,c)



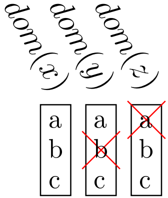
$$gacValues[x] = \{a\}$$
$$gacValues[y] = \{a\}$$
$$gacValues[z] = \{c\}$$

Illustration of STR

$table[c_{xyz}]$

$x \; y \; z$

(a,a,c) ✓
~~(a,b,a)~~
(a,c,b)
(b,a,a)
(b,b,c)
(c,a,b)
(c,c,c)



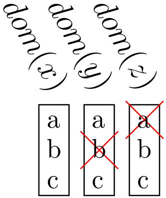
$$gacValues[x] = \{a\}$$
$$gacValues[y] = \{a\}$$
$$gacValues[z] = \{c\}$$

Illustration of STR

$table[c_{xyz}]$

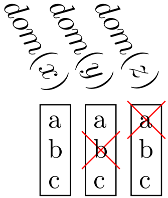
$x \ y \ z$

(a,a,c) ✓
~~(a,b,a)~~
(a,c,b) ✓
(b,a,a)
(b,b,c)
(c,a,b)
(c,c,c)



$$gacValues[x] = \{a\}$$
$$gacValues[y] = \{a, c\}$$
$$gacValues[z] = \{b, c\}$$

Illustration of STR



$table[c_{xyz}]$

$x \; y \; z$

(a,a,c) ✓
~~(a,b,a)~~
(a,c,b) ✓
~~(b,a,a)~~
(b,b,c)
(c,a,b)
(c,c,c)

$$gacValues[x] = \{a\}$$
$$gacValues[y] = \{a, c\}$$
$$gacValues[z] = \{b, c\}$$

Illustration of STR

$table[c_{xyz}]$

$x \; y \; z$

(a,a,c) ✓
~~(a,b,a)~~
(a,c,b) ✓
~~(b,a,a)~~
~~(b,b,c)~~
(c,a,b)
(c,c,c)

$dom(x)$	$dom(y)$	$dom(z)$
a	a	~~a~~
b	~~b~~	b
c	c	c

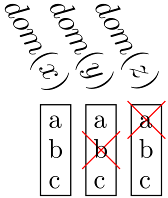
$$gacValues[x] = \{a\}$$
$$gacValues[y] = \{a, c\}$$
$$gacValues[z] = \{b, c\}$$

Illustration of STR

$table[c_{xyz}]$

$x\ y\ z$

(a,a,c) ✓
~~(a,b,a)~~
(a,c,b) ✓
~~(b,a,a)~~
~~(b,b,c)~~
(c,a,b) ✓
(c,c,c)



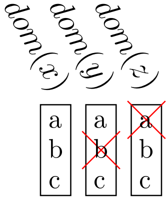
$$gacValues[x] = \{a, c\}$$
$$gacValues[y] = \{a, c\}$$
$$gacValues[z] = \{b, c\}$$

Illustration of STR

$table[c_{xyz}]$

$x \; y \; z$

(a,a,c) ✓
~~(a,b,a)~~
(a,c,b) ✓
~~(b,a,a)~~
~~(b,b,c)~~
(c,a,b) ✓
(c,c,c) ✓



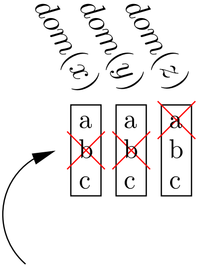
$$gacValues[x] = \{a, c\}$$
$$gacValues[y] = \{a, c\}$$
$$gacValues[z] = \{b, c\}$$

Illustration of STR

$table[c_{xyz}]$

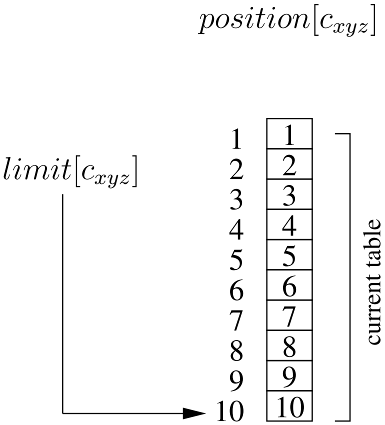
$x \ y \ z$

(a,a,c) ✓
~~(a,b,a)~~
(a,c,b) ✓
~~(b,a,a)~~
~~(b,b,c)~~
(c,a,b) ✓
(c,c,c) ✓



$$gacValues[x] = \{a, c\}$$
$$gacValues[y] = \{a, c\}$$
$$gacValues[z] = \{b, c\}$$

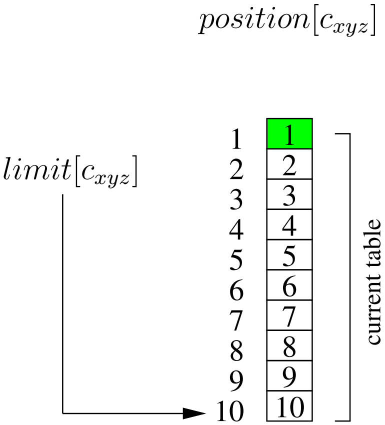
Updating after (y, b) being removed



$table[c_{xyz}]$

	$x\ y\ z$
1	(a,a,a)
2	(a,a,b)
3	(a,b,b)
4	(b,a,a)
5	(b,a,b)
6	(b,b,c)
7	(b,c,a)
8	(c,a,a)
9	(c,b,a)
10	(c,c,a)

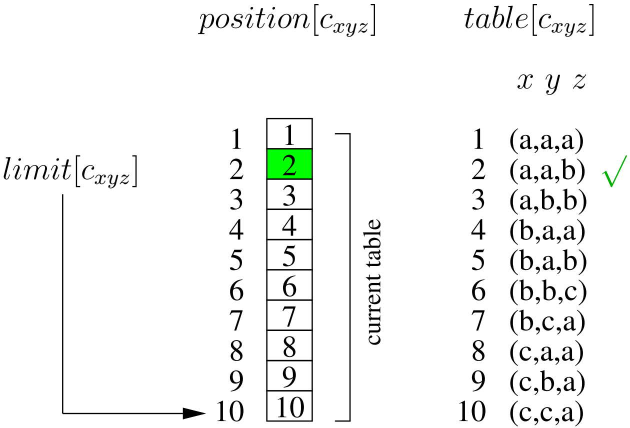
Updating after (y, b) being removed



$table[c_{xyz}]$

	$x\ y\ z$	
1	(a,a,a)	✓
2	(a,a,b)	
3	(a,b,b)	
4	(b,a,a)	
5	(b,a,b)	
6	(b,b,c)	
7	(b,c,a)	
8	(c,a,a)	
9	(c,b,a)	
10	(c,c,a)	

Updating after (y, b) being removed

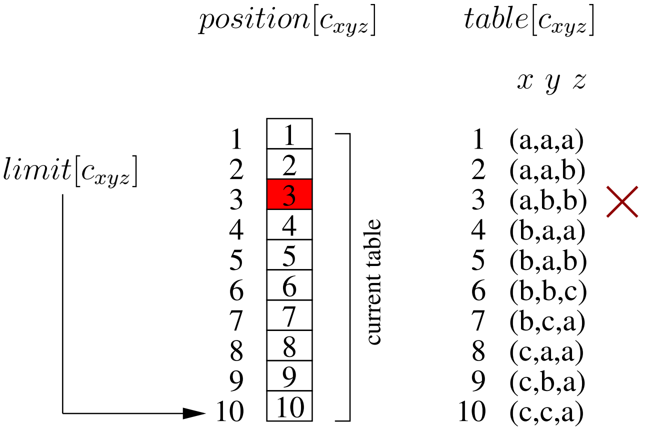


$limit[c_{xyz}]$

current table

	$position[c_{xyz}]$		$table[c_{xyz}]$ $x\ y\ z$	
1	1	1	(a,a,a)	
2	🟩 2	2	(a,a,b)	✓
3	3	3	(a,b,b)	
4	4	4	(b,a,a)	
5	5	5	(b,a,b)	
6	6	6	(b,b,c)	
7	7	7	(b,c,a)	
8	8	8	(c,a,a)	
9	9	9	(c,b,a)	
10	10	10	(c,c,a)	

Updating after (y, b) being removed

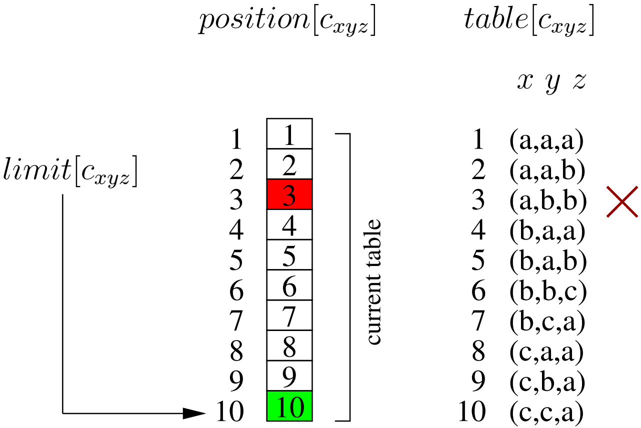


$limit[c_{xyz}]$

current table

	$position[c_{xyz}]$		$table[c_{xyz}]$ $x\ y\ z$
1	1	1	(a,a,a)
2	2	2	(a,a,b)
3	3	3	(a,b,b)
4	4	4	(b,a,a)
5	5	5	(b,a,b)
6	6	6	(b,b,c)
7	7	7	(b,c,a)
8	8	8	(c,a,a)
9	9	9	(c,b,a)
10	10	10	(c,c,a)

