Constraint Reasoning

Part 1

Christophe Lecoutre
lecoutre@cril.fr

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

ECAI Tutorial – Montpellier – August 28th, 2012

Outline

1. Modelling Constraint Problems
2. Solving Constraint Satisfaction Problems
3. Constraint Propagation
4. Filtering Algorithms for Table, MDD and Regular Constraints
5. Strong Inference

Outline

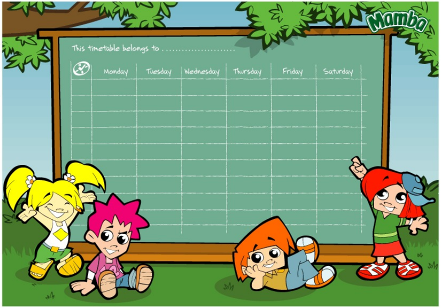
1. Modelling Constraint Problems
2. Solving Constraint Satisfaction Problems
3. Constraint Propagation
4. Filtering Algorithms for Table, MDD and Regular Constraints
5. Strong Inference

Ubiquity of Constraints

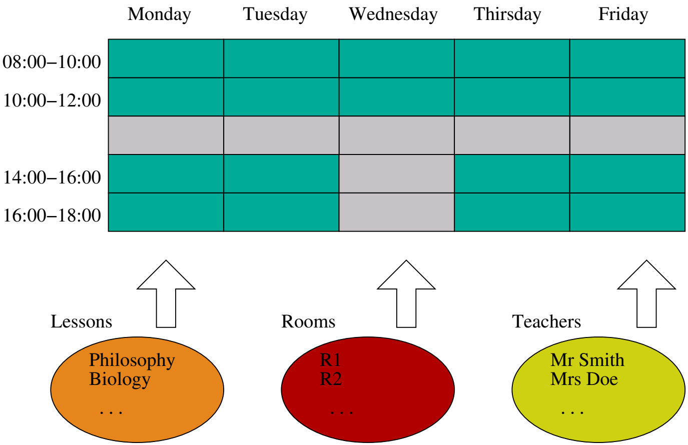
A number of human activities requires dealing with the concept of constraints. A *constraint* limits the field of possibilities in a certain universe/context.

Example

When a school timetable must be set at the beginning of the school year, the person in charge of this task has to take into account many kinds of constraints.



Timetabling Problem

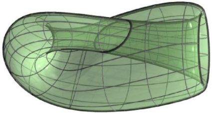


Constraint Programming

Constraint programming (CP) is a general framework whose objective is to propose simple, general and efficient algorithmic solutions to constraint problems.

They are then two main issues that need to be addressed when this framework is used to deal with a combinatorial problem:

1. In a first modelling stage, the problem must be represented by introducing variables, constraints, and potentially objective functions.



2. In a second solving stage, the problem modelled by the user must be tackled by a software tool in order to automatically obtain one solution, all solutions or an optimal solution.

Constraint Satisfaction

The **constraint satisfaction problem** (CSP) resides at the core of constraint programming. An *instance* of this problem is represented by a **constraint network** (CN).

CSP SAT

Note that SAT is closely related to CSP:

- variables are Boolean
- constraints are clauses (disjunctions of variables and their negations)

Remark

SAT and CSP are NP-complete problems

Warning

We shall only deal with discrete variables

Variables and Constraints

Definition (Variable)

A variable (with name) x is an unknown entity that must be given a value from a set called the current domain of x and denoted by $dom(x)$.

Definition (Constraint)

A constraint (with name) c is defined over a (totally ordered) set of variables, called scope of c and denoted by $scp(c)$, by a mathematical relation that describes the set of tuples allowed by c for the variables of its scope.

Remark

The arity of a constraint c is the number of variables involved in c, i.e. $|scp(c)|$.

Representation of Constraints

Formally, a constraint is defined a mathematical relation. In practice there are three different ways of representing a constraint:

- in intension, by using a Boolean formula (predicate),
- implicitly by referring to a so-called global constraint,
- in extension, by listing tuples.

Intensional Constraints

Definition (Intensional Constraint)

A constraint c is intensional (or defined in intension) iff it is described by a Boolean formula (predicate) that represents a function that is defined from $\Pi_{x \in scp(c)} dom(x)$ to $\{false, true\}$.

Example

A binary constraint:

$$c_{vw} : v \leq w + 2$$

A ternary constraint:

$$c_{xyz} : x \neq y \wedge x \neq z \wedge y \neq z$$

Global Constraints

Definition (Global Constraint)

A global constraint is a constraint pattern that captures a precise relational semantics and that can be applied over an arbitrary number of variables.

For example, the semantics of AllDifferent is that all variables must take a different value.

Example

Our previous ternary constraint can be defined by:

$$c_{xyz} : \mathit{AllDifferent}(x, y, z)$$

Extensional Constraints

Definition (Extensional Constraint)

A constraint c is extensional (or defined in extension) iff it is explicitly described, either positively by listing the tuples allowed by c or negatively by listing the tuples disallowed by c.

Example

If $dom(x) \times dom(y) \times dom(z) = \{0, 1, 2\}^3$, then our ternary constraint can be defined positively by:

$$c_{xyz} : \left\{ \begin{array}{c} (0,1,2), \\ (0,2,1), \\ (1,0,2), \\ (1,2,0) \\ (2,1,0), \\ (2,0,1) \end{array} \right\}$$

Constraint Networks

Definition

A Constraint Network (CN) P is composed of:

- a finite set of variables, denoted by $vars(P)$,
- a finite set of constraints, denoted by $cons(P)$.

Warning

We will call a pair (x, a) with $x \in vars(P)$ and $a \in dom(x)$ a **value of** P.

Sudoku as a CN

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

Sudoku as a CN

We can simply define a CN P such that:

- $vars(P) =$

$$\begin{aligned} &\{x_{1,1}, x_{1,2}, \ldots, x_{1,9},\\ &\ x_{2,1}, x_{2,2}, \ldots, x_{2,9},\\ &\ \ldots\\ &\} \textit{ with } dom(x_{i,j}) = \{0, 1, \ldots, 9\}, \forall i, j \in 1..9 \end{aligned}$$

- $cons(P) =$

$$\begin{aligned} &\{AllDifferent(x_{1,1}, x_{1,2}, \ldots, x_{1,9}),\\ &\ AllDifferent(x_{2,1}, x_{2,2}, \ldots, x_{2,9}),\\ &\ \ldots\\ &\} \end{aligned}$$

Remark

For each hint, add unary constraints

A Solution to the Sudoku Instance

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

Outline

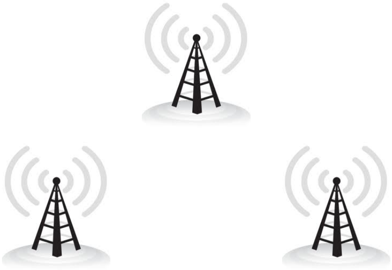
1. Modelling Constraint Problems
2. Solving Constraint Satisfaction Problems
3. Constraint Propagation
4. Filtering Algorithms for Table, MDD and Regular Constraints
5. Strong Inference

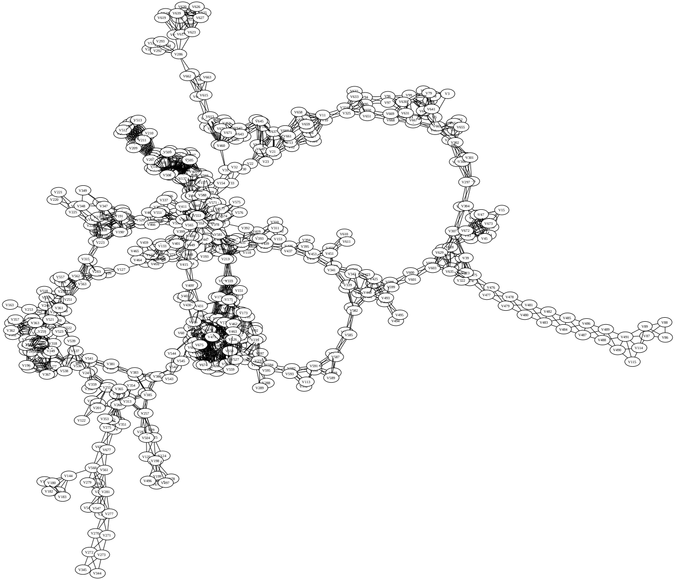
RLFAP (CELAR, project CALMA, Cabon et al. 1999)

Problem: assigning frequencies to radio-links while avoiding interferences

Model:

- a set of variables to represent unidirectional radio links
- a set of binary constraints of the form
  - $|x_i - x_j| = d_{ij}$
  - $|x_i - x_j| > d_{ij}$
- several criteria to optimize (minimum span, minimum cardinality, etc.)

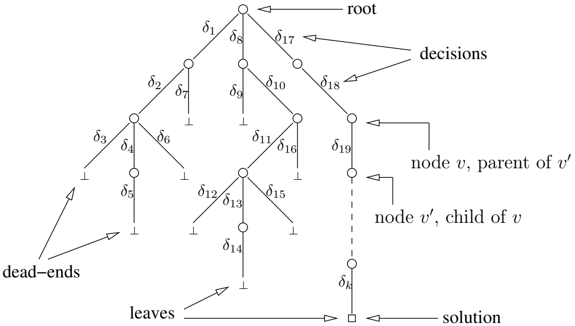




Structure of CSP instances: scen11, scen11-f12, scen11-f6, scen11-f1
680 variables, 4,103 binary constraints

Solving Problem Instances with Backtrack Search

- Complete search
- Depth-first exploration
- Backtracking mechanism
- Interleaving of
  - decisions (e.g. variable assignments)
  - constraint propagation



Backtrack Search (using Binary Branching)

Algorithm 1: backtrackSearch(P: CN): Boolean

$P \leftarrow \phi(P)$
if $\exists x \in vars(P), dom(x) = \emptyset$ **then**
  return *false*
if $\forall x \in vars(P), |dom(x)| = 1$ **then**
  return *true*
select a value (x, a) of P such that $|dom(x)| > 1$
return $backtrackSearch(P|_{x=a}) \vee backtrackSearch(P|_{x \neq a})$

Remark

ϕ denotes the process of constraint propagation

Results (1) – MAC

Instances	nodes	CPU
scen11		$> 10{,}000$
scen11-f12		$> 10{,}000$
scen11-f8		$> 10{,}000$
scen11-f8		$> 10{,}000$
scen11-f4		$> 10{,}000$
scen11-f2		$> 10{,}000$
scen11-f1		$> 10{,}000$

Using Heuristics to Guide Search

General principles:

- It is better to start assigning those variables that belong to the most difficult part(s) of the problem instance: "to succeed, try first where you are most likely to fail" (fail-first principle).
- To find a solution quickly, it is better to select a value that belongs to the most promising subtree.
- The initial variable/value choices are particularly important.

Some classical variable ordering heuristics :

- *dom*
- *dom/deg*
- *dom+deg*

Results (2) – MAC-*dom/deg*

Instances	nodes	CPU (2)	CPU (1)
scen11	31,816	5.42	$>$ 10,000
scen11-f12		$>$ 10,000	$>$ 10,000
scen11-f8		$>$ 10,000	$>$ 10,000
scen11-f6		$>$ 10,000	$>$ 10,000
scen11-f4		$>$ 10,000	$>$ 10,000
scen11-f2		$>$ 10,000	$>$ 10,000
scen11-f1		$>$ 10,000	$>$ 10,000

Adaptive Variable Ordering Heuristics

The heuristic *dom/wdeg* is a generic state-of-the-art variable ordering heuristic.

The principle is the following:

- a weight is associated with each constraint,
- everytime a conflict occurs while filtering through a constraint c, the weight associated with c is incremented,
- the weight of a variable is the sum of the weights of all its involving constraints.

The interest is that this heuristic is **adaptive**, with the expectation to focus on the hard part(s) of the instance.

Results (3) – MAC-*dom/wdeg*

Instances	nodes	CPU (3)	CPU (2)
scen11	912	1.47	5.42
scen11-f12	699	1.49	$>$ 10,000
scen11-f8	14,077	2.8	$>$ 10,000
scen11-f6	252,557	25.2	$>$ 10,000
scen11-f4	3,477,514	292	$>$ 10,000
scen11-f2	38,263,495	3,158	$>$ 10,000
scen11-f1	96,066,349	7,805	$>$ 10,000

Illustration of Constraint Weighting with scen11-f6

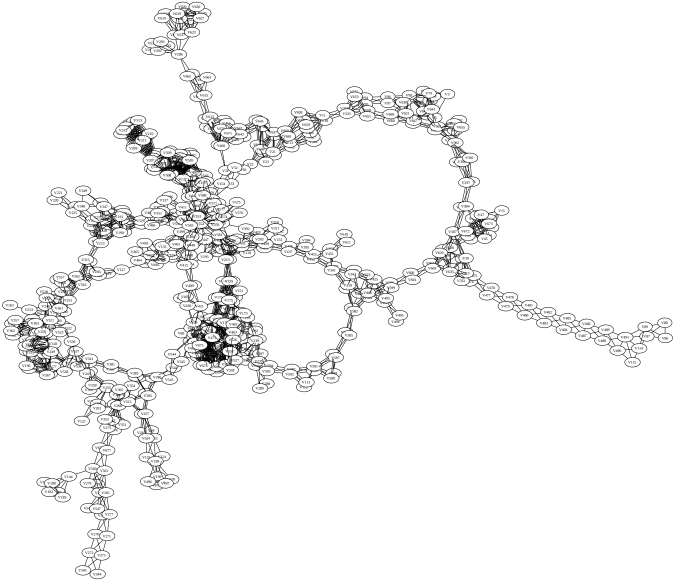


Illustration of Constraint Weighting with scen11-f6

Illustration of Constraint Weighting with scen11-f6

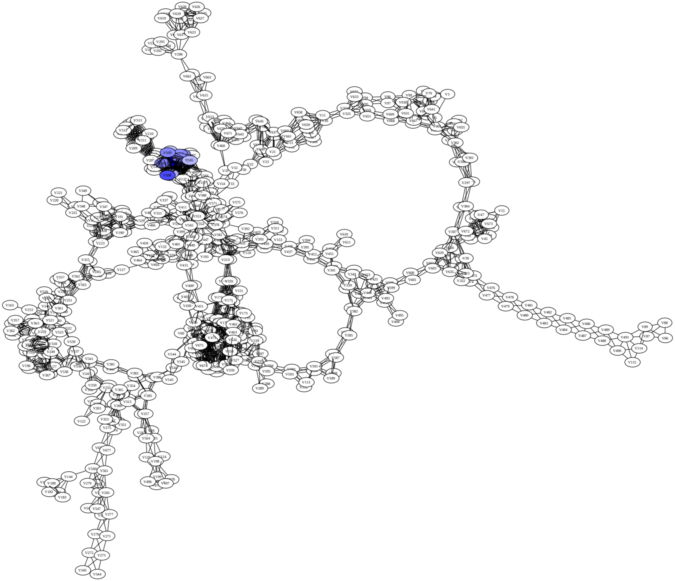
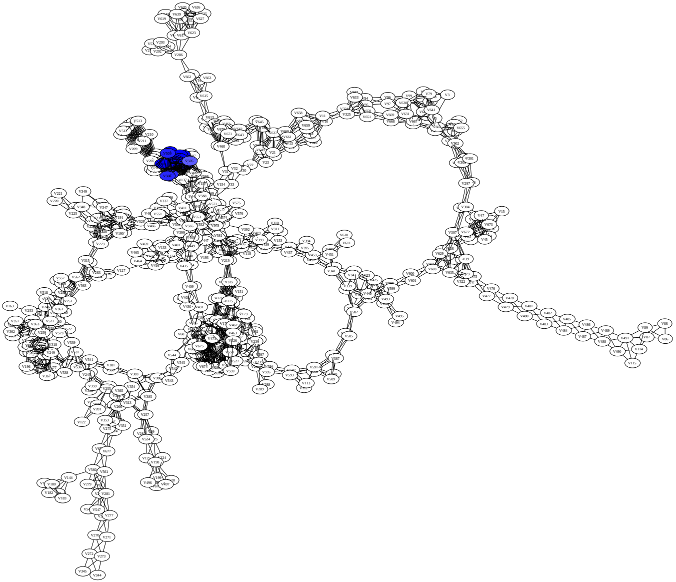


Illustration of Constraint Weighting with scen11-f6



Restarts

Restarting search may help the constraint solver to find far quicker a solution because :

- it permits diversification of search
- it avoids being stuck in a large unsatisfiable subtree after some bad initial choices
- it can be combined with nogood recording

Results (4) – MAC-*dom/wdeg*-nrr

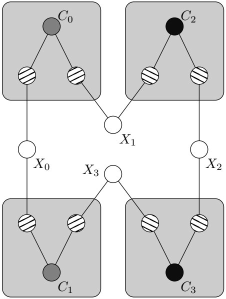
Instances	nodes	CPU (4)	CPU (3)
scen11	882	1.48	1.47
scen11-f12	353	1.39	1.49
scen11-f8	1,264	1.56	2.80
scen11-f6	33,542	4.45	25.20
scen11-f4	421,097	37.2	292.0
scen11-f2	4,310,576	356	3,158
scen11-f1	11,096,549	921	7,805

Symmetry Breaking

Definition

Let P be a CN with $vars(P) = \{x_1, \ldots, x_n\}$. A variable symmetry σ of P is a bijection on $vars(P)$ such that $\{x_1 = a_1, \ldots, x_n = a_n\}$ is a solution of P iff $\{\sigma(x_1) = a_1, \ldots, \sigma(x_n) = a_n\}$ is a solution of P.

First step to break symmetries **automatically**: construction of a colored graph.



Symmetry Breaking

Second step to break symmetries automatically: execution of a a software tool such as Nauty or Saucy to compute an automorphism group.

Third step to break symmetries automatically: post a constraint *lex* for every generator of the group.

Definition

A laxicographic constraint *lex* is defined on two vectors $\overrightarrow{X}$ and $\overrightarrow{Y}$ of variables. We have:

$$\overrightarrow{X} = \langle x_1, x_2, \ldots, x_r \rangle \leq_{lex} \overrightarrow{Y} = \langle y_1, y_2, \ldots, y_r \rangle$$

iff

$$\overrightarrow{X} = \overrightarrow{Y} = \langle\rangle \textit{ (both vectors are empty)}$$
$$\textit{or } x_1 < y_1$$
$$\textit{or } x_1 = y_1 \textit{ and } \langle x_2, \ldots, x_r \rangle \leq_{lex} \langle y_2, \ldots, y_r \rangle$$

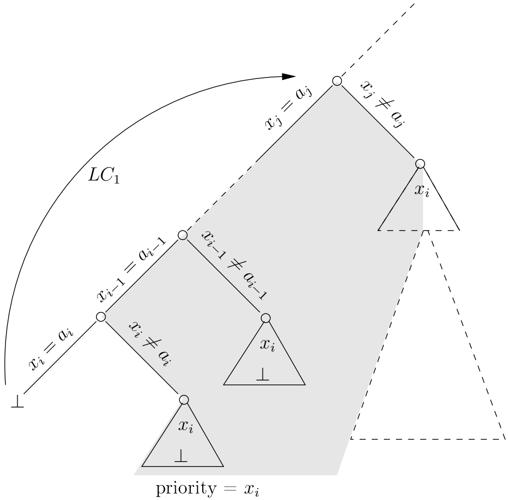
Results (5) – MAC-*dom/wdeg*-nrr-sb

Instances	nodes	CPU (5)	CPU (4)
scen11	1,103	1.59	1.48
scen11-f12	571	1.51	1.39
scen11-f8	654	1.56	1.56
scen11-f6	1,388	1.69	4.45
scen11-f4	2,071	1.86	37.20
scen11-f2	12,027	2.96	356.00
scen11-f1	13,125	3.03	921.00

Last-conflict based Reasoning

The principle is the following: after each conflict (dead-end), keep selecting the last assigned variable as long as no consistent value can be found.

This looks like a lazy form of intelligent backtracking



Results (6) – MAC-*dom/wdeg*-nrr-sb-lc

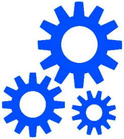
Instances	nodes	CPU (6)	CPU (5)
scen11	1,173	1.57	1.59
scen11-f12	187	1.48	1.51
scen11-f8	191	1.48	1.56
scen11-f6	273	1.51	1.69
scen11-f4	957	1.82	1.86
scen11-f2	5,101	2.19	2.96
scen11-f1	11,305	2.84	3.03

Strong Preprocessing

Before search, one can try to make the CN more explicit.

For example, this can be achieved by enforcing some properties that identify inconsistent pairs of values.

Here, strong Conservative Dual Consistency (sCDC) combined with symmetry breaking is enough to solve instances scen11-fx **without any search**.



Results (7) – sCDC-MAC-sb

Instances	nodes	CPU (7)	CPU (6)
scen11	680 (83435)	7.82	1.57
scen11-f12	0 (1474)	1.59	1.48
scen11-f8	0 (3793)	1.86	1.48
scen11-f6	0 (4391)	1.96	1.51
scen11-f4	0 (16207)	2.88	1.82
scen11-f2	0 (29044)	3.78	2.19
scen11-f1	0 (43808)	4.95	2.84

Outline

1. Modelling Constraint Problems
2. Solving Constraint Satisfaction Problems
3. Constraint Propagation
4. Filtering Algorithms for Table, MDD and Regular Constraints
5. Strong Inference

Filtering Domains through Constraints

Every constraint represents a "sub-problem" from which some inconsistent values can be eliminated, i.e., some values that belong to no solutions (of the constraint).

Several levels of filtering can be defined:

- AC (Arc Consistency): all inconsistent values are identified and eliminated
- BC (Bounds Consistency): only inconsistent values corresponding to bounds of domains are identified and eliminated
- ...

Example

Constraint $c_{xy} : x < y$ with

- $dom(x) = [10..20]$
- $dom(y) = [0..15]$

After filtering (either AC or BC), we get:

- $dom(x) = [10..14]$
- $dom(y) = [11..15]$

Example

Constraint $c_{wz} : w + 3 = z$ with

- $dom(w) = \{1, 3, 4, 5\}$
- $dom(z) = \{4, 5, 8\}$

After filtering (AC), we get:

- $dom(w) = \{1, 5\}$
- $dom(z) = \{4, 8\}$

GAC for the Constraint AllDifferent

Warning

For non-binary constraints, AC is often referred to as GAC.

Proposition

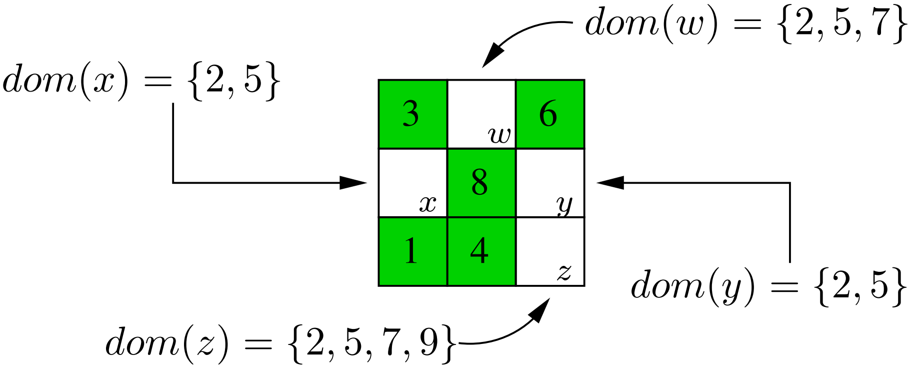
A constraint AllDifferent(X) is GAC iff

$$\forall X' \subseteq X, |dom(X')| = |X'| \Rightarrow \forall x \in X \setminus X', dom(x) = dom(x) \setminus dom(X')$$

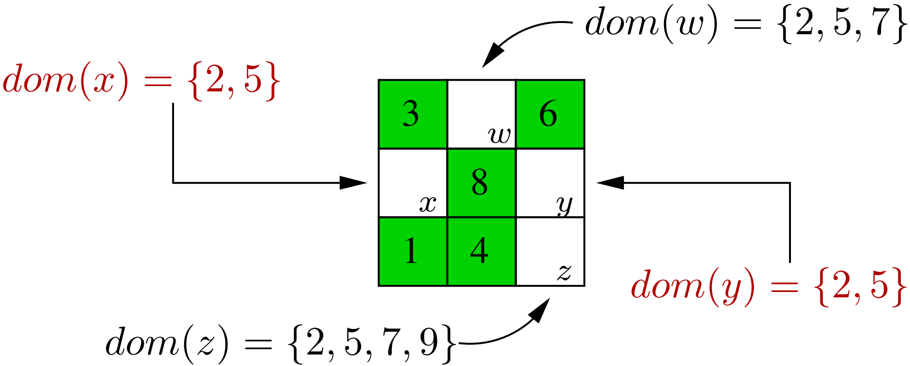
where X denotes the scope of the constraint and $dom(X') = \cup_{x' \in X'} dom(x')$

See (Régin, 1994)