Updating after (y, b) being removed

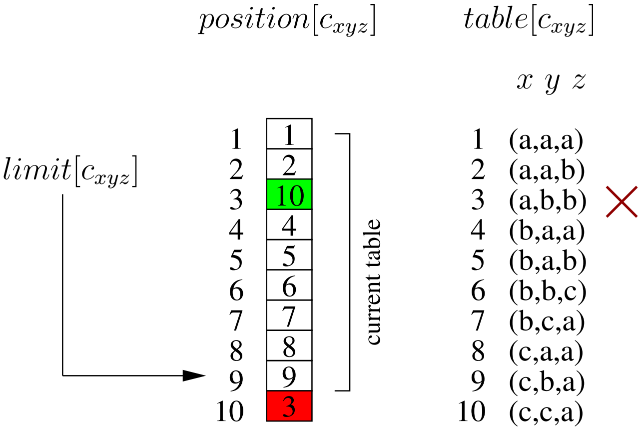


$limit[c_{xyz}]$ → 10

current table

	$position[c_{xyz}]$	$table[c_{xyz}]$ $x\ y\ z$	
1	1	(a,a,a)	
2	2	(a,a,b)	
3	3	(a,b,b)	×
4	4	(b,a,a)	
5	5	(b,a,b)	
6	6	(b,b,c)	
7	7	(b,c,a)	
8	8	(c,a,a)	
9	9	(c,b,a)	
10	10	(c,c,a)	

Updating after (y, b) being removed



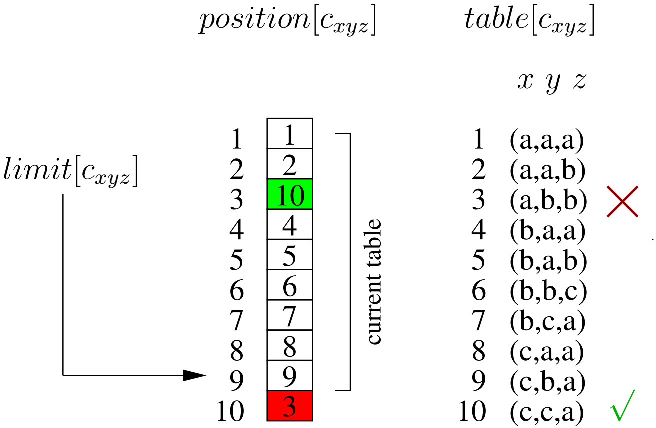
$limit[c_{xyz}]$

current table

	$position[c_{xyz}]$
1	1
2	2
3	10
4	4
5	5
6	6
7	7
8	8
9	9
10	3

	$table[c_{xyz}]$ $x\ y\ z$	
1	(a,a,a)	
2	(a,a,b)	
3	(a,b,b)	✗
4	(b,a,a)	
5	(b,a,b)	
6	(b,b,c)	
7	(b,c,a)	
8	(c,a,a)	
9	(c,b,a)	
10	(c,c,a)	

Updating after (y, b) being removed



$limit[c_{xyz}]$

$position[c_{xyz}]$

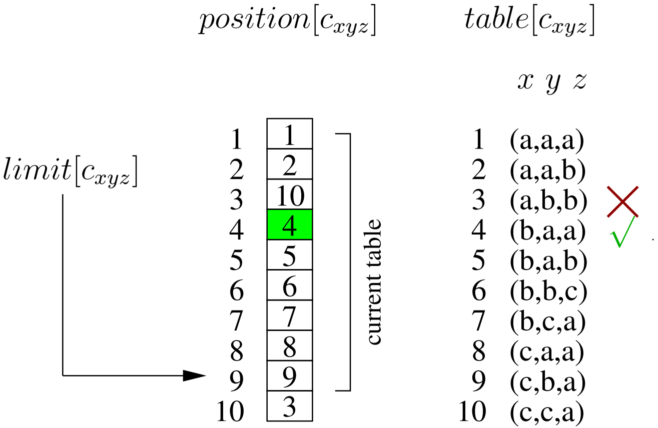
1	1
2	2
3	🟩 10
4	4
5	5
6	6
7	7
8	8
9	9
10	🟥 3

current table

$table[c_{xyz}]$

	$x\ y\ z$	
1	(a,a,a)	
2	(a,a,b)	
3	(a,b,b)	✗
4	(b,a,a)	
5	(b,a,b)	
6	(b,b,c)	
7	(b,c,a)	
8	(c,a,a)	
9	(c,b,a)	
10	(c,c,a)	✓

Updating after (y, b) being removed



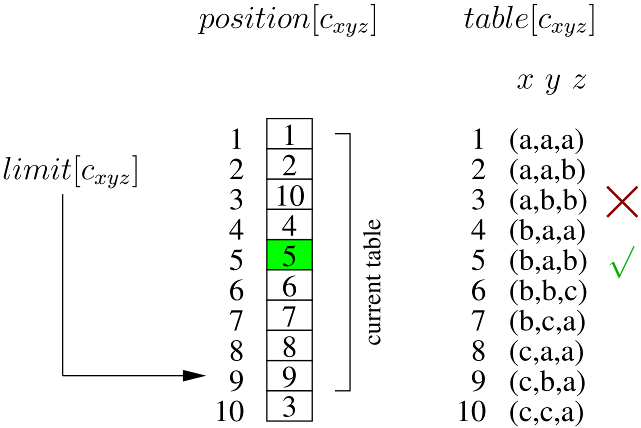
$limit[c_{xyz}]$

current table

	$position[c_{xyz}]$
1	1
2	2
3	10
4	🟩 4
5	5
6	6
7	7
8	8
9	9
10	3

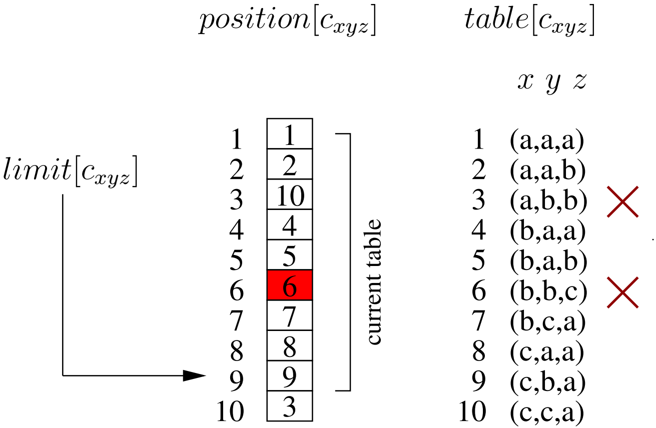
	$table[c_{xyz}]$ x y z	
1	(a,a,a)	
2	(a,a,b)	
3	(a,b,b)	✗
4	(b,a,a)	✓
5	(b,a,b)	
6	(b,b,c)	
7	(b,c,a)	
8	(c,a,a)	
9	(c,b,a)	
10	(c,c,a)	

Updating after (y, b) being removed



	$position[c_{xyz}]$		$table[c_{xyz}]$	
			x y z	
1	1	1	(a,a,a)	
2	2	2	(a,a,b)	
3	10	3	(a,b,b)	✗
4	4	4	(b,a,a)	
5	5	5	(b,a,b)	✓
6	6	6	(b,b,c)	
7	7	7	(b,c,a)	
8	8	8	(c,a,a)	
9	9	9	(c,b,a)	
10	3	10	(c,c,a)	

Updating after (y, b) being removed

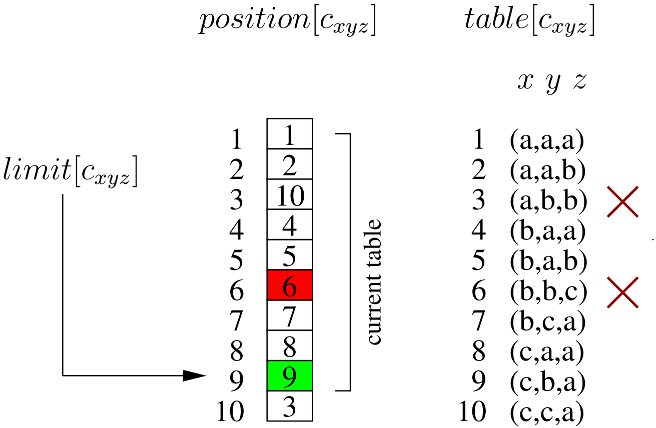


	$position[c_{xyz}]$		$table[c_{xyz}]$ $x\ y\ z$	
1	1	1	(a,a,a)	
2	2	2	(a,a,b)	
3	10	3	(a,b,b)	✗
4	4	4	(b,a,a)	
5	5	5	(b,a,b)	
6	6	6	(b,b,c)	✗
7	7	7	(b,c,a)	
8	8	8	(c,a,a)	
9	9	9	(c,b,a)	
10	3	10	(c,c,a)	

$limit[c_{xyz}]$

current table

Updating after (y, b) being removed

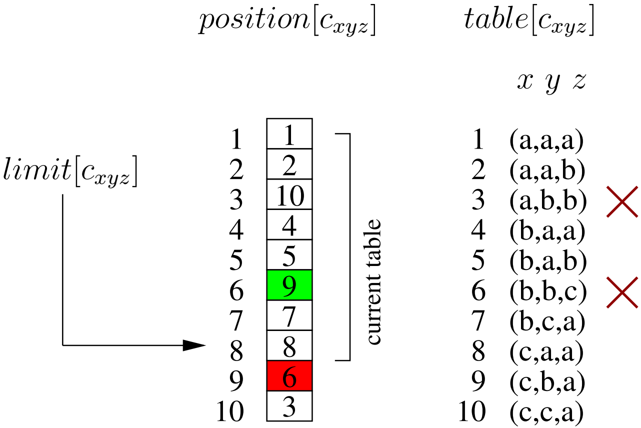


$limit[c_{xyz}]$

current table

	$position[c_{xyz}]$		$table[c_{xyz}]$ $x\ y\ z$	
1	1	1	(a,a,a)	
2	2	2	(a,a,b)	
3	10	3	(a,b,b)	✗
4	4	4	(b,a,a)	
5	5	5	(b,a,b)	
6	6	6	(b,b,c)	✗
7	7	7	(b,c,a)	
8	8	8	(c,a,a)	
9	9	9	(c,b,a)	
10	3	10	(c,c,a)	