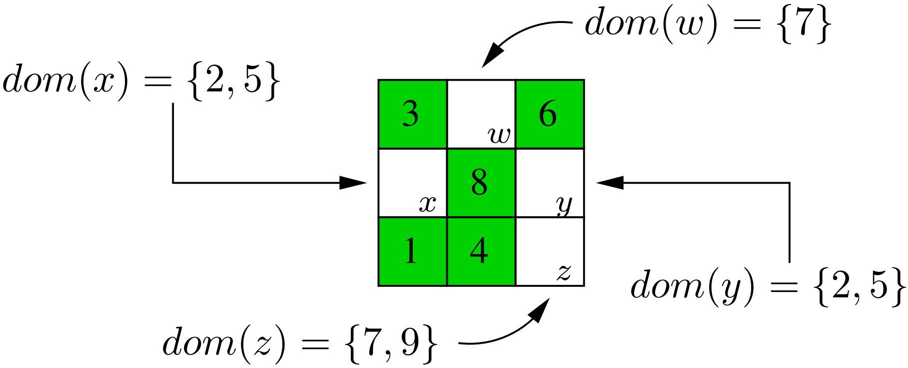
Example



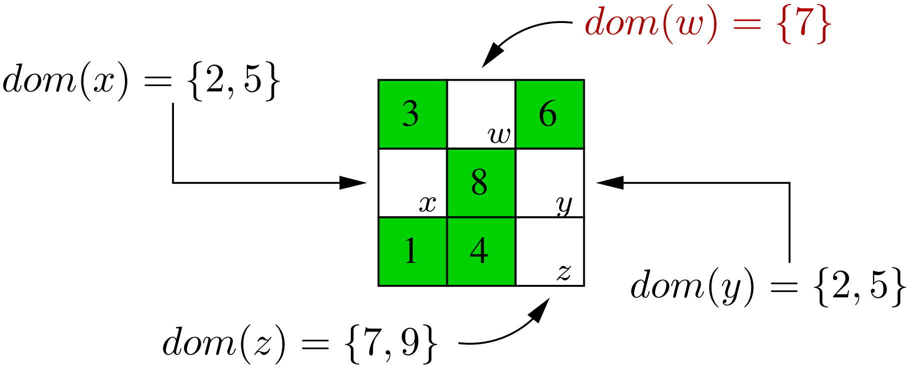
Example



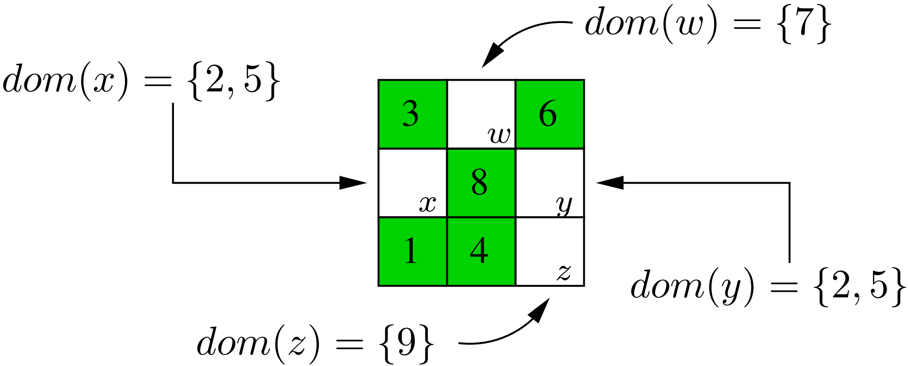
Example



Example



Example



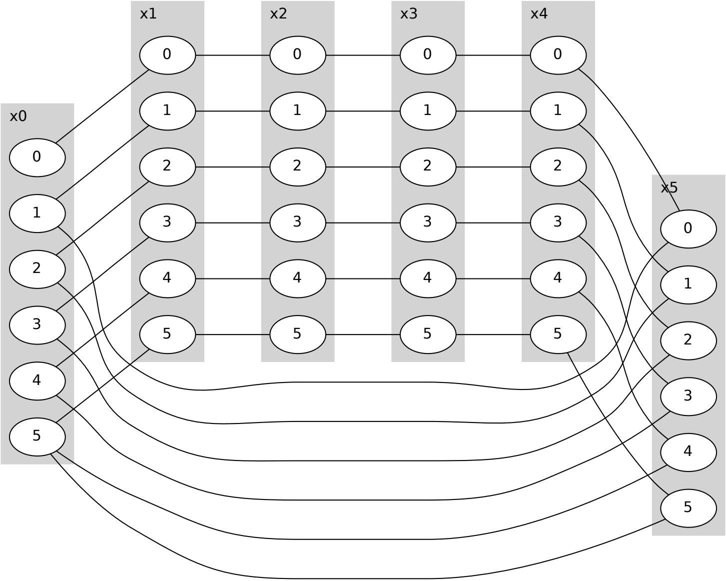
Constraint Propagation

When a constraint filters out one or several inconsistent values, this may trigger the possibility for some other constraints to filter too (and again). This process of iterative filtering operations, led constraint per constraint, is called constraint propagation.

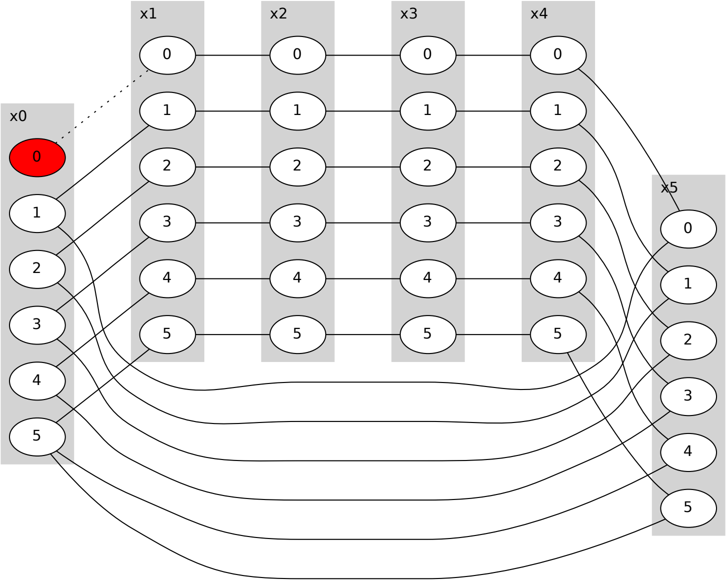
Algorithm 2: runConstraintPropagationOn(P: CN): Boolean

```
Q ← cons(P)
while Q ≠ ∅ do
    pick and delete c from Q
    X_evt ← c.filter() // X_evt denotes the set of variables with reduced
    domains (after filtering by means of c)
    if ∃x ∈ X_evt such that dom(x) = ∅ then
        return false            // global inconsistency detected
    foreach c' ∈ cons(P) such that c' ≠ c and X_evt ∩ scp(c') ≠ ∅ do
        add c' to Q
return true
```