Updating after (y, b) being removed



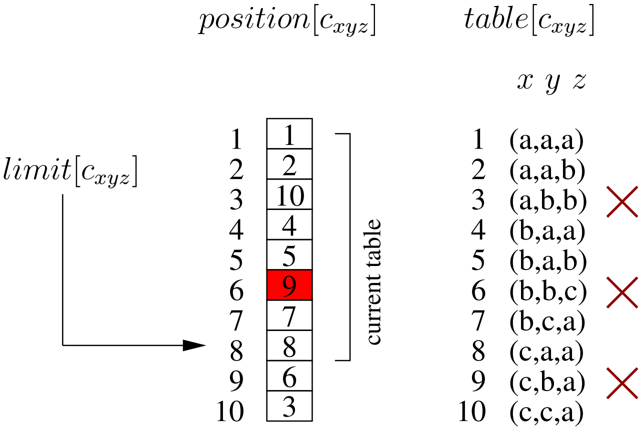
$limit[c_{xyz}]$ → 8

current table: 1–8

	$position[c_{xyz}]$
1	1
2	2
3	10
4	4
5	5
6	🟩 9
7	7
8	8
9	🟥 6
10	3

	$table[c_{xyz}]$ $x\ y\ z$	
1	(a,a,a)	
2	(a,a,b)	
3	(a,b,b)	✗
4	(b,a,a)	
5	(b,a,b)	
6	(b,b,c)	✗
7	(b,c,a)	
8	(c,a,a)	
9	(c,b,a)	
10	(c,c,a)	

Updating after (y, b) being removed



$limit[c_{xyz}]$ → 8

current table: 1–8

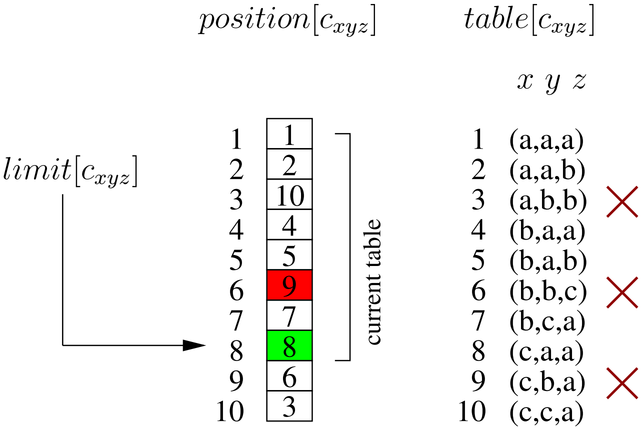
$position[c_{xyz}]$

1	1
2	2
3	10
4	4
5	5
6	9
7	7
8	8
9	6
10	3

$table[c_{xyz}]$

	$x\ y\ z$	
1	(a,a,a)	
2	(a,a,b)	
3	(a,b,b)	✗
4	(b,a,a)	
5	(b,a,b)	
6	(b,b,c)	✗
7	(b,c,a)	
8	(c,a,a)	
9	(c,b,a)	✗
10	(c,c,a)	

Updating after (y, b) being removed



$position[c_{xyz}]$

1	1
2	2
3	10
4	4
5	5
6	9
7	7
8	8
9	6
10	3

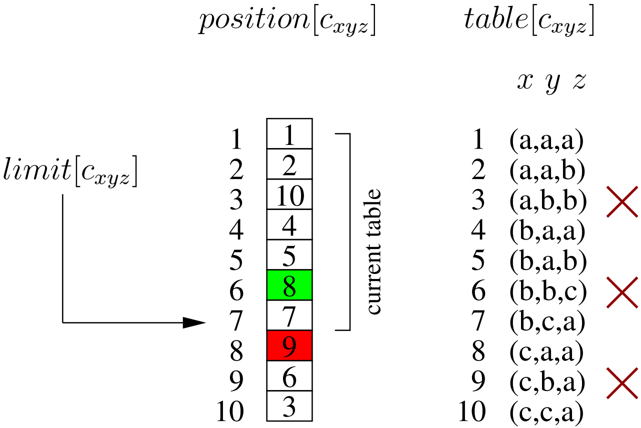
$limit[c_{xyz}]$

current table

$table[c_{xyz}]$

	$x\ y\ z$	
1	(a,a,a)	
2	(a,a,b)	
3	(a,b,b)	✗
4	(b,a,a)	
5	(b,a,b)	
6	(b,b,c)	✗
7	(b,c,a)	
8	(c,a,a)	
9	(c,b,a)	✗
10	(c,c,a)	

Updating after (y, b) being removed

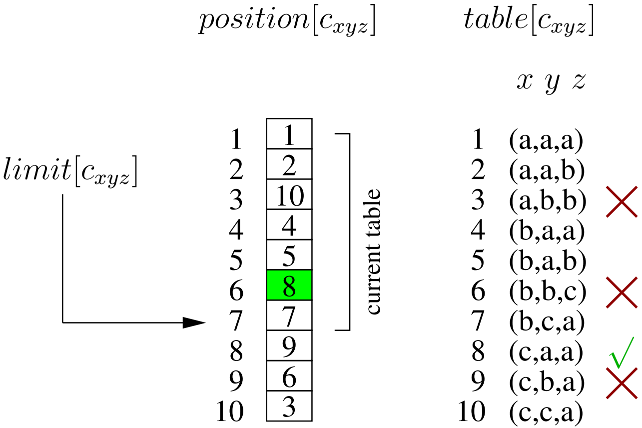


$limit[c_{xyz}]$ → 7

current table: 1–7

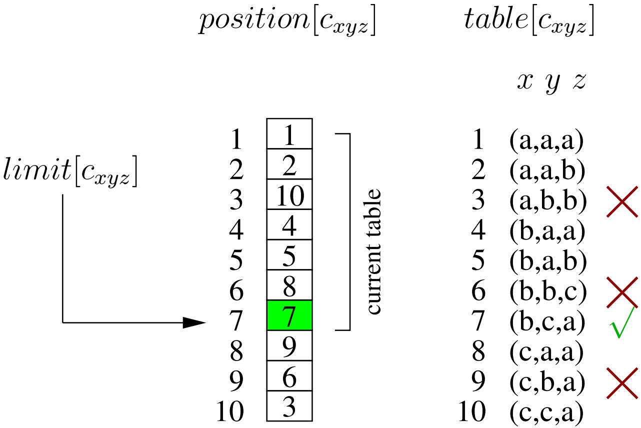
	$position[c_{xyz}]$	$table[c_{xyz}]$ $x\ y\ z$	
1	1	(a,a,a)	
2	2	(a,a,b)	
3	10	(a,b,b)	✗
4	4	(b,a,a)	
5	5	(b,a,b)	
6	8	(b,b,c)	✗
7	7	(b,c,a)	
8	9	(c,a,a)	
9	6	(c,b,a)	✗
10	3	(c,c,a)	

Updating after (y, b) being removed



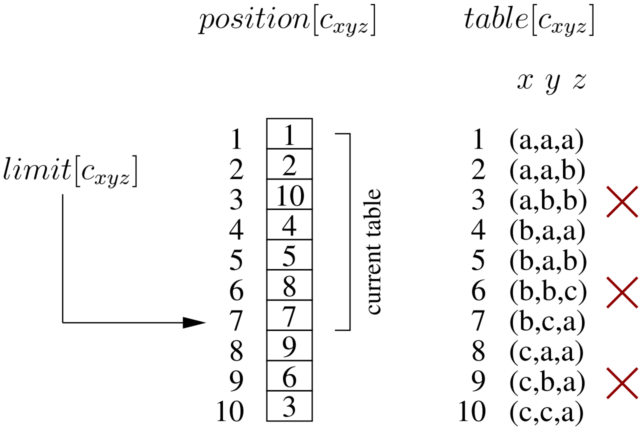
	$position[c_{xyz}]$		$table[c_{xyz}]$ x y z	
1	1	1	(a,a,a)	
2	2	2	(a,a,b)	
3	10	3	(a,b,b)	✗
4	4	4	(b,a,a)	
5	5	5	(b,a,b)	
6	8	6	(b,b,c)	✗
7	7	7	(b,c,a)	
8	9	8	(c,a,a)	✓
9	6	9	(c,b,a)	✗
10	3	10	(c,c,a)	

Updating after (y, b) being removed



	$position[c_{xyz}]$	$table[c_{xyz}]$ $x\ y\ z$	
1	1	(a,a,a)	
2	2	(a,a,b)	
3	10	(a,b,b)	✗
4	4	(b,a,a)	
5	5	(b,a,b)	
6	8	(b,b,c)	✗
7	🟩 7	(b,c,a)	✓
8	9	(c,a,a)	
9	6	(c,b,a)	✗
10	3	(c,c,a)	

Updating after (y, b) being removed



$position[c_{xyz}]$

$limit[c_{xyz}]$

current table

	$position[c_{xyz}]$
1	1
2	2
3	10
4	4
5	5
6	8
7	7
8	9
9	6
10	3

$table[c_{xyz}]$

	$x\ y\ z$	
1	(a,a,a)	
2	(a,a,b)	
3	(a,b,b)	✗
4	(b,a,a)	
5	(b,a,b)	
6	(b,b,c)	✗
7	(b,c,a)	
8	(c,a,a)	
9	(c,b,a)	✗
10	(c,c,a)	

Outline

1. Tables and MDDs

2. Specific Algorithms for Table Constraints

3. **Compact Table**

4. Local Consistencies

Successful Techniques for Table Constraints

Over the last decade, many developments for enforcing AC on extensional constraints. Among successful techniques, we find:

- bitwise operations that allow performing parallel operations on bit vectors,
- residual supports (residues) that store the last found supports of each value,
- tabular reduction, which is a technique that dynamically maintains the tables of supports,
- resetting operations that saves substantial computing efforts in some particular situations.