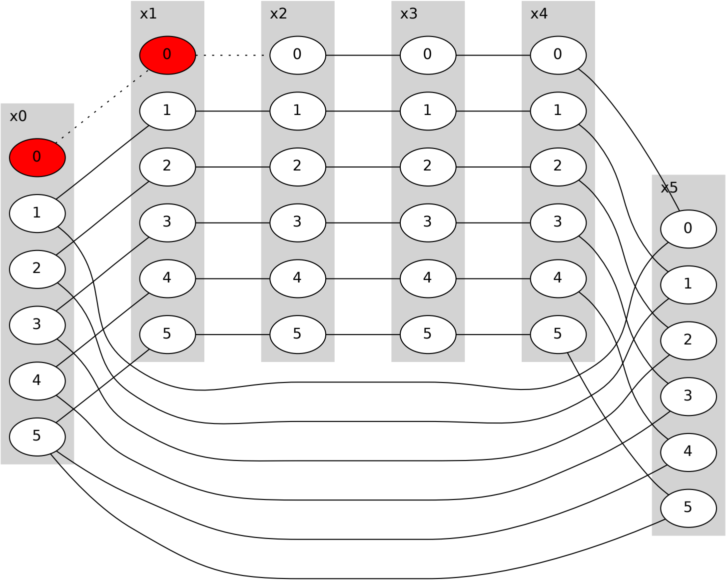
Example : domino-6-6



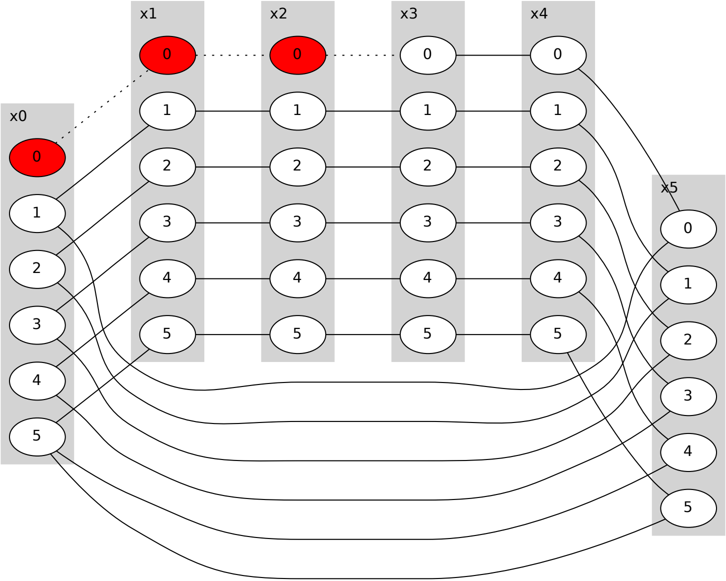
Example : domino-6-6



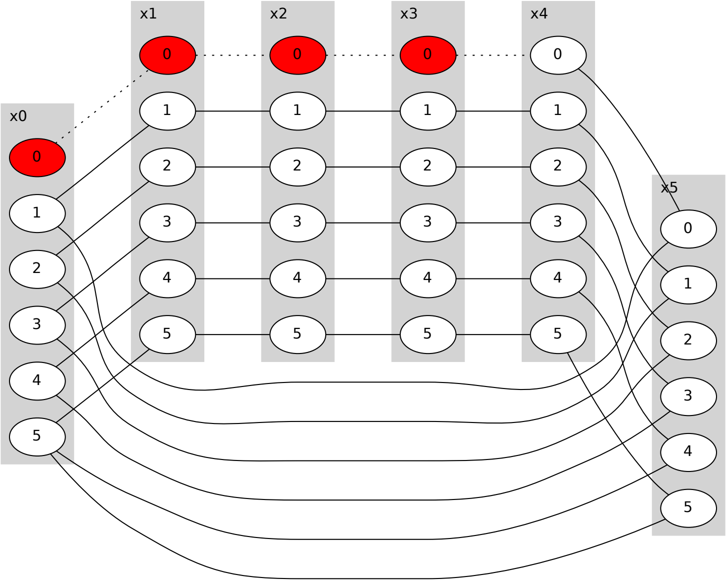
Example : domino-6-6



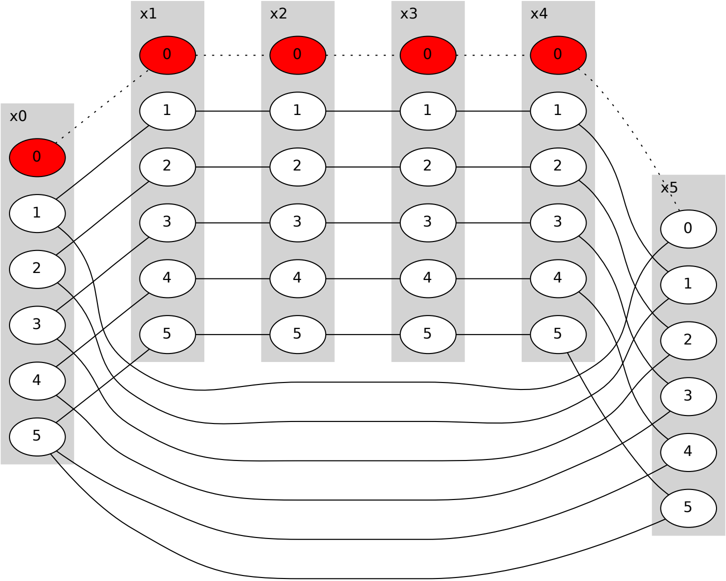
Example : domino-6-6



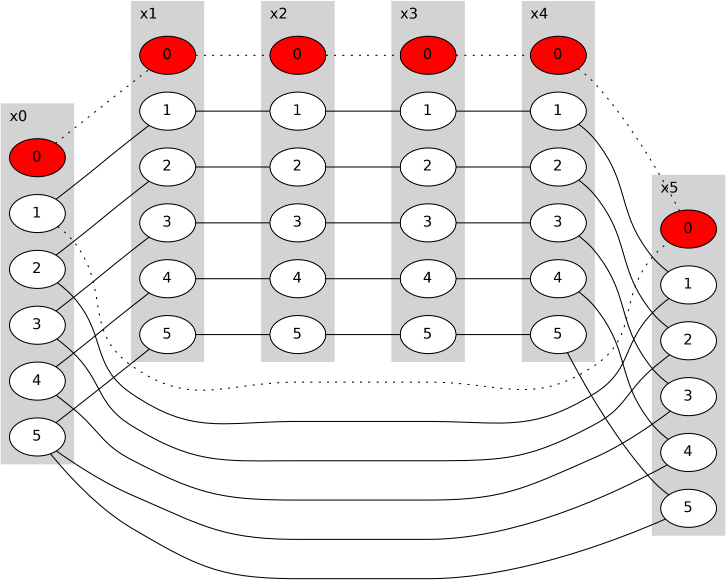
Example : domino-6-6



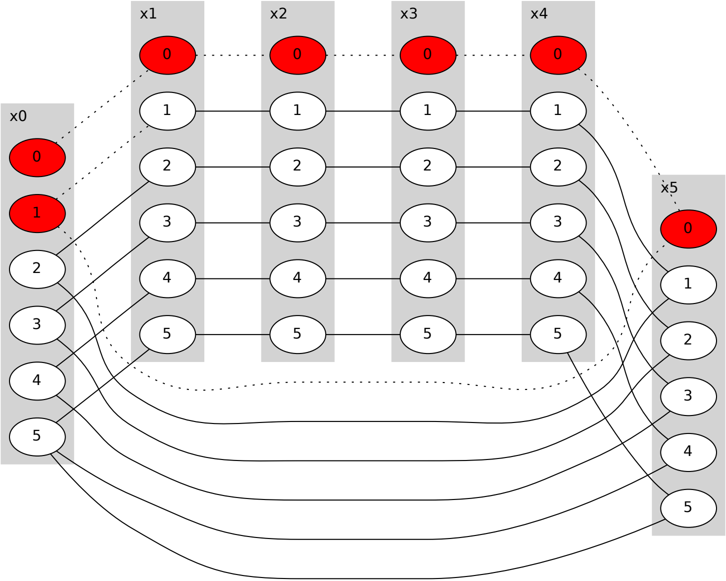
Example : domino-6-6



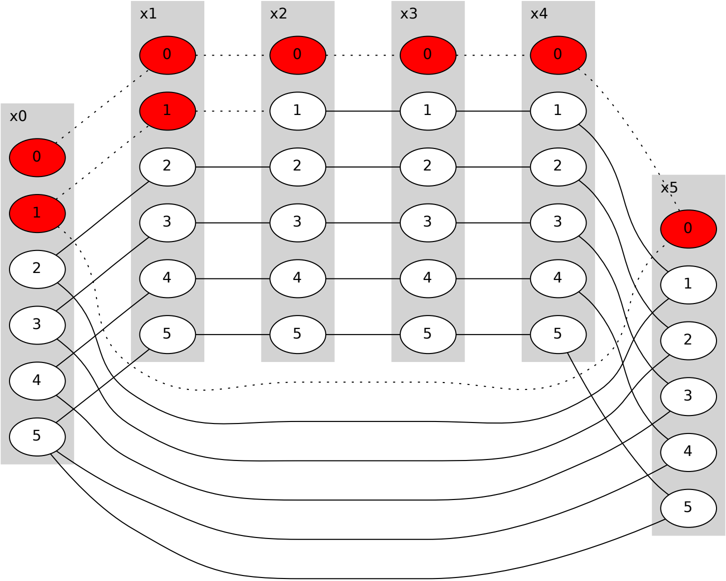
Example : domino-6-6



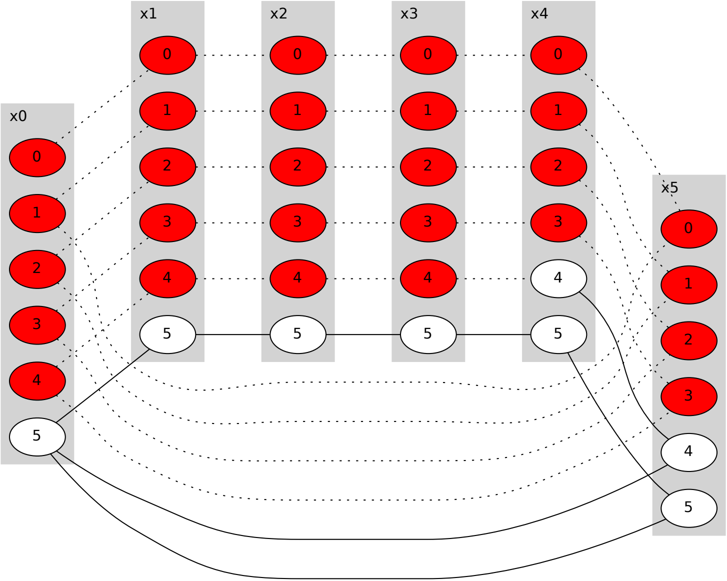
Example : domino-6-6



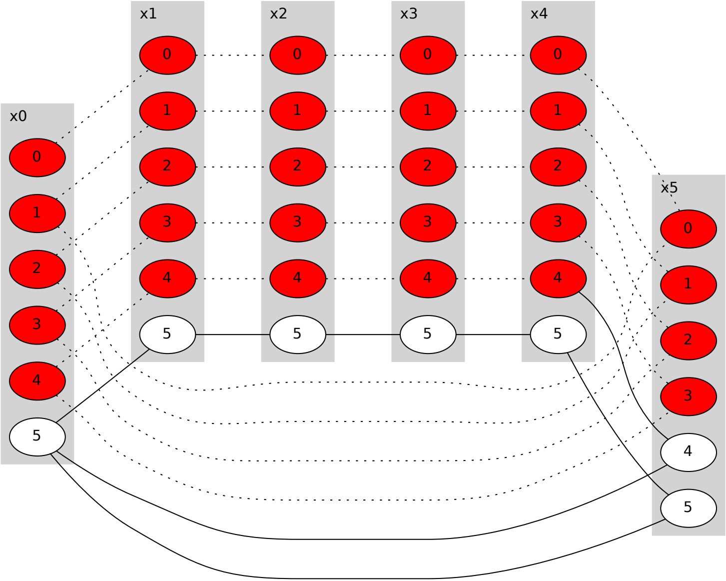
Example : domino-6-6



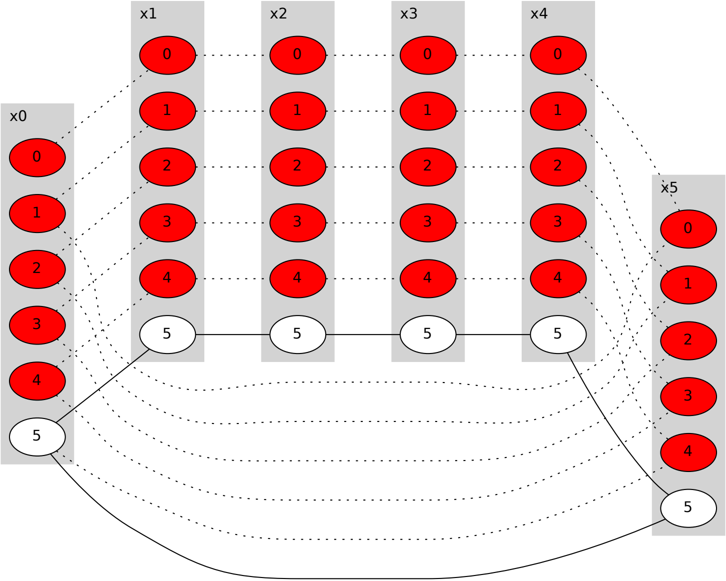
Example : domino-6-6



Example : domino-6-6



Example : domino-6-6



Generalized Arc Consistency (GAC)

Definition

Let P be a CN.

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

- If there is a constraint c involving a variable x such that there is no support for (x, a) on c, then (x, a) is **not GAC**.
- A **GAC algorithm** is an algorithm that removes all values from a CN P that are not GAC.
- A GAC algorithm computes the so-called **GAC-closure** of P by propagating constraints until a fixed-point is reached.
- A GAC algorithm is **generic** iff it can be applied to any CN (set of constraints).

Generalized Arc Consistency (GAC)

Definition

Let P be a CN.

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

- If there is a constraint c involving a variable x such that there is no support for (x, a) on c, then (x, a) is **not GAC**.
- A **GAC algorithm** is an algorithm that removes all values from a CN P that are not GAC.
- A GAC algorithm computes the so-called **GAC-closure** of P by propagating constraints until a fixed-point is reached.
- A GAC algorithm is **generic** iff it can be applied to any CN (set of constraints).

Generalized Arc Consistency (GAC)

Definition

Let P be a CN.

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

- If there is a constraint c involving a variable x such that there is no support for (x, a) on c, then (x, a) is **not GAC**.
- A **GAC algorithm** is an algorithm that removes all values from a CN P that are not GAC.
- A GAC algorithm computes the so-called **GAC-closure** of P by propagating constraints until a fixed-point is reached.
- A GAC algorithm is **generic** iff it can be applied to any CN (set of constraints).

Generalized Arc Consistency (GAC)

Definition

Let P be a CN.

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

- If there is a constraint c involving a variable x such that there is no support for (x, a) on c, then (x, a) is **not GAC**.
- A **GAC algorithm** is an algorithm that removes all values from a CN P that are not GAC.
- A GAC algorithm computes the so-called **GAC-closure** of P by propagating constraints until a fixed-point is reached.
- A GAC algorithm is **generic** iff it can be applied to any CN (set of constraints).

Generalized Arc Consistency (GAC)

Definition

Let P be a CN.

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

- If there is a constraint c involving a variable x such that there is no support for (x, a) on c, then (x, a) is **not GAC**.
- A **GAC algorithm** is an algorithm that removes all values from a CN P that are not GAC.
- A GAC algorithm computes the so-called **GAC-closure** of P by propagating constraints until a fixed-point is reached.
- A GAC algorithm is **generic** iff it can be applied to any CN (set of constraints).

Generalized Arc Consistency (GAC)

Definition

Let P be a CN.

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

- If there is a constraint c involving a variable x such that there is no support for (x, a) on c, then (x, a) is **not GAC**.
- A **GAC algorithm** is an algorithm that removes all values from a CN P that are not GAC.
- A GAC algorithm computes the so-called **GAC-closure** of P by propagating constraints until a fixed-point is reached.
- A GAC algorithm is **generic** iff it can be applied to any CN (set of constraints).

Generalized Arc Consistency (GAC)

Definition

Let P be a CN.

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

- If there is a constraint c involving a variable x such that there is no support for (x, a) on c, then (x, a) is **not GAC**.
- A **GAC algorithm** is an algorithm that removes all values from a CN P that are not GAC.
- A GAC algorithm computes the so-called **GAC-closure** of P by propagating constraints until a fixed-point is reached.
- A GAC algorithm is **generic** iff it can be applied to any CN (set of constraints).

Generalized Arc Consistency (GAC)

Definition

Let P be a CN.

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

- If there is a constraint c involving a variable x such that there is no support for (x, a) on c, then (x, a) is **not GAC**.
- A **GAC algorithm** is an algorithm that removes all values from a CN P that are not GAC.
- A GAC algorithm computes the so-called **GAC-closure** of P by propagating constraints until a fixed-point is reached.
- A GAC algorithm is **generic** iff it can be applied to any CN (set of constraints).

(G)AC Algorithms

Algorithm	Time	Space	Grain	Author(s)
$AC3$	$O(ed^3)$	$-$	gros	(Mackworth, 1977)
$AC4$	$O(ed^2)$	$O(ed^2)$	fin	(Mohr & Henderson, 1986)
$AC6$	$O(ed^2)$	$O(ed)$	fin	(Bessiere, 1994)
$AC7$	$O(ed^2)$	$O(ed)$	fin	(Bessiere *et al.* , 1999)
$AC3_d$	$O(ed^3)$	$O(e + nd)$	gros	(van Dongen, 2002)
$AC2001/3.1$	$O(ed^2)$	$O(ed)$	gros	(Bessiere *et al.* , 2005)
$AC3.2/3.3$	$O(ed^2)$	$O(ed)$	gros	(Lecoutre *et al.* , 2003)
$AC3^{rm}$	$O(ed^2/ed^3)$	$O(ed)$	gros	(Lecoutre & Hemery, 2007)
$AC3^{bit(+rm)}$	$O(ed^3)$	$-$	gros	(Lecoutre & Vion, 2008)

Complexities for binary CNs
(e: number of constraints, d: greatest domain size, n: number of variables)

Establishing Arc Consistency on Domino instances

Instances		AC2001	AC3	$AC3^{rm}$	$AC3^{bit}$	$AC3^{bit+rm}$
800-800	CPU	48.4	$2,437$	34.5	13.4	**8.7**
	mem	$49M$	$33M$	$41M$	$33M$	$33M$
1000-1000	CPU	89.5	$5,911$	62.4	25.1	**14.3**
	mem	$66M$	$42M$	$54M$	$42M$	$46M$
2000-2000	CPU	678	$> 5h$	443	289	**91**
	mem	$210M$		$156M$	$117M$	$132M$
3000-3000	CPU	$2,349$	$> 5h$	$1,564$	$1,274$	**278**
	mem	$454M$		$322M$	$240M$	$275M$

Results on instances domino-n-d
(n variables, domain size d).