Reversible Sparse Bit-sets

For example, for a set initially containing 82 elements, we build an array with $p = \lceil 82/64 \rceil = 2$ words:

```
words: 11111111111111111111111111111111 11111111111111111111111111111111
       11111111111111111100000000000000 00000000000000000000000000000000
index: 0 1
limit : 1
```

If we suppose that the 66 first elements are removed, we obtain:

```
words: 00000000000000000000000000000000 00000000000000000000000000000000
       00111111111111111100000000000000 00000000000000000000000000000000
index: 1 0
limit: 0
```

The class invariant for reversible sparse bit-sets is:

- `index` is a permutation of $[0, \ldots, p-1]$, and
- $\texttt{words}[\texttt{index}[i]] \neq 0^{64} \Leftrightarrow i \leq \texttt{limit}, \forall i \in 0..p-1$

Reversible Sparse Bit-sets

Algorithm 2: Class RSparseBitSet

```
words: array of rlong                          // words.length = p
index: array of int                            // index.length = p
limit: rint
mask: array of long                            // mask.length = p
Method isEmpty(): Boolean
Method clearMask()
Method reverseMask()
Method addToMask(m: array of long)
    foreach $i$ from 0 to limit do
        pos ← index[$i$]
        mask[pos] ← mask[pos] | $m$[pos]       // bitwise OR
Method intersectWithMask()
```

Initialization of $\langle x, y, z \rangle \in T$

Consider $\langle x, y, z \rangle \in T$, where $dom(x) = \{a, b\}$, $dom(y) = \{a, b, d\}$, $dom(z) = \{a, b, c\}$. We build static arrays `supports`:

	T
0	a a a
1	a a b
2	a b c
3	b a a
	a c b
4	a b b
5	b a b
6	b b a
7	b b b

	0	1	2	3	4	5	6	7
`currTable`	1	1	1	1	1	1	1	1
`supports`[x, a]	1	1	1	0	1	0	0	0
`supports`[x, b]	0	0	0	1	0	1	1	1
`supports`[y, a]	1	1	0	1	0	1	0	0
`supports`[y, b]	0	0	1	0	1	0	1	1
`supports`[y, d]	0	0	0	0	0	0	0	0
`supports`[z, a]	1	0	0	1	0	0	1	0
`supports`[z, b]	0	1	0	0	1	1	0	1
`supports`[z, c]	0	0	1	0	0	0	0	0

- The tuple (a, c, b) is initially invalid because $c \notin dom(y)$, and thus will not be indexed.
- Value d will be removed from $dom(y)$ given that it is not supported by any tuple.

Algorithm CT (for enforcing AC)

① updating (reducing) the current table
② filtering variable domains

Example.

Hypothesis: $x \neq a$

1. updateTable() invalidates tuples supporting (x, a)

`currTable`^{in}	1	1	1	1	1	1	1	1
`supports`$[x, a]$	1	1	1	0	1	0	0	0
`currTable`^{out}	0	0	0	1	0	1	1	1

2. filterDomains() removes (z, c)

`currTable`	0	0	0	1	0	1	1	1
`supports`$[x, b] \cap$ `currTable`	0	0	0	1	0	1	1	1
`supports`$[y, a] \cap$ `currTable`	0	0	0	1	0	1	0	0
`supports`$[y, b] \cap$ `currTable`	0	0	0	0	0	0	1	1
`supports`$[z, a] \cap$ `currTable`	0	0	0	1	0	0	1	0
`supports`$[z, b] \cap$ `currTable`	0	0	0	0	0	1	0	1
`supports`$[z, c] \cap$ `currTable`	0	0	0	0	0	0	0	0

Methods `enforceAC()` and `updateTable()`

```
Method enforceAC()
    updateTable()
    if currTable.isEmpty() then
        return Backtrack
    filterDomains()
```

```
Method updateTable()
    foreach variable $x \in scp(c)$ do
        if $|\Delta_x| \neq \emptyset$ then
            currTable.clearMask()
            foreach value $a \in \Delta_x$ do
                currTable.addToMask(supports[$x, a$])
            currTable.reverseMask()
            currTable.intersectWithMask()
            if currTable.isEmpty() then
                break
```

Method filterDomains()

```
Method filterDomains()
    foreach variable $x \in scp(c)$ do
        foreach value $a \in dom(x)$ do
            index ← residues[$x, a$]
            if currTable.words[index] & supports($x, a$)[index] $= 0^{64}$ then
                index ← currTable.intersectIndex(supports[$x, a$])
                if index $\neq -1$ then
                    residues[$x, a$] ← index
                else
                    $dom(x) \leftarrow dom(x) \setminus \{a\}$
```

Performance

Speedup of CT compared to other algorithms.

Speedup	STR2	STR3	GAC4	GAC4R	MDD4R	AC5-TC	Best2
average	9.11	5.07	15.59	11.37	10.38	50.40	3.77
min	0.76	1.09	0.92	1.13	0.13	1.05	0.13
max	88.58	51.04	173.24	208.52	50.84	1850	15.99
std	10.64	4.36	19.67	18.57	9.46	134.13	2.87

Take-Away Message concerning Table Constraints

Efficient filtering algorithms for extensional constraints:

- AC3^{bit+rm} for binary constraints
- CT for non-binary constraints with large tables
- STR2 or STR3 for non-binary constraints with tables of moderate sizes
- MDD for constraints that can be highly compressed

Many developments still to do about:

- filtering negative table constraints with * and refutations
- extending the scope of smart constraints
- automatic generation of smart constraints
- automatic compilation of subsets of constraints into table/smart constraints

Outline

1. Tables and MDDs

2. Specific Algorithms for Table Constraints

3. Compact Table

4. Local Consistencies

Nogood

Definition

A nogood for a CN P is an instantiation of a subset of variables of P that cannot be extended to a solution.

Definition

A (local) consistency is a property defined on CNs. Typically, it reveals some nogoods.

Remark.

Recording nogoods identified by consistencies usually permits to improve the process of exploring the search space, especially when the nogoods are of size 1 (i.e., inconsistent values).

Nogood

Definition

A nogood for a CN P is an instantiation of a subset of variables of P that cannot be extended to a solution.

Definition

A (local) consistency is a property defined on CNs. Typically, it reveals some nogoods.

Remark.

Recording nogoods identified by consistencies usually permits to improve the process of exploring the search space, especially when the nogoods are of size 1 (i.e., inconsistent values).

Nogood

Definition

A nogood for a CN P is an instantiation of a subset of variables of P that cannot be extended to a solution.

Definition

A (local) consistency is a property defined on CNs. Typically, it reveals some nogoods.

Remark.

Recording nogoods identified by consistencies usually permits to improve the process of exploring the search space, especially when the nogoods are of size 1 (i.e., inconsistent values).

Nogood

Definition

A nogood for a CN P is an instantiation of a subset of variables of P that cannot be extended to a solution.

Definition

A (local) consistency is a property defined on CNs. Typically, it reveals some nogoods.

Remark.

Recording nogoods identified by consistencies usually permits to improve the process of exploring the search space, especially when the nogoods are of size 1 (i.e., inconsistent values).

Example.

- Nogood of size 1

  $$\{x = a\}$$

  meaning

  $$\neg(x = a)$$

- Nogood of size 2

  $$\{x = a, y = b\}$$

  meaning

  $$\neg(x = a \wedge y = b)$$

- Nogood of size 3

  $$\{x = a, y = b, z = v\}$$

  meaning

  $$\neg(x = a \wedge y = b \wedge z = c)$$

- $\ldots$

Consistency

A domain-filtering consistency allows us to identify inconsistent values (nogoods of size 1). For example:

- Arc Consistency (AC)
- Path Inverse Consistency (PIC)
- Singleton Arc Consistency (SAC)

Some consistencies allows us to identify inconsistent pairs of values (nogoods of size 2). For example:

- Path Consistency (PC)
- Dual Consistency (DC)
- Conservative variants of PC and DC

A relation-filtering consistency allows us to identify inconsistent tuples (nogoods of size r) in constraint relations. For example:

- Pairwise Consistency (PWC)
- k-wise Consistency

Consistency

A domain-filtering consistency allows us to identify inconsistent values (nogoods of size 1). For example:

- Arc Consistency (AC)
- Path Inverse Consistency (PIC)
- Singleton Arc Consistency (SAC)

Some consistencies allows us to identify inconsistent pairs of values (nogoods of size 2). For example:

- Path Consistency (PC)
- Dual Consistency (DC)
- Conservative variants of PC and DC

A relation-filtering consistency allows us to identify inconsistent tuples (nogoods of size r) in constraint relations. For example:

- Pairwise Consistency (PWC)
- k-wise Consistency

Consistency

A domain-filtering consistency allows us to identify inconsistent values (nogoods of size 1). For example:

- Arc Consistency (AC)
- Path Inverse Consistency (PIC)
- Singleton Arc Consistency (SAC)

Some consistencies allows us to identify inconsistent pairs of values (nogoods of size 2). For example:

- Path Consistency (PC)
- Dual Consistency (DC)
- Conservative variants of PC and DC

A relation-filtering consistency allows us to identify inconsistent tuples (nogoods of size r) in constraint relations. For example:

- Pairwise Consistency (PWC)
- k-wise Consistency

Domain-filtering Consistency

To define a domain-filtering consistency ϕ, it is sufficient to give the conditions under which a value (x, a) is considered as ϕ-inconsistent.

Then, we can adopt the following definitions:

Definition

Let ϕ be a domain-filtering consistency.

- A constraint c is ϕ-consistent iff any value of c is ϕ-consistent, i.e. $\forall x \in scp(c), \forall a \in dom(x)$, (x, a) is ϕ-consistent.
- A constraint network P is ϕ-consistent iff any value of P is ϕ-consistent, i.e. $\forall x \in vars(P), \forall a \in dom(x)$, (x, a) is ϕ-consistent.

Domain-filtering Consistency

To define a domain-filtering consistency ϕ, it is sufficient to give the conditions under which a value (x, a) is considered as ϕ-inconsistent.

Then, we can adopt the following definitions:

Definition

Let ϕ be a domain-filtering consistency.

- A constraint c is ϕ-consistent iff any value of c is ϕ-consistent, i.e. $\forall x \in scp(c), \forall a \in dom(x)$, (x, a) is ϕ-consistent.
- A constraint network P is ϕ-consistent iff any value of P is ϕ-consistent, i.e. $\forall x \in vars(P), \forall a \in dom(x)$, (x, a) is ϕ-consistent.

Recall: AC

Definition (Arc Consistency)

- A value (x, a) of a constraint network P is AC iff for every constraint c of P involving x, there exists a support of (x, a) on c.