Outline

1. Modelling Constraint Problems
2. Solving Constraint Satisfaction Problems
3. Constraint Propagation
4. Filtering Algorithms for Table, MDD and Regular Constraints
5. Strong Inference

GAC Algorithms for Table Constraints

A table constraint is a a constraint defined in extension. Is is said to be:

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

Many schemes/algorithms proposed in the literature:

- GAC-valid: iterating the list of valid tuples
- GAC-allowed: iterating the list of allowed tuples (Bessiere & Régin, 1997)
- GAC-valid+allowed: visiting both lists (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)
- STR (Ullmann, 2007; Lecoutre, 2008)

GAC Algorithms for Table Constraints

A table constraint is a a constraint defined in extension. Is is said to be:

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

Many schemes/algorithms proposed in the literature:

- GAC-valid: iterating the list of valid tuples
- GAC-allowed: iterating the list of allowed tuples (Bessiere & Régin, 1997)
- GAC-valid+allowed: visiting both lists (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)
- STR (Ullmann, 2007; Lecoutre, 2008)

An Illustrative Table Constraint

A constraint c such that:

- $scp(c) = \{x_1, x_2, x_3, x_4, x_5\}$
- c is positive

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-allowed

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples	
(0,0,0,0,0)	X
(0,0,0,0,1)	
(0,0,0,1,0)	
(0,0,0,1,1)	
(0,0,1,0,0)	
(0,0,1,0,1)	
(0,0,1,1,0)	
(0,0,1,1,1)	
(0,1,0,0,0)	
(0,1,0,0,1)	
(0,1,0,1,0)	
(0,1,0,1,1)	
(0,1,1,0,0)	
(0,1,1,0,1)	
(0,1,1,1,0)	
(2,2,2,2,2)	

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples	
(0,0,0,0,0)	X
(0,0,0,0,1)	
(0,0,0,1,0)	
(0,0,0,1,1)	
(0,0,1,0,0)	
(0,0,1,0,1)	
(0,0,1,1,0)	
(0,0,1,1,1)	
(0,1,0,0,0)	
(0,1,0,0,1)	
(0,1,0,1,0)	
(0,1,0,1,1)	
(0,1,1,0,0)	
(0,1,1,0,1)	
(0,1,1,1,0)	
(2,2,2,2,2)	

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples	
(0,0,0,0,0)	X
(0,0,0,0,1)	X
(0,0,0,1,0)	
(0,0,0,1,1)	
(0,0,1,0,0)	
(0,0,1,0,1)	
(0,0,1,1,0)	
(0,0,1,1,1)	
(0,1,0,0,0)	
(0,1,0,0,1)	
(0,1,0,1,0)	
(0,1,0,1,1)	
(0,1,1,0,0)	
(0,1,1,0,1)	
(0,1,1,1,0)	
(2,2,2,2,2)	

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples	
(0,0,0,0,0)	X
(0,0,0,0,1)	X
(0,0,0,1,0)	
(0,0,0,1,1)	
(0,0,1,0,0)	
(0,0,1,0,1)	
(0,0,1,1,0)	
(0,0,1,1,1)	
(0,1,0,0,0)	
(0,1,0,0,1)	
(0,1,0,1,0)	
(0,1,0,1,1)	
(0,1,1,0,0)	
(0,1,1,0,1)	
(0,1,1,1,0)	
(2,2,2,2,2)	

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples	
(0,0,0,0,0)	X
(0,0,0,0,1)	X
(0,0,0,1,0)	X
(0,0,0,1,1)	
(0,0,1,0,0)	
(0,0,1,0,1)	
(0,0,1,1,0)	
(0,0,1,1,1)	
(0,1,0,0,0)	
(0,1,0,0,1)	
(0,1,0,1,0)	
(0,1,0,1,1)	
(0,1,1,0,0)	
(0,1,1,0,1)	
(0,1,1,1,0)	
(2,2,2,2,2)	

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples	
(0,0,0,0,0)	X
(0,0,0,0,1)	X
(0,0,0,1,0)	X
(0,0,0,1,1)	X
(0,0,1,0,0)	X
(0,0,1,0,1)	X
(0,0,1,1,0)	X
(0,0,1,1,1)	X
(0,1,0,0,0)	X
(0,1,0,0,1)	X
(0,1,0,1,0)	X
(0,1,0,1,1)	X
(0,1,1,0,0)	X
(0,1,1,0,1)	X
(0,1,1,1,0)	
(2,2,2,2,2)	

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples	
(0,0,0,0,0)	X
(0,0,0,0,1)	X
(0,0,0,1,0)	X
(0,0,0,1,1)	X
(0,0,1,0,0)	X
(0,0,1,0,1)	X
(0,0,1,1,0)	X
(0,0,1,1,1)	X
(0,1,0,0,0)	X
(0,1,0,0,1)	X
(0,1,0,1,0)	X
(0,1,0,1,1)	X
(0,1,1,0,0)	X
(0,1,1,0,1)	X
(0,1,1,1,0)	
(2,2,2,2,2)	

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Illustration of GAC-allowed

Allowed Tuples	
(0,0,0,0,0)	X
(0,0,0,0,1)	X
(0,0,0,1,0)	X
(0,0,0,1,1)	X
(0,0,1,0,0)	X
(0,0,1,0,1)	X
(0,0,1,1,0)	X
(0,0,1,1,1)	X
(0,1,0,0,0)	X
(0,1,0,0,1)	X
(0,1,0,1,0)	X
(0,1,0,1,1)	X
(0,1,1,0,0)	X
(0,1,1,0,1)	X
(0,1,1,1,0)	X
(2,2,2,2,2)	

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

$\Rightarrow 2^r - 1$ operations (validity checks)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples
(0,1,1,1,1)

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples
(0,1,1,1,1)

X

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples
(0,1,1,1,1) X
(0,1,1,1,2)

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples	
(0,1,1,1,1)	X
(0,1,1,1,2)	X

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples	
(0,1,1,1,1)	X
(0,1,1,1,2)	X
(0,1,1,2,1)	

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples	
(0,1,1,1,1)	X
(0,1,1,1,2)	X
(0,1,1,2,1)	X

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples	
(0,1,1,1,1)	X
(0,1,1,1,2)	X
(0,1,1,2,1)	X
(0,1,1,2,2)	X
(0,1,2,1,1)	X
(0,1,2,1,2)	X
(0,1,2,2,1)	X
(0,1,2,2,2)	X
(0,2,1,1,1)	X
(0,2,1,1,2)	X
(0,2,1,2,1)	X
(0,2,1,2,2)	X
(0,2,2,1,1)	X
(0,2,2,1,2)	X
(0,2,2,2,1)	X

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples	
(0,1,1,1,1)	X
(0,1,1,1,2)	X
(0,1,1,2,1)	X
(0,1,1,2,2)	X
(0,1,2,1,1)	X
(0,1,2,1,2)	X
(0,1,2,2,1)	X
(0,1,2,2,2)	X
(0,2,1,1,1)	X
(0,2,1,1,2)	X
(0,2,1,2,1)	X
(0,2,1,2,2)	X
(0,2,2,1,1)	X
(0,2,2,1,2)	X
(0,2,2,2,1)	X
(0,2,2,2,2)	

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
(2,2,2,2,2)

Illustration of GAC-valid

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

Is there a support for $(x_1, 0)$ on c?

Valid Tuples		Allowed Tuples
(0,1,1,1,1)	X	(0,0,0,0,0)
(0,1,1,1,2)	X	(0,0,0,0,1)
(0,1,1,2,1)	X	(0,0,0,1,0)
(0,1,1,2,2)	X	(0,0,0,1,1)
(0,1,2,1,1)	X	(0,0,1,0,0)
(0,1,2,1,2)	X	(0,0,1,0,1)
(0,1,2,2,1)	X	(0,0,1,1,0)
(0,1,2,2,2)	X	(0,0,1,1,1)
(0,2,1,1,1)	X	(0,1,0,0,0)
(0,2,1,1,2)	X	(0,1,0,0,1)
(0,2,1,2,1)	X	(0,1,0,1,0)
(0,2,1,2,2)	X	(0,1,0,1,1)
(0,2,2,1,1)	X	(0,1,1,0,0)
(0,2,2,1,2)	X	(0,1,1,0,1)
(0,2,2,2,1)	X	(0,1,1,1,0)
(0,2,2,2,2)	X	(2,2,2,2,2)

$\Rightarrow 2^r$ operations (constraint checks)

GAC-valid+allowed (Algorithm)

At the heart of the algorithm, we have the procedure:

Algorithm 3: seekSupportGACva(c: Constraint, x: Variable, a: Value) : Tuple

$\tau \leftarrow \mathit{setFirstValidTuple}(c, x, a)$
while $\tau \neq \top$ **do**
  $\tau' \leftarrow \mathit{binarySearch}(\mathit{allowedTuples}(c, x, a), \tau)$
  if $\tau' = \top$ **then** **return** $\top$
  $j \leftarrow \mathit{seekInvalidPosition}(c, \tau')$
  if $j = NO$ **then** **return** τ'
  $\tau \leftarrow \mathit{setNextValid}(c, x, a, \tau', j)$
return $\top$

Illustration of GAC-valid+allowed

Valid Tuples

Allowed Tuples
...
...
...
...
$\mathcal{T}_2$
...
...
...
...
...
...
$\mathcal{T}_4$
...
...
...
...

Illustration of GAC-valid+allowed

Valid Tuples
τ_1

Allowed Tuples
...
...
...
...
τ_2
...
...
...
...
...
...
τ_4
...
...
...
...

Illustration of GAC-valid+allowed

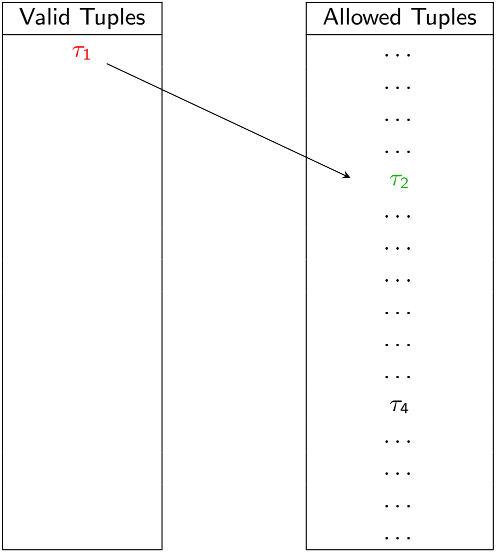


Illustration of GAC-valid+allowed

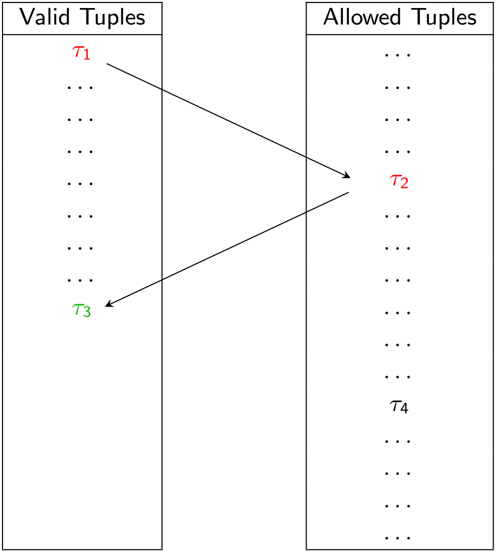


Illustration of GAC-valid+allowed

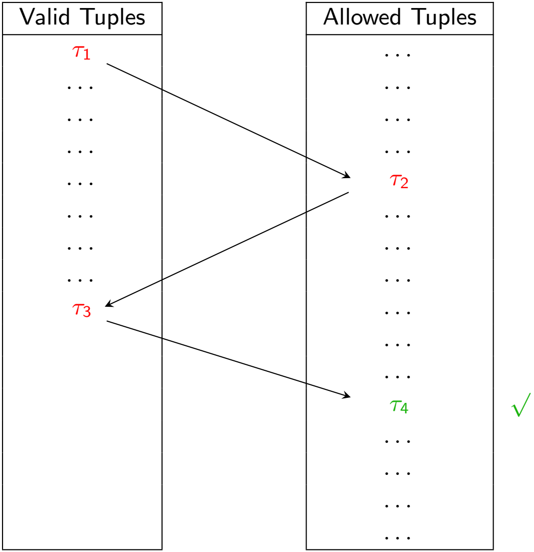


Illustration of GAC-valid+allowed

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

A support for $(x_1, 0)$ on c?

Valid Tuples

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
nil

Illustration of GAC-valid+allowed

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

A support for $(x_1, 0)$ on c?

Valid Tuples
(0,1,1,1,1)

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
nil

Illustration of GAC-valid+allowed

The current domains:

- $dom(x_1) = \{0\}$
- $dom(x_2) = \{1, 2\}$
- $dom(x_3) = \{1, 2\}$
- $dom(x_4) = \{1, 2\}$
- $dom(x_5) = \{1, 2\}$

A support for $(x_1, 0)$ on c?

Valid Tuples
(0,1,1,1,1)

Allowed Tuples
(0,0,0,0,0)
(0,0,0,0,1)
(0,0,0,1,0)
(0,0,0,1,1)
(0,0,1,0,0)
(0,0,1,0,1)
(0,0,1,1,0)
(0,0,1,1,1)
(0,1,0,0,0)
(0,1,0,0,1)
(0,1,0,1,0)
(0,1,0,1,1)
(0,1,1,0,0)
(0,1,1,0,1)
(0,1,1,1,0)
nil

$\Rightarrow$ 1 operation (constraint check)

Observations

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

- applying GAC3v is $O(2^{r-1})$.
- applying GAC3a is $O(2^{r-1})$.
- applying GAC3va is $O(r^2)$

However, the previous schemes proceed **gradually**: a support is sought for each value in turn: $(x_1, 0)$, $(x_2, 1)$, $(x_2, 2)$, ...