Hence, we can say that:

- A constraint c is AC iff $\forall x \in scp(c), \forall a \in dom(x)$, (x, a) is AC (equivalently, there exists a support of (x, a) on c).
- A constraint network P is AC iff every constraint of P is AC.

Remark.

Note how we usually simply write AC, instead of AC-consistent.

Recall: AC

Definition (Arc Consistency)

- A value (x, a) of a constraint network P is AC iff for every constraint c of P involving x, there exists a support of (x, a) on c.

Hence, we can say that:

- A constraint c is AC iff $\forall x \in scp(c), \forall a \in dom(x)$, (x, a) is AC (equivalently, there exists a support of (x, a) on c).
- A constraint network P is AC iff every constraint of P is AC.

Remark.

Note how we usually simply write AC, instead of AC-consistent.

SAC

Definition (Singleton Arc Consistency)

- A value (x, a) of a constraint network P is SAC iff $AC(P|_{x=a}) \neq \bot$.

Remark.
SAC is stronger than AC

Of course, it is possible to generalize the principle of checking one step in advance if a given local consistency holds as follows:

Definition (Singleton ϕ-consistency)

- A value (x, a) of a constraint network P is singleton ϕ-consistent, with ϕ being a consistency, iff $\phi(P|_{x=a}) \neq \bot$.

SAC

Definition (Singleton Arc Consistency)

- A value (x, a) of a constraint network P is SAC iff $AC(P|_{x=a}) \neq \bot$.

Remark.
SAC is stronger than AC

Of course, it is possible to generalize the principle of checking one step in advance if a given local consistency holds as follows:

Definition (Singleton ϕ-consistency)

- A value (x, a) of a constraint network P is singleton ϕ-consistent, with ϕ being a consistency, iff $\phi(P|_{x=a}) \neq \bot$.

SAC

Definition (Singleton Arc Consistency)

- A value (x, a) of a constraint network P is SAC iff $AC(P|_{x=a}) \neq \bot$.

Remark.

SAC is stronger than AC

Of course, it is possible to generalize the principle of checking one step in advance if a given local consistency holds as follows:

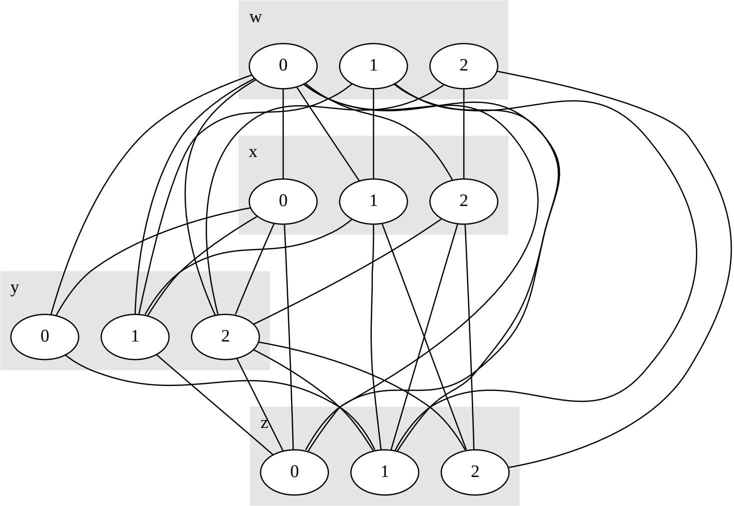
Definition (Singleton ϕ-consistency)

- A value (x, a) of a constraint network P is singleton ϕ-consistent, with ϕ being a consistency, iff $\phi(P|_{x=a}) \neq \bot$.

SAC Algorithms

Algorithm	Time	Space	Author(s)
SAC-1	$O(en^2d^4)$	$O(ed)$	(Debruyne & Bessiere, 1997)
SAC-2	$O(en^2d^4)$	$O(n^2d^2)$	(Bartak & Erben, 2004)
SAC-Opt	$O(end^3)$	$O(end^2)$	(Bessiere & Debruyne, 2004)
SAC-SDS	$O(end^4)$	$O(n^2d^2)$	(Bessiere & Debruyne, 2005)
SAC-3	$O(bed^2)$	$O(ed)$	(Lecoutre & Cardon, 2005)
SAC-3+	$O(bed^2)$	$O(b_{max}nd + ed)$	(Lecoutre & Cardon, 2005)

A binary CN to be made SAC



The singleton check for $(w, 0)$.

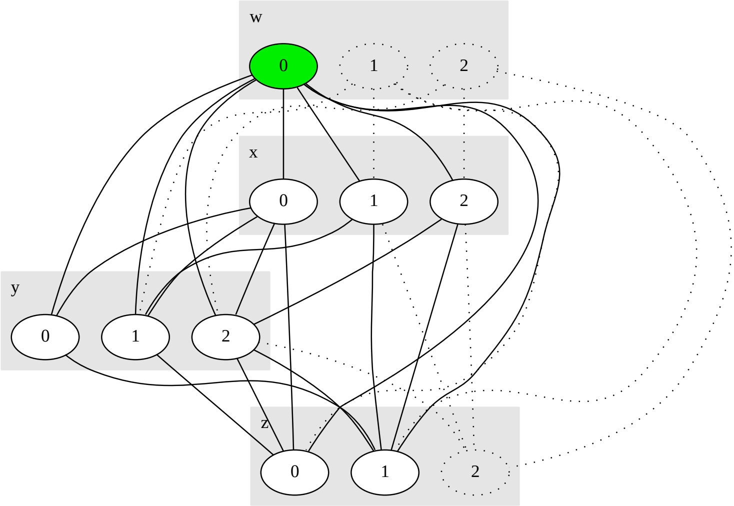


Figure: Singleton Check $AC(P|_{w=0})$

The singleton check for $(w, 1)$.

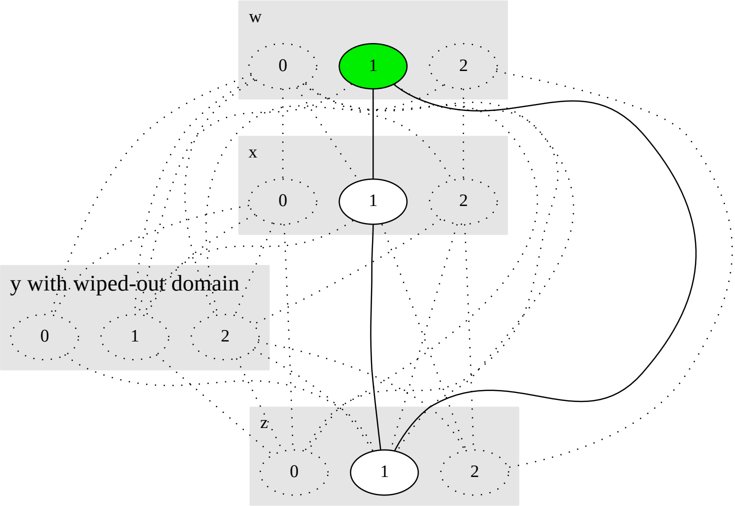


Figure: Singleton Check $AC(P|_{w=1})$

Using Algorithm SAC-1

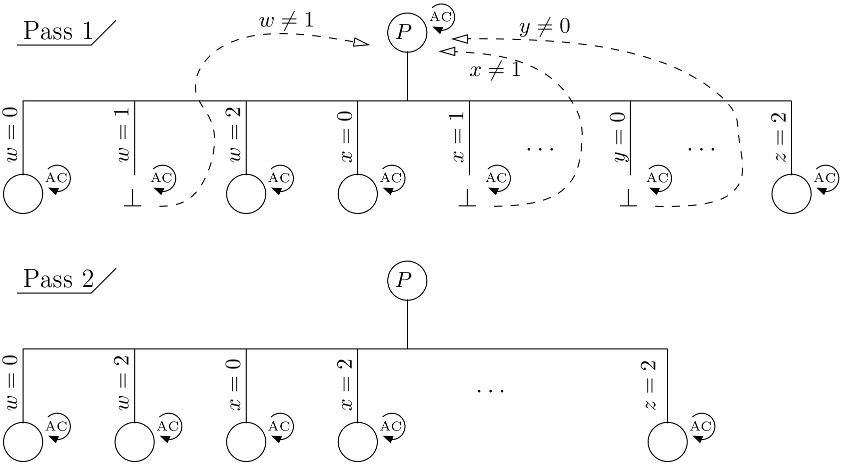
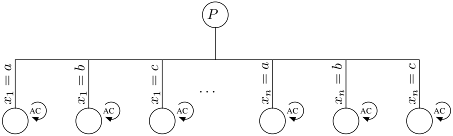


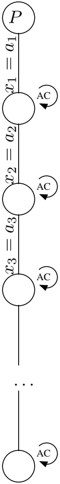
Figure: Singleton checks performed by SAC-1

Exploiting Incrementality of AC Algorithms



The complexity of enforcing AC on a node is $O(ed^2)$.

The complexity of enforcing AC on a branch is $O(ed^2)$.