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

- STR
- MDD

Simple Tabular Reduction

Simple tabular reduction (STR)

- original approach introduced by J. Ullmann
- principle: to dynamically maintain 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)

Algorithm 4: STR(c: constraint): set of variables

Output: the set of variables in $scp(c)$ with reduced domain

```
foreach variable x ∈ scp(c) do
    gacValues[x] ← ∅
foreach tuple τ ∈ table[c] do
    if isValid(c, τ) then
        foreach variable x ∈ scp(c) do
            if τ[x] ∉ gacValues[x] then
                add τ[x] to gacValues[x]
    else
        removeTuple(c, τ)
// domains are now updated and X_evt computed

X_evt ← ∅
foreach variable x ∈ scp(c) do
    if gacValues[x] ⊂ dom(x) then
        dom(x) ← gacValues[x]
        X_evt ← X_evt ∪ {x}
return X_evt
```

Illustration with 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 with 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 with STR

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

$table[c_{xyz}]$

$x\ y\ z$

(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 with 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

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

Illustration with STR

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

$table[c_{xyz}]$

$x\ y\ z$

(a,a,c) $\checkmark$
~~(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 with 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	$\cancel{\text{a}}$
b	$\cancel{\text{b}}$	b
c	c	c

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

Illustration with STR

$table[c_{xyz}]$

$x \; y \; z$

(a,a,c) $\surd$
~~(a,b,a)~~
(a,c,b) $\surd$
~~(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 with STR

$table[c_{xyz}]$

$x\ y\ z$

(a,a,c) $\surd$
~~(a,b,a)~~
(a,c,b) $\surd$
~~(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 with 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

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

Illustration with 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

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

Illustration with STR

$dom(x)$ $dom(y)$ $dom(z)$

$dom(x)$	$dom(y)$	$dom(z)$
a	a	$\cancel{\text{a}}$
$\cancel{\text{b}}$	$\cancel{\text{b}}$	b
c	c	c

$table[c_{xyz}]$

$x\ y\ z$

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

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

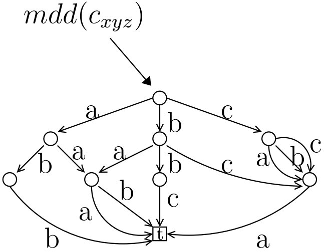
A Table Constraint as a MDD Constraint

$table[c_{xyz}]$

	$x\ y\ z$
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)

(a) A table

level	
1	x
2	y
3	z
4	



(b) A MDD

Algorithm 5: enforceGAC-mdd(c: constraint): set of variables

Output: the set of variables in $scp(c)$ with reduced domain

$\Sigma^{true} \leftarrow \emptyset$
$\Sigma^{false} \leftarrow \emptyset$
foreach *variable* $x \in scp(c)$ **do**
    $gacValues[x] \leftarrow \emptyset$
exploreMDD($mdd(c)$)   `// ` *gacValues* `is updated during exploration`
`// domains are now updated and` X_{evt} `computed`

$X_{evt} \leftarrow \emptyset$
foreach *variable* $x \in scp(c)$ **do**
    if $gacValues[x] \subset dom(x)$ **then**
        $dom(x) \leftarrow gacValues[x]$
        $X_{evt} \leftarrow X_{evt} \cup \{x\}$
return X_{evt}

Algorithm 6: exploreMDD(*node*: Node): Boolean

Output: *true* iff *node* is supported

```
if node = [t] then
    return true                       // since we are at a leaf
if node ∈ Σ^true then
    return true                       // since already proved to be supported
if node ∈ Σ^false then
    return false                      // since already proved to be unsupported
x ← node.variable ; supported ← false
foreach arc ∈ node.outs do
    if arc.value ∈ dom(x) then
        if exploreMDD(arc.destination) then
            supported ← true
            gacValues[x] ← gacValues[x] ∪ {arc.value}
if supported = true then  Σ^true ← Σ^true ∪ {node}
else  Σ^false ← Σ^false ∪ {node}
return supported
```

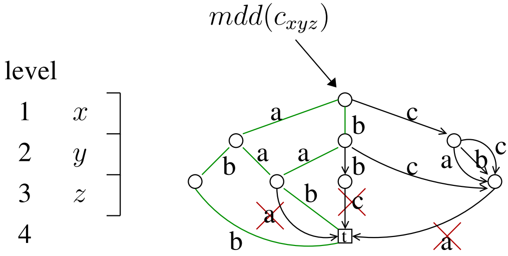
Illustration with MDD

Event: $z = b$

Domains before filtering:
$dom(x) \leftarrow \{a, b, c\}$
$dom(y) \leftarrow \{a, b, c\}$
$dom(z) \leftarrow \{b\}$

Collected values:
$gacValues[x] \leftarrow \{a, b\}$
$gacValues[y] \leftarrow \{a, b\}$
$gacValues[z] \leftarrow \{b\}$

Domains after filtering:
$dom(x) \leftarrow \{a, b\}$
$dom(y) \leftarrow \{a, b\}$
$dom(z) \leftarrow \{b\}$



The Regular Constraint

The general form of a regular constraint (Pesant, 2004) is regular(X, A) where:

- X denotes the scope of the constraint (an ordered set of variables)
- A denotes a deterministic finite automata

An instantiation I of X satisfies the constraint iff the word formed by the sequence of values in I is recognized by the automata A.

Remark

The constraint regular is a generalization of the stretch constraint.

The Stretch Constraint

The general form of a stretch constraint (Pesant, 2001) is stretch(X, L, U, P) where:

- X denotes the scope of the constraint (an ordered set of variables)
- L and U are mappings from $\cup_{x \in X} dom(x)$ to $\mathbb{N}$
- P is a set of pairs of distinct values chosen in $\cup_{x \in X} dom(x)$

An instantiation I of X satisfies the constraint iff

1. every stretch in I, with value v, has a length comprised between $L(v)$ and $U(v)$,
2. every two consecutive stretches in I form a pair of values contained in P.

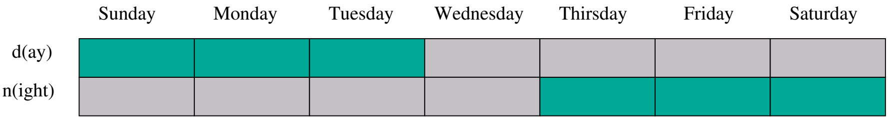
Remark

A stretch is a a maximal sequence of consecutive variables that take the same value.

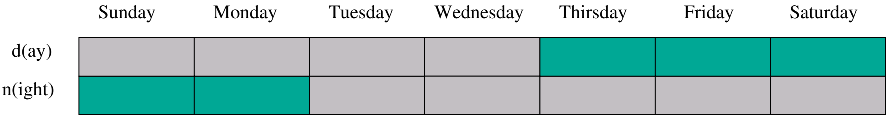
Example

We have a set X of variables for representing the successive shifts of an employee:

- $\forall x \in X, dom(x) = \{d, o, n\}$ // working d(ay), o(ff), n(ight)
- $\forall v \in \{d, o, n\}, L(v) = 2$ and $U(v) = 3$
- $P = \{(d, o), (o, d), (o, n), (n, o)\}$



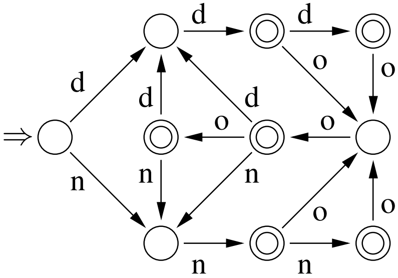
is not satisfying the stretch constraint



is satisfying the stretch constraint

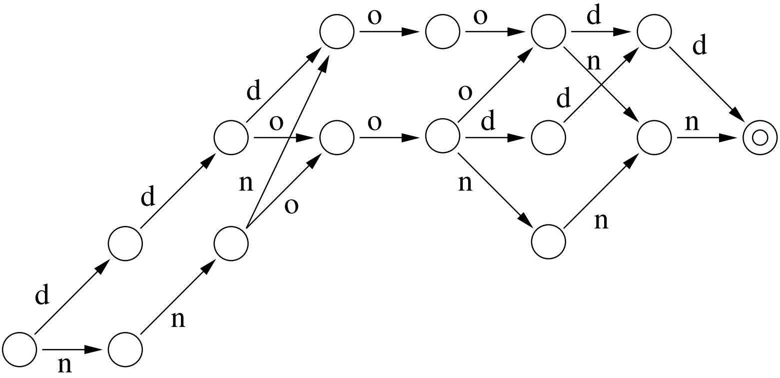
Example

Here is the automata for the stretch constraint introduced previously :



A Regular constraint as a MDD Constraint

Here is the MDD developed from the automata over a scope of 7 variables :



Converting the Stretch constraint into a MDD or Table constraint:

Scope	MDD	Table
7 variables	15 nodes	12 tuples
14 variables	58 nodes	176 tuples
28 variables	170 nodes	72,800 tuples
42 variables	282 nodes	? tuples

Regular for Nonogram Puzzles

				2	2	2	2	2	2	2	
			3	3	2	2	2	2	2	3	3
	2	2									
	4	4									
1	3	1									
2	1	2									
	1	1									
	2	2									
	2	2									
		3									
		1									

Table: Nonogram Puzzle to be solved (see Chapter 14 in Gecode Documentation)

				2	2	2	2	2	2	2	
			3	3	2	2	2	2	2	3	3
	2	2		■	■				■	■	
	4	4	■	■	■	■		■	■	■	■
1	3	1	■			■	■	■			■
2	1	2	■	■			■			■	■
	1	1		■						■	
	2	2		■	■				■	■	
	2	2			■	■		■	■		
		3				■	■	■			
		1					■				

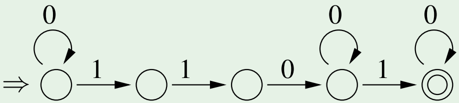
Table: Solution to the Nonogram Puzzle

Regular for Nonogram Puzzles

Each hint corresponds to a regular expression.

Example

The hint 2 1 corresponds to
$0^*1^20^+10^*$



When considering the instances of the benchmarks proposed by G. Pesant,

- tables are very large (over 1,000,000 tuples for some of them)
- MDDs are rather compact (a few hundreds of nodes, at most)

Table for Kakuro Puzzles

\	\	\	7 \	21 \	\	\	29 \	17 \	\	29 \	23 \
\	\	6 \ 4			\	16 \ 7			14 \ 16		
\	4 \ 6				3 \ 39						
\ 28								22 \ 24			
\ 3			10 \ 3			\ 20					\
\	\	11 \ 9			10 \	\	10 \ 4			24 \	16 \
\	7 \ 10					9 \ 3			23 \ 16		
\ 6				4 \ 42							
\ 21							\ 21				\
\ 4			\ 3			\	\ 16			\	\

Table: Kakuro puzzle to be solved (see Chapter 18 in Gecode Documentation)

Table for Kakuro Puzzles

			7\	21\			29\	17\		29\	23\
		6\4	1	3		16\7	8	9	14\16	7	9
	4\6	3	2	1	3\39	9	7	8	4	5	6
\28	3	1	4	6	2	7	5	22\24	7	9	8
\3	1	2	10\3	2	1	\20	9	1	2	8	
		11\9	4	5	10\		10\4	3	1	24\	16\
	7\10	2	1	4	3	9\3	1	2	23\16	7	9
\6	2	1	3	4\42	4	3	9	5	6	8	7
\21	4	5	2	3	1	6	\21	4	8	9	
\4	1	3	\3	1	2		\16	7	9		

Table: Solution to the Kakuro Puzzle

Table for Kakuro Puzzles

For a maximal sequence of variables X, we can post two distinct constraints:

- allDifferent(X)
- sum(X) = v (i.e., $\Sigma_{x \in X} = v$) where v is the value of the hint

and we can benefit from sophisticated filtering algorithms for these constraints.

However, we deal with separate constraints sharing the same scope.

One solution (Simonis, 2008) is to build table constraints by computing solutions to pairs of constraints "allDifferent-sum". In the worst-case, $362,880$ tuples (but far less, most of the time)

Summary

Table constraints:

- universal representation (but space complexity to be considered)
- simple solution to end-users of CP systems

MDD constraints:

- compact representation
- can be derived from automata

What about decomposition approaches of automata-based constraints (Beldiceanu *et al.* , 2005)?