Using Algorithm SAC-Opt and SAC-SDS

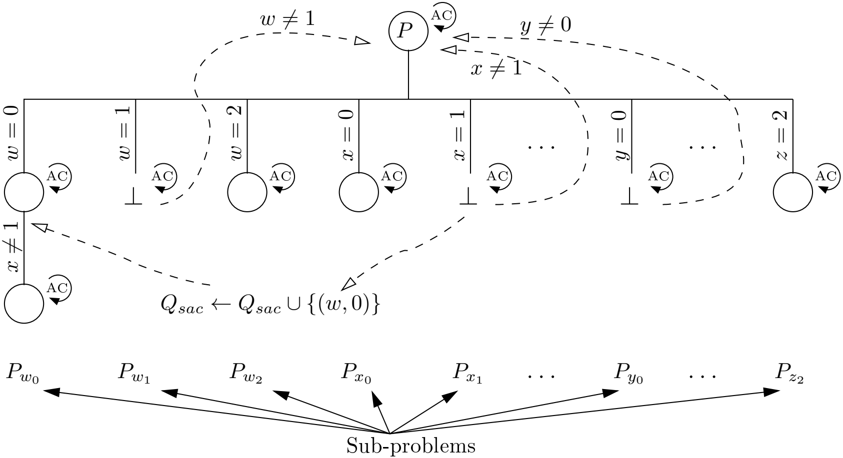


Figure: Singleton checks performed by SAC-Opt and SAC-SDS

Using Algorithm SAC-3+

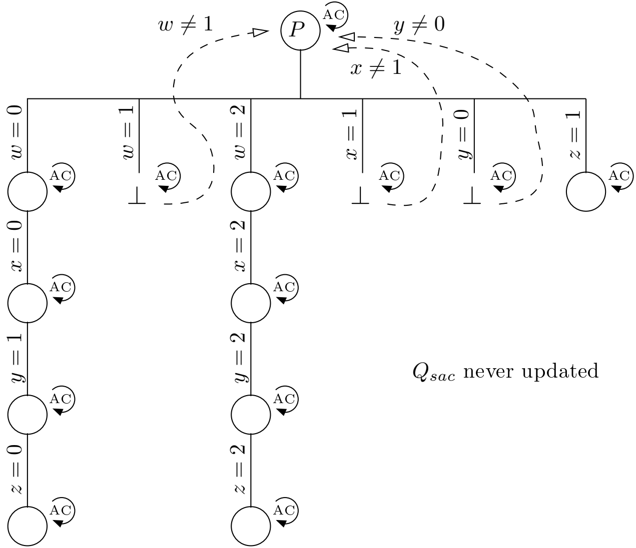


Figure: Branches built by SAC3+

PIC and MaxRPC

Definition

An instantiation I of a subset of variables of a CN P is locally consistent iff each constraint of P covered by I is satisfied by I.

Definition (Path Inverse Consistency)

- A value (x, a) of a constraint network P is PIC iff for any set $\{y, z\}$ of two variables of P, with $x \neq y$ and $x \neq z$, there exists $b \in dom(y)$ and $c \in dom(z)$ such that $\{(x, a), (y, b), (z, c)\}$ is locally consistent.

Definition (Max-restricted Path Consistency)

- A value (x, a) of a constraint network P is MaxRPC iff for any binary constraint c_{xy} of P involving x and another variable y, there exists a locally consistent instantiation $\{(x, a), (y, b)\}$ such that for any other variable z of P, there exists a value $c \in dom(z)$ guaranteeing that $\{(x, a), (z, c)\}$ and $\{(y, b), (z, c)\}$ are both locally consistent.

PIC and MaxRPC

Definition

An instantiation I of a subset of variables of a CN P is locally consistent iff each constraint of P covered by I is satisfied by I.

Definition (Path Inverse Consistency)

- A value (x, a) of a constraint network P is PIC iff for any set $\{y, z\}$ of two variables of P, with $x \neq y$ and $x \neq z$, there exists $b \in dom(y)$ and $c \in dom(z)$ such that $\{(x, a), (y, b), (z, c)\}$ is locally consistent.

Definition (Max-restricted Path Consistency)

- A value (x, a) of a constraint network P is MaxRPC iff for any binary constraint c_{xy} of P involving x and another variable y, there exists a locally consistent instantiation $\{(x, a), (y, b)\}$ such that for any other variable z of P, there exists a value $c \in dom(z)$ guaranteeing that $\{(x, a), (z, c)\}$ and $\{(y, b), (z, c)\}$ are both locally consistent.

PIC and MaxRPC

Definition

An instantiation I of a subset of variables of a CN P is locally consistent iff each constraint of P covered by I is satisfied by I.

Definition (Path Inverse Consistency)

- A value (x, a) of a constraint network P is PIC iff for any set $\{y, z\}$ of two variables of P, with $x \neq y$ and $x \neq z$, there exists $b \in dom(y)$ and $c \in dom(z)$ such that $\{(x, a), (y, b), (z, c)\}$ is locally consistent.

Definition (Max-restricted Path Consistency)

- A value (x, a) of a constraint network P is MaxRPC iff for any binary constraint c_{xy} of P involving x and another variable y, there exists a locally consistent instantiation $\{(x, a), (y, b)\}$ such that for any other variable z of P, there exists a value $c \in dom(z)$ guaranteeing that $\{(x, a), (z, c)\}$ and $\{(y, b), (z, c)\}$ are both locally consistent.

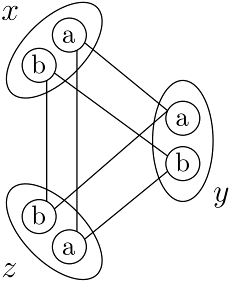


Figure: A constraint network with three binary constraints. Each value is arc-consistent but no one is path inverse consistent.

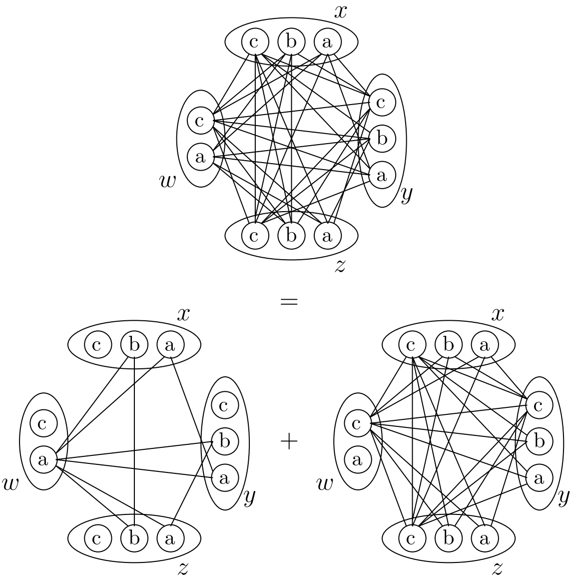


Figure: A constraint network with six binary constraints. Each value is path inverse consistent but (x, a) is not max restricted path consistent.

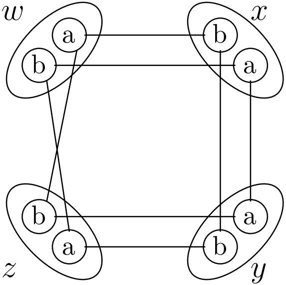
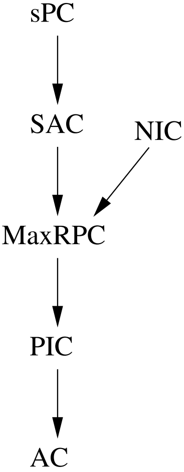


Figure: A constraint network with four binary constraints. Each value is max restricted path consistent but no one is singleton arc consistent.

Relationships between Domain-filtering Consistencies



$\phi \longrightarrow \psi$

means

ϕ is strictly stronger than ψ

Path Consistency

Definition

- A constraint network P is PC iff for every locally consistent instantiation $\{(x,a),(y,b)\}$ on P, there exists a value c in the domain of any third variable z of P such that $\{(x,a),(z,c)\}$ and $\{(y,b),(z,c)\}$ are both locally consistent.

Definition

Strong Path Consistency (sPC) is Arc Consistency together with Path Consistency.

Remark.

Enforcing AC on a PC constraint network guarantees sPC.

Path Consistency

Definition

- A constraint network P is PC iff for every locally consistent instantiation $\{(x, a), (y, b)\}$ on P, there exists a value c in the domain of any third variable z of P such that $\{(x, a), (z, c)\}$ and $\{(y, b), (z, c)\}$ are both locally consistent.

Definition

Strong Path Consistency (sPC) is Arc Consistency together with Path Consistency.

Remark.

Enforcing AC on a PC constraint network guarantees sPC.

Path Consistency

Definition

- A constraint network P is PC iff for every locally consistent instantiation $\{(x, a), (y, b)\}$ on P, there exists a value c in the domain of any third variable z of P such that $\{(x, a), (z, c)\}$ and $\{(y, b), (z, c)\}$ are both locally consistent.

Definition

Strong Path Consistency (sPC) is Arc Consistency together with Path Consistency.

Remark.

Enforcing AC on a PC constraint network guarantees sPC.

Dual Consistency

Definition

- A pair of values $\{(x, a), (y, b)\}$ on a constraint network P is DC iff $(y, b) \in \mathrm{AC}(P|_{x=a})$ and $(x, a) \in \mathrm{AC}(P|_{y=b})$.
- A constraint network P is DC iff every pair of values $\{(x, a), (y, b)\}$ on P is DC-consistent.

Remark.

CDC (Conservative DC) is DC restricted on existing binary constraints.

Dual Consistency

Definition

- A pair of values $\{(x, a), (y, b)\}$ on a constraint network P is DC iff $(y, b) \in \mathrm{AC}(P|_{x=a})$ and $(x, a) \in \mathrm{AC}(P|_{y=b})$.
- A constraint network P is DC iff every pair of values $\{(x, a), (y, b)\}$ on P is DC-consistent.

Remark.

CDC (Conservative DC) is DC restricted on existing binary constraints.

Dual Consistency

Definition

- A pair of values $\{(x, a), (y, b)\}$ on a constraint network P is DC iff $(y, b) \in \mathrm{AC}(P|_{x=a})$ and $(x, a) \in \mathrm{AC}(P|_{y=b})$.
- A constraint network P is DC iff every pair of values $\{(x, a), (y, b)\}$ on P is DC-consistent.

Remark.

CDC (Conservative DC) is DC restricted on existing binary constraints.