Outline

1. Modelling Constraint Problems
2. Solving Constraint Satisfaction Problems
3. Constraint Propagation
4. Filtering Algorithms for Table, MDD and Regular Constraints
5. Strong Inference

Filtering through Consistencies

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

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

- Generalized Arc Consistency (GAC)
- Path Inverse Consistency (PIC)
- Singleton Arc Consistency (SAC)

A *second-order consistency* 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

Filtering through Consistencies

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

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

- Generalized Arc Consistency (GAC)
- Path Inverse Consistency (PIC)
- Singleton Arc Consistency (SAC)

A *second-order consistency* 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

Filtering through Consistencies

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

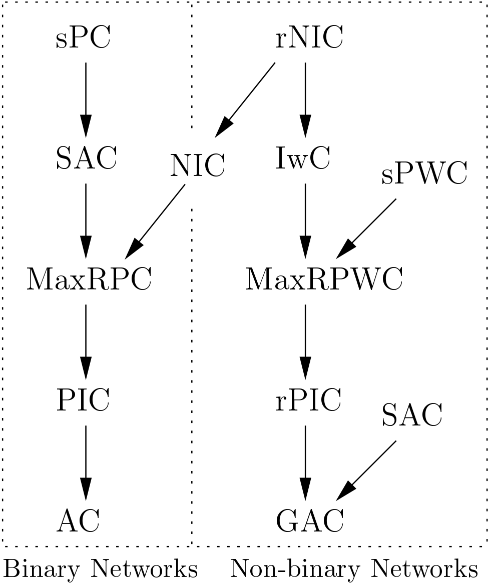
A *first-order consistency* (or domain-filtering consistency) allows us to identify inconsistent values (nogoods of size 1). For example:

- Generalized Arc Consistency (GAC)
- Path Inverse Consistency (PIC)
- Singleton Arc Consistency (SAC)

A *second-order consistency* allows us to identify inconsistent pairs of values (nogoods of size 2). For example:

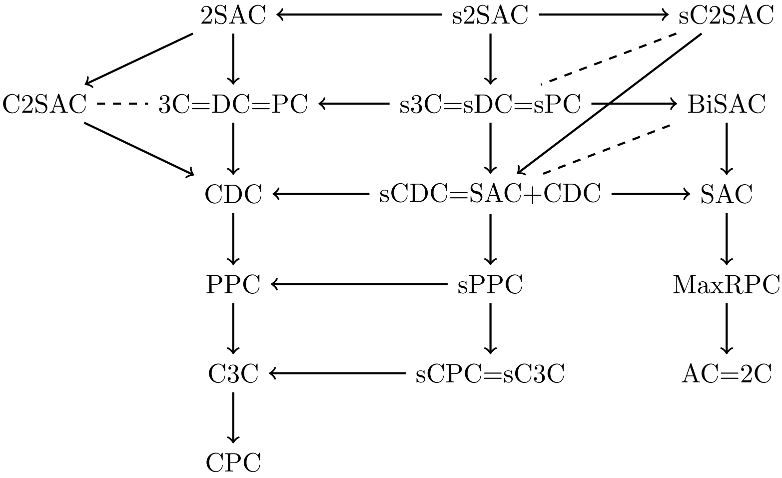
- Path Consistency (PC)
- Dual Consistency (DC)
- Conservative variants of PC and DC

Relationships between first-order Consistencies



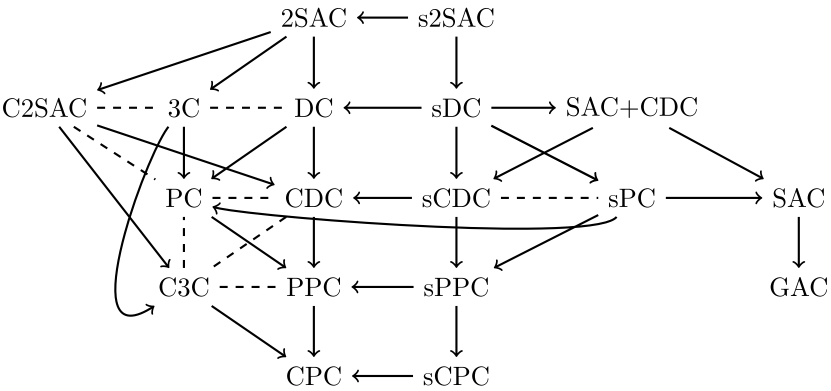
$\phi \longrightarrow \psi$
means
ϕ is strictly stronger than ψ

Relationships between 2-order Consistencies (binary CNs)



⟶ strictly stronger   - - - incomparable   = equivalent

Relationships between 2-order Consistencies (non-binary)



$\longrightarrow$ strictly stronger $\quad$ - - - incomparable $\quad$ = equivalent

A focus on SAC

Definition (Singleton Arc Consistency)

Let P be a CN.

- A value (x, a) of P is singleton arc-consistent (SAC) iff $AC(P|_{x=a}) \neq \bot$.
- A variable x of P is SAC iff $\forall a \in dom(x)$, (x, a) is SAC.
- P is SAC iff any variable of P is SAC.

Remark

SAC is stronger than (G)AC

A focus on SAC

Definition (Singleton Arc Consistency)

Let P be a CN.

- A value (x, a) of P is singleton arc-consistent (SAC) iff $AC(P|_{x=a}) \neq \bot$.
- A variable x of P is SAC iff $\forall a \in dom(x)$, (x, a) is SAC.
- P is SAC iff any variable of P is SAC.

Remark

SAC is stronger than (G)AC

A focus on SAC

Definition (Singleton Arc Consistency)

Let P be a CN.

- A value (x, a) of P is singleton arc-consistent (SAC) iff $AC(P|_{x=a}) \neq \bot$.
- A variable x of P is SAC iff $\forall a \in dom(x)$, (x, a) is SAC.
- P is SAC iff any variable of P is SAC.

Remark

SAC is stronger than (G)AC

A focus on SAC

Definition (Singleton Arc Consistency)

Let P be a CN.

- A value (x, a) of P is singleton arc-consistent (SAC) iff $AC(P|_{x=a}) \neq \bot$.
- A variable x of P is SAC iff $\forall a \in dom(x)$, (x, a) is SAC.
- P is SAC iff any variable of P is SAC.

Remark

SAC is stronger than (G)AC

A focus on SAC

Definition (Singleton Arc Consistency)

Let P be a CN.

- A value (x, a) of P is singleton arc-consistent (SAC) iff $AC(P|_{x=a}) \neq \perp$.
- A variable x of P is SAC iff $\forall a \in dom(x)$, (x, a) is SAC.
- P is SAC iff any variable of P is SAC.

Remark

SAC is stronger than (G)AC

A focus on SAC

Definition (Singleton Arc Consistency)

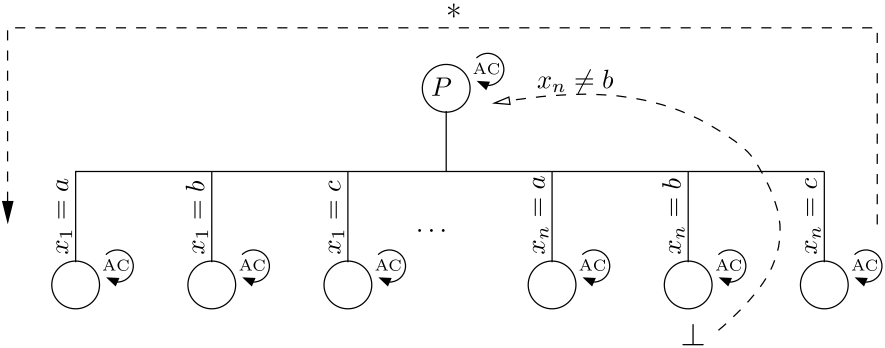
Let P be a CN.

- A value (x, a) of P is singleton arc-consistent (SAC) iff $AC(P|_{x=a}) \neq \bot$.
- A variable x of P is SAC iff $\forall a \in dom(x)$, (x, a) is SAC.
- P is SAC iff any variable of P is SAC.

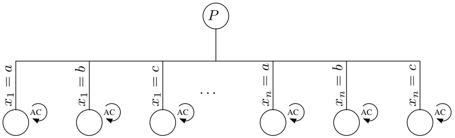
Remark

SAC is stronger than (G)AC

Algorithm SAC-1

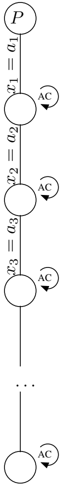


Exploiting Incrementality of GAC Algorithms

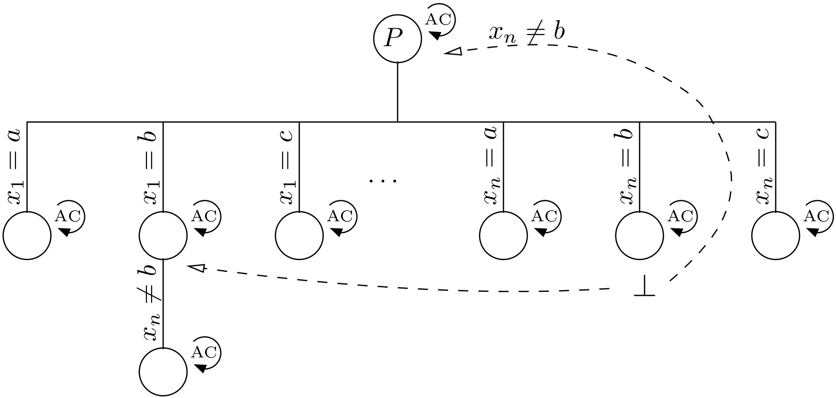


The complexity of enforcing AC on a node is $O(ed^2)$.

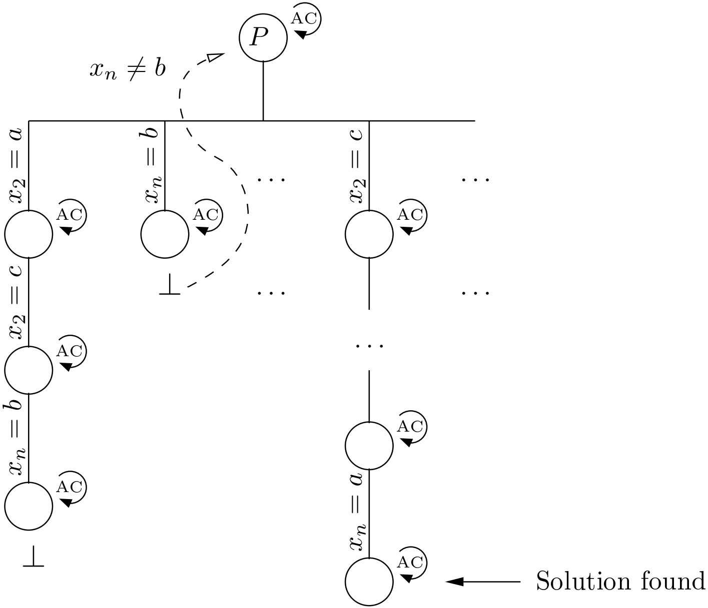
The complexity of enforcing AC on the branch is $O(ed^2)$.



Algorithms SAC-opt and SAC-SDS



Algorithms SAC-3



(Worst-case) Complexities

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)

Some Experimental Results

		SAC-1	SAC-SDS	SAC-3	SAC-3+
cc-20-3	CPU	23	22	7	7
(#×=0)	#scks	1,200	1,200	1,200	1,200
gr-34-9	CPU	111	31	91	32
(#×=513)	#scks	8,474	4,720	11,017	2,013
qa-6	CPU	27	14	8.4	4.3
(#×=48)	#scks	2,523	1,702	2,855	1,448
*scen*05	CPU	11	20	1.5 (1)	1.8
(#×=13814)	#scks	6,513	4,865	4,241	2,389
*graph*03	CPU	215	136	74	39
(#×=1274)	#scks	20,075	17,069	22,279	8,406

A focus on DC

Definition (Dual Consistency - DC)

Let P be a constraint network.

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

A focus on DC

Definition (Dual Consistency - DC)

Let P be a constraint network.

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

A focus on DC

Definition (Dual Consistency - DC)

Let P be a constraint network.

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

A focus on DC

Definition (Dual Consistency - DC)

Let P be a constraint network.

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

A focus on DC

Definition (Dual Consistency - DC)

Let P be a constraint network.

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

- $GAC \circ DC(P) = sDC(P)$
- $GAC \circ CDC(P) = sCDC(P)$

But

- $AC \circ CPC(P) \neq sCPC(P)$
- $AC \circ PPC(P) \neq sPPC(P)$.

$s\phi$ is $\phi + (G)AC$

sCDC1

Algorithm 7: sCDC1

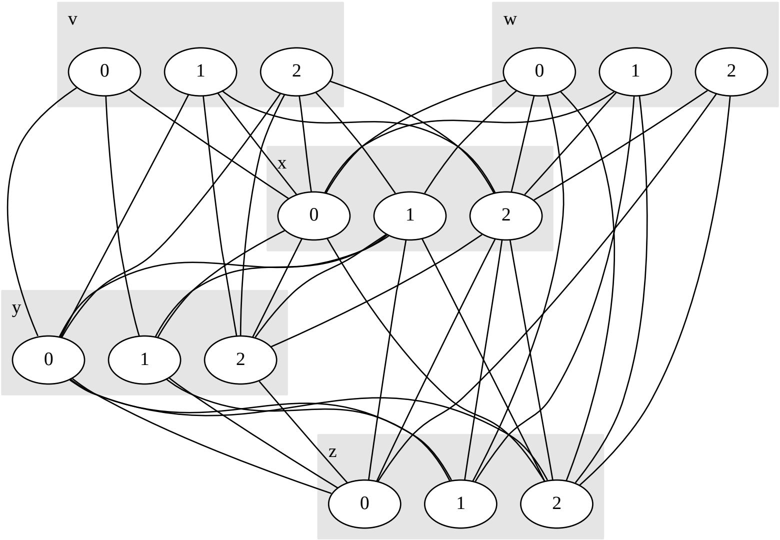
$P \leftarrow GAC(P)$ `// GAC is initially enforced`
finished $\leftarrow$ *false*
repeat
    finished $\leftarrow$ *true*
    foreach $x \in vars(P)$ **do**
        if *revise-sCDC*1(x) **then**
            $P \leftarrow GAC(P)$ `// GAC is maintained`
            finished $\leftarrow$ *false*
until *finished*

sCDC1

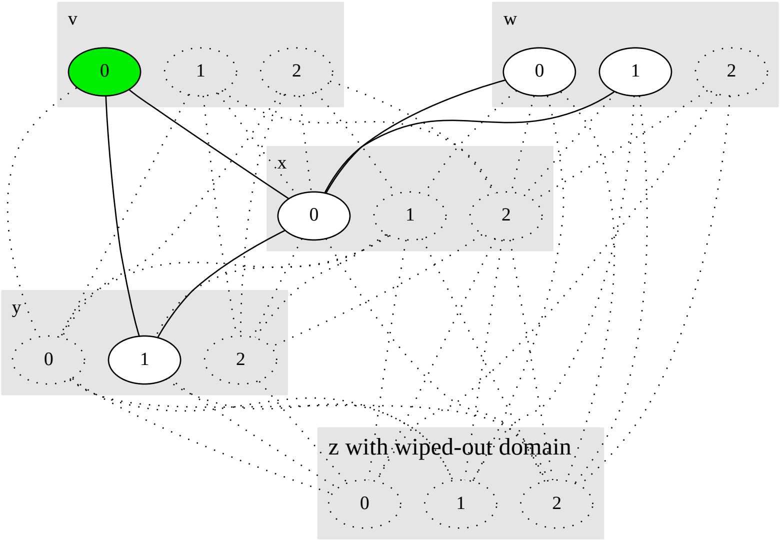
Algorithm 8: revise-sCDC1(**var** x: variable): Boolean

```
modified ← false
foreach value a ∈ dom(x) do
    P′ ← GAC(P|x=a)                              // Singleton check on (x, a)
    if P′ = ⊥ then
        remove a from dom(x)                     // SAC-inconsistent value
        modified ← true
    else
        foreach constraint cxy ∈ cons(P) do
            foreach value b ∈ dom(y) do
                if b ∉ dom^P′(y) then
                    remove (a, b) from rel(cxy)  // CDC-inconsistent values
                    modified ← true
return modified
```