Properties

Proposition

- *DC is strictly stronger than PC*
- *On binary CNs, DC is equivalent to PC*

Proposition

For any constraint network P, we have:

- $\mathrm{AC} \circ DC(P) = sDC(P)$
- $\mathrm{AC} \circ \mathrm{CDC}(P) = \mathrm{sCDC}(P)$

$s\phi$ is $\phi + AC$

A sCDC (Strong Conservative Dual Consistency) Algorithm

Algorithm 3: sCDC

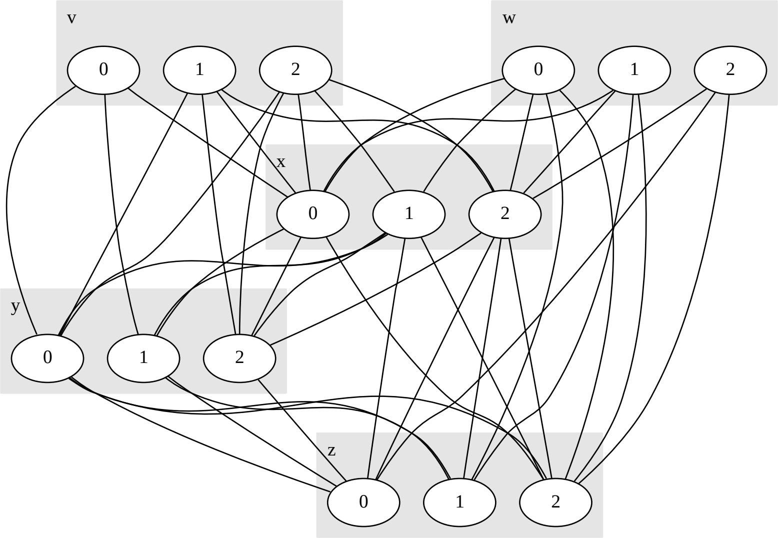
```
$P \leftarrow AC(P)$                          // AC is initially enforced
$finished \leftarrow false$
repeat
    $finished \leftarrow true$
    foreach $x \in vars(P)$ do
        if $revise\text{-}sCDC(x)$ then
            $P \leftarrow AC(P)$              // AC is maintained
            $finished \leftarrow false$
until $finished$
```

A sCDC (Strong Conservative Dual Consistency) Algorithm

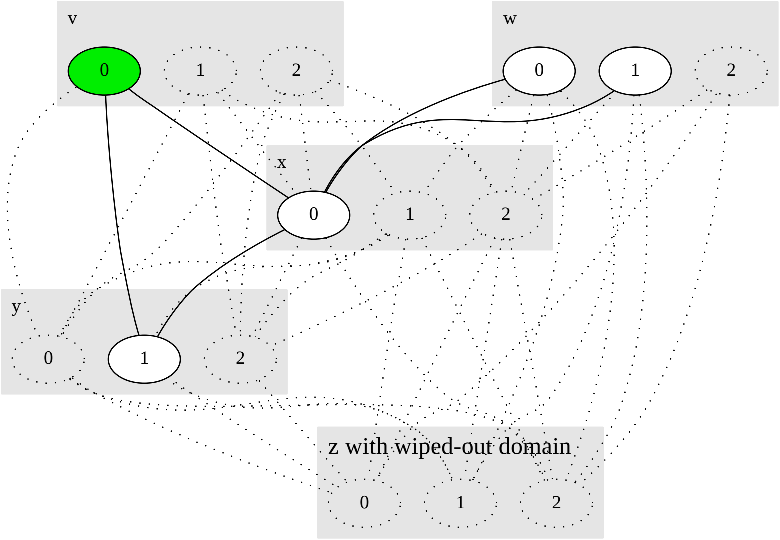
Algorithm 4: revise-sCDC(**var** x: variable): Boolean

```
$modified \leftarrow false$
foreach $value\ a \in dom(x)$ do
    $P' \leftarrow AC(P|_{x=a})$                          // Singleton check on $(x, a)$
    if $P' = \bot$ then
        remove $a$ from $dom(x)$                          // SAC-inconsistent
        $modified \leftarrow true$
    else
        foreach $constraint\ c_{xy} \in ctrs(P)$ do
            foreach $value\ b \in dom(y)$ do
                if $b \notin dom^{P'}(y)$ then
                    remove $(a, b)$ from $rel(c_{xy})$    // CDC-inconsistent
                    $modified \leftarrow true$
return $modified$
```