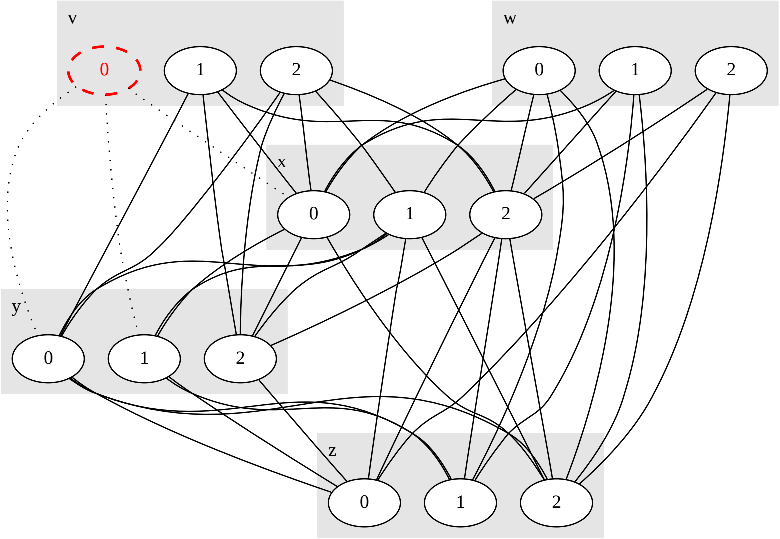
Example



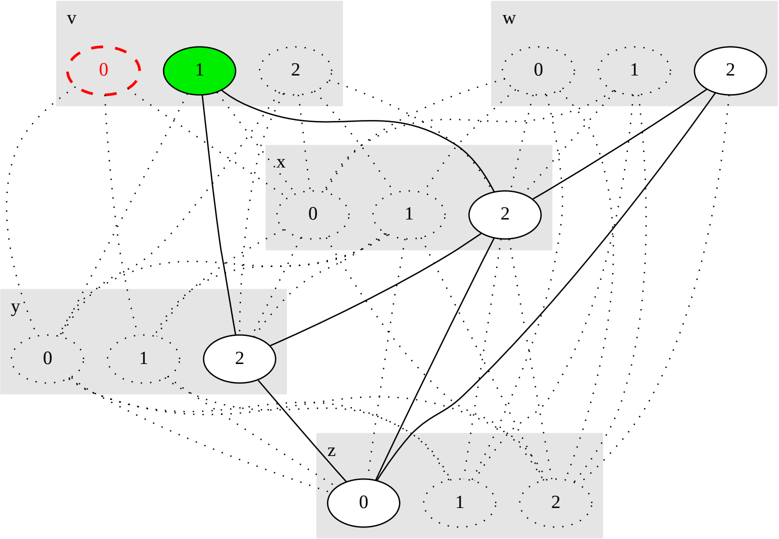
Example



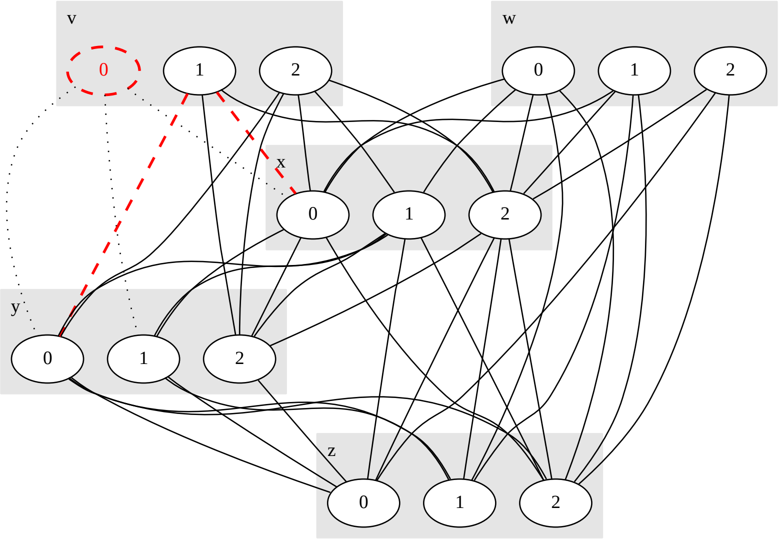
Example



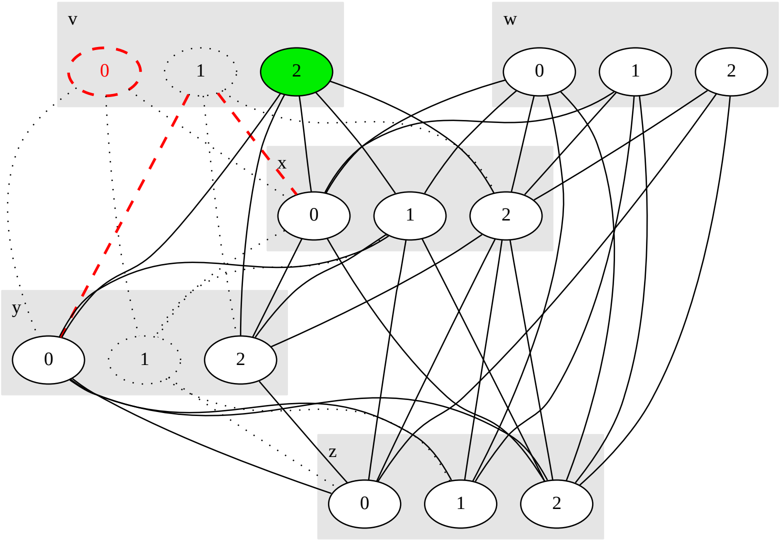
Example



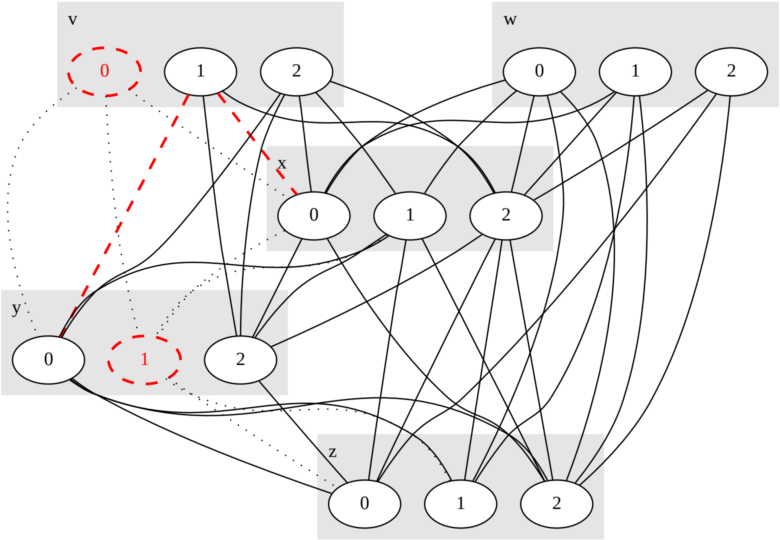
Example



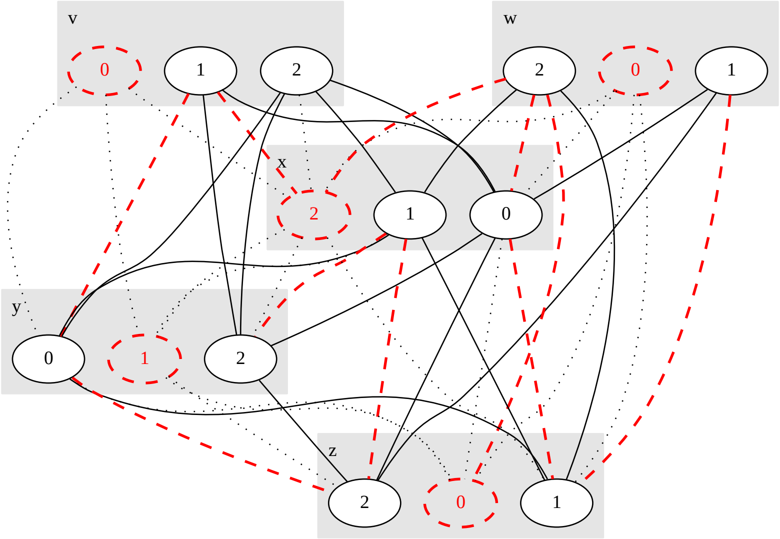
Example



Example



Example



Impact for Search

Instance		MAC	sCDC1-MAC
scen11-f8	CPU	8.0	14.3
	nodes	$14,068$	$4,946$
scen11-f6	CPU	68.4	58.2
	nodes	$302K$	$145K$
scen11-f4	CPU	582	559
	nodes	$2,826K$	$1,834K$
scen11-f3	CPU	$2,338$	$1,725$
	nodes	$12M$	$5,863K$
scen11-f2	CPU	$7,521$	$5,872$
	nodes	$37M$	$21M$
scen11-f1	CPU	$17,409$	$13,136$
	nodes	$93M$	$55M$

SAC and DC as Decision-based Consistencies

$$
\begin{tikzcd}
 & S^\phi_\Delta + E^\phi_\Delta \arrow[r] \arrow[dr] & E^\phi_\Delta & \\
S^\phi_\Delta + D^\phi_\Delta \arrow[r] \arrow[dr] & B^\phi_\Delta \arrow[u, dashed] \arrow[r] & S^\phi_\Delta \arrow[r, dotted] & \phi \\
 & D^\phi_\Delta & &
\end{tikzcd}
$$

Figure: Relationships between general classes of consistencies.

$$
\begin{tikzcd}
 & \mathit{1}\text{-}AC \arrow[r] \arrow[dr] & E^{AC}_{\Delta^=} & & \\
sDC \arrow[r] \arrow[dr] & BiSAC \arrow[u] \arrow[r] & SAC \arrow[r] & BoundSAC \arrow[r] & AC \\
 & DC & & &
\end{tikzcd}
$$

Figure: Relationships when $\phi = AC$ and $\Delta = \Delta^=$.

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

Beldiceanu, N., Carlsson, M., Debruyne, R., & Petit, T. 2005.
Reformulation of Global Constraints Based on Constraint Checkers.
Constraints, **10**(4), 339–362.

Bessiere, C. 1994.
Arc consistency and arc consistency again.
Artificial Intelligence, **65**, 179–190.

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. 1997.
Arc consistency for general constraint networks: preliminary results.
Pages 398–404 of: Proceedings of IJCAI'97.

Bessiere, C., Freuder, E.C., & Régin, J. 1999.
Using constraint metaknowledge to reduce arc consistency computation.
Artificial Intelligence, **107**, 125–148.

Bessiere, C., Régin, J.C., Yap, R., & Zhang, Y. 2005.
An optimal coarse-grained arc consistency algorithm.
Artificial Intelligence, **165**(2), 165–185.

Cheng, K., & Yap, R. 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.

Gent, I.P., 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, G., & Walsh, T. 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., & Hemery, F. 2007.
A study of residual supports in Arc Consistency.
Pages 125–130 of: Proceedings of IJCAI'07.

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

Lecoutre, C., & Vion, J. 2008.
Enforcing Arc Consistency using Bitwise Operations.
Constraint Programming Letters, **2**, 21–35.

Lecoutre, C., Boussemart, F., & Hemery, F. 2003.
Exploiting multidirectionality in coarse-grained arc consistency algorithms.
Pages 480–494 of: Proceedings of CP'03.

Lecoutre, C., Likitvivatanavong, C., & Yap, R. 2012.
A path-optimal GAC algorithm for table constraints.
Page to appear of: Proceedings of ECAI'12.

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

Mackworth, A.K. 