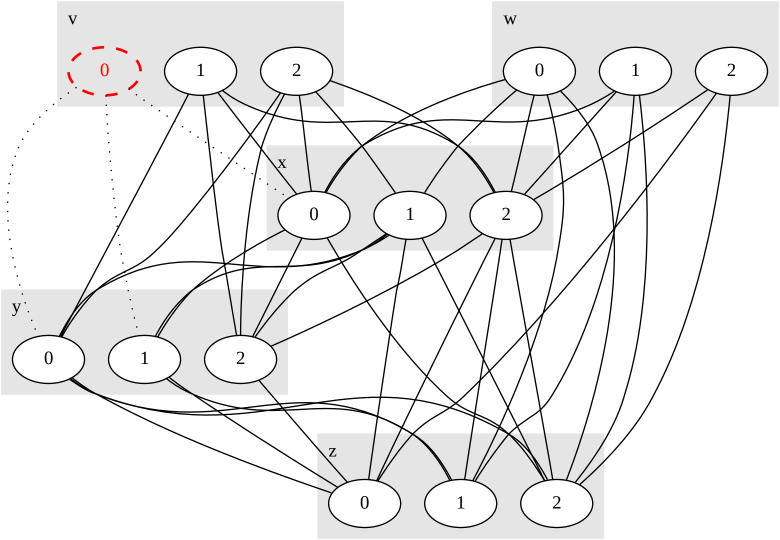
Example



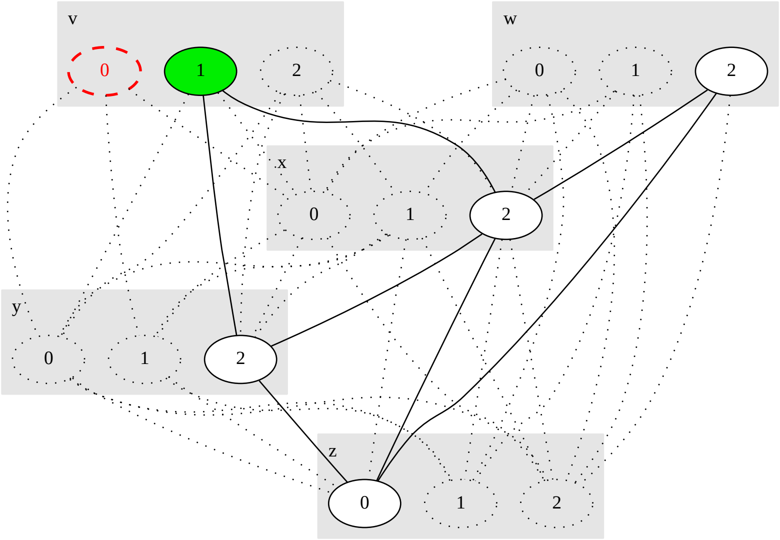
Example



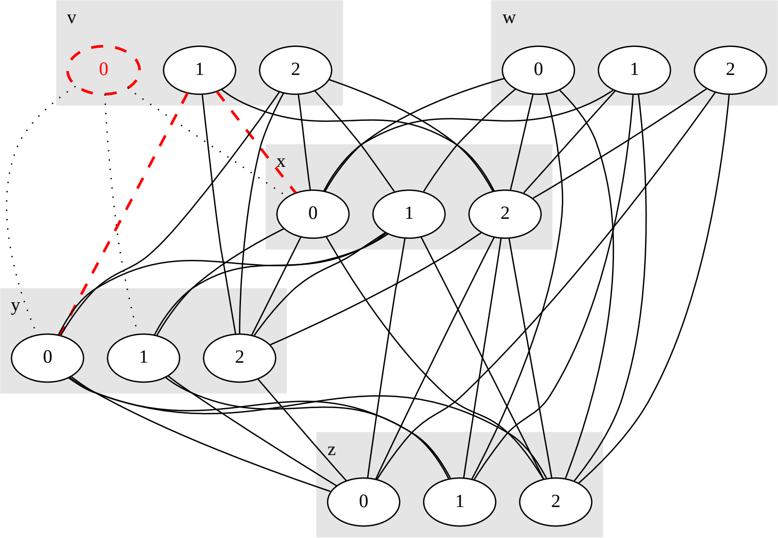
Example



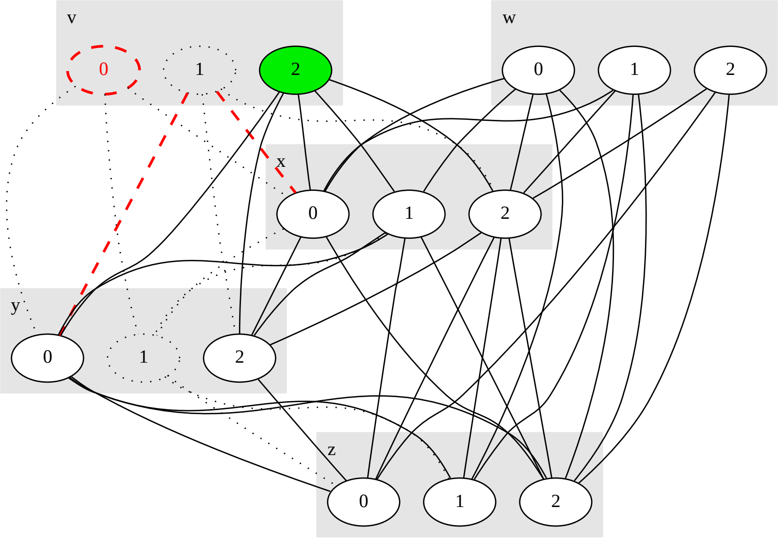
Example



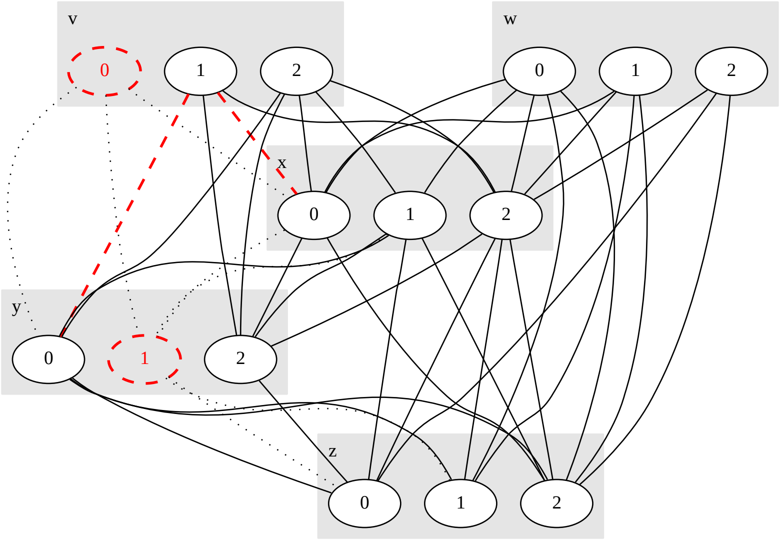
Example



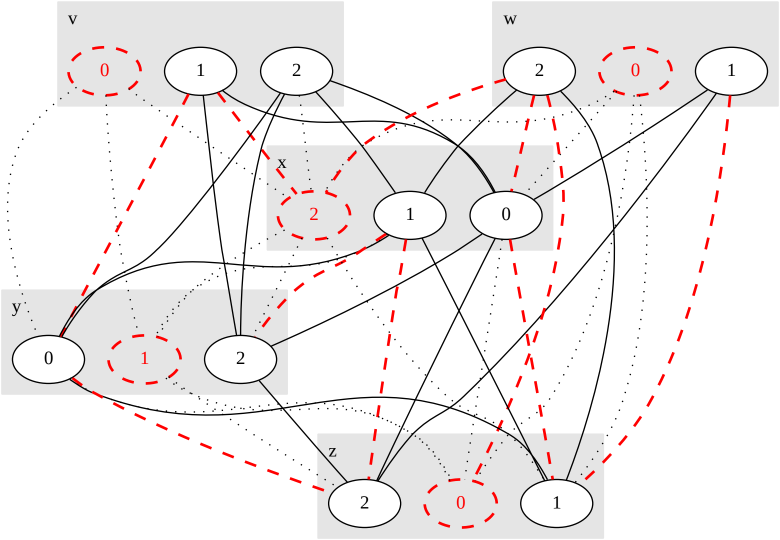
Example



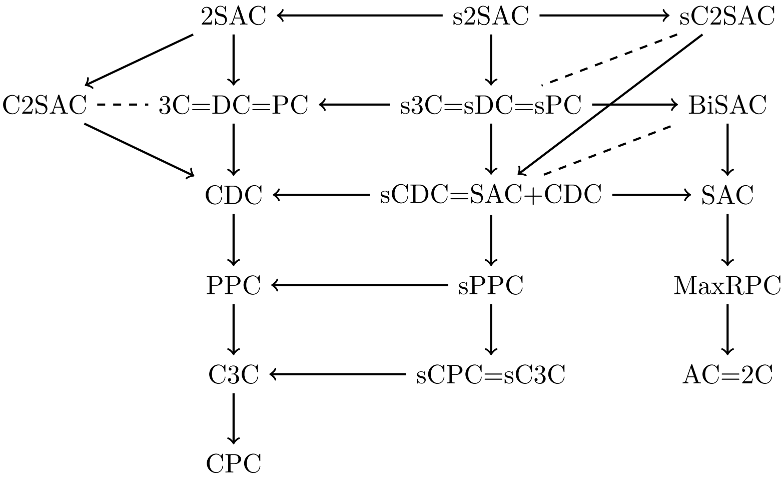
Example



Example

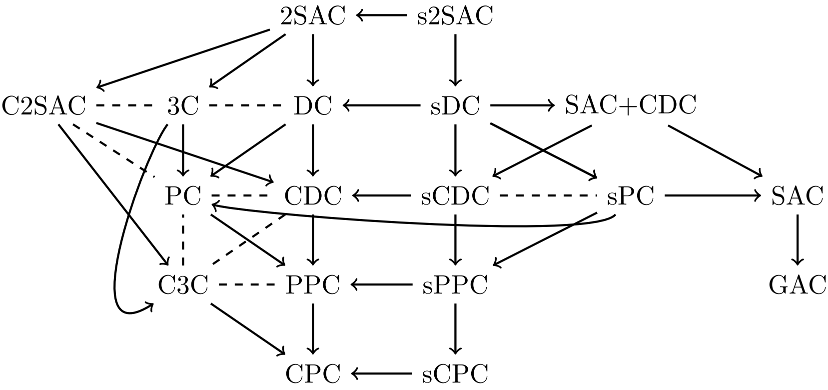


Relationships between 2-order Consistencies (binary CNs)



⟶ strictly stronger · - - - incomparable = equivalent

Relationships between 2-order Consistencies (non-binary)



2SAC s2SAC

C2SAC 3C DC sDC SAC+CDC

PC CDC sCDC sPC SAC

C3C PPC sPPC GAC

CPC sCPC

⟶ strictly stronger - - - incomparable = equivalent

PWC

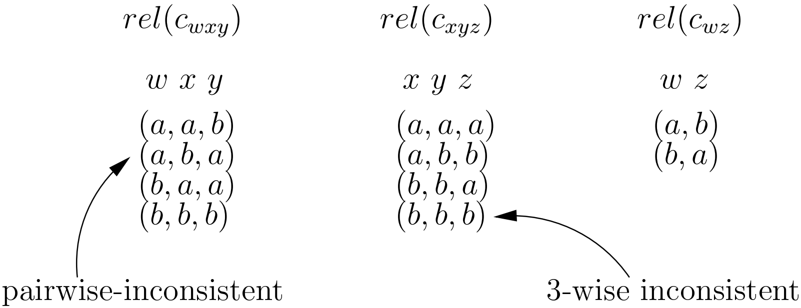
Definition (Pairwise Consistency)

- A tuple allowed by a constraint c is pairwise-consistent with respect to a constraint $c' \neq c$ iff it can be extended over $scp(c')$ into an instantiation that satisfies c'.
- A constraint network P is pairwise-consistent iff any tuple allowed by a constraint c of P is pairwise-consistent with respect to any constraint $c' \neq c$ of P.

Definition (k-wise Consistency)

- A tuple allowed by a constraint c is k-wise-consistent with respect to a set C of $k-1$ constraints iff it can be extended over $\cup_{c' \in C} scp(c')$ into an instantiation that satisfies any constraint in C.
- A constraint network P is k-wise-consistent iff any tuple allowed by a constraint c of P is k-wise-consistent with respect to any set C of $k-1$ constraints of P.

Example



$rel(c_{wxy})$	$rel(c_{xyz})$	$rel(c_{wz})$
$w\ x\ y$	$x\ y\ z$	$w\ z$
(a, a, b)	(a, a, a)	(a, b)
(a, b, a)	(a, b, b)	(b, a)
(b, a, a)	(b, b, a)	
(b, b, b)	(b, b, b)	

pairwise-inconsistent

3-wise inconsistent

Figure: Three "intersecting" constraints. The tuple (a, b, a) in $rel(c_{wxy})$ is not pairwise-consistent since it cannot be extended to c_{xyz}. The tuple (b, b, b) in $rel(c_{xyz})$ is not 3-wise consistent since it cannot be extended to the two other constraints.

Bartak, R., & Erben, R. 2004.
A new algorithm for singleton arc consistency.
Pages 257–262 of: Proceedings of FLAIRS'04.

Bessiere, C., & Debruyne, R. 2004.
Theoretical analysis of singleton arc consistency.
Pages 20–29 of: Proceedings of ECAI'04 workshop on modelling and solving problems with constraints.

Bessiere, C., & Debruyne, R. 2005.
Optimal and suboptimal singleton arc consistency algorithms.
Pages 54–59 of: Proceedings of IJCAI'05.

Bessiere, C., & Régin, J.-C. 1997.
Arc consistency for general constraint networks: preliminary results.
Pages 398–404 of: Proceedings of IJCAI'97.

Cheng, Kenil C. K., & Yap, Roland H. C. 2010.
An MDD-based Generalized Arc Consistency Algorithm for Positive and Negative Table Constraints and Some Global Constraints.
Constraints, **15**(2), 265–304.

Debruyne, R., & Bessiere, C. 1997.
Some practical filtering techniques for the constraint satisfaction problem.
Pages 412–417 of: Proceedings of IJCAI'97.

Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.-C., & Schaus, P. 2016.
Compact-Table: efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets.
Pages 207–223 of: Proceedings of CP'16.

Gent, I., Jefferson, C., Miguel, I., & Nightingale, P. 2007.
Data Structures for Generalised Arc Consistency for Extensional Constraints.
Pages 191–197 of: Proceedings of AAAI'07.

Katsirelos, George, & Walsh, Toby. 2007.
A compression algorithm for large arity extensional constraints.
Pages 379–393 of: Proceedings of CP'07.

Lecoutre, C. 2008.
Optimization of Simple Tabular Reduction for Table Constraints.
Pages 128–143 of: Proceedings of CP'08.

Lecoutre, C., & Cardon, S. 2005.
A greedy approach to establish singleton arc consistency.
Pages 199–204 of: Proceedings of IJCAI'05.

Lecoutre, C., Likitvivatanavong, C., & Yap, R. 2012.
A path-optimal GAC algorithm for table constraints.
Pages 510–515 of: Proceedings of ECAI'12.

Lecoutre, Christophe, & Szymanek, Radoslaw. 2006.
Generalized Arc Consistency for Positive Table Constraints.
Pages 284–298 of: Proceedings of CP'06.

Lhomme, Olivier, & Régin, Jean-Charles. 2005.
A fast arc consistency algorithm for n-ary constraints.
Pages 405–410 of: Proceedings of AAAI'05.

Mairy, Jean-Baptiste, Hentenryck, Pascal Van, & Deville, Yves. 2012.
An Optimal Filtering Algorithm for Table Constraints.
Pages 496–511 of: Proceedings of CP'12.

Mairy, Jean-Baptiste, Deville, Yves, & Lecoutre, Christophe. 2015.
The Smart Table Constraint.
Pages 271–287 of: Proceedings of CPAIOR'15.

Nightingale, P., Gent, I.P., Jefferson, C., & Miguel, I. 2013.
Short and Long Supports for Constraint Propagation.
Journal of Artificial Intelligence Research, **46**, 1–45.

Perez, Guillaume, & Régin, Jean-Charles. 2014.
Improving GAC-4 for Table and MDD Constraints.
Pages 606–621 of: Proceedings of CP'14.

Ullmann, Julian R. 2007.
Partition search for non-binary constraint satisfaction.
Information Science, **177**, 3639–3678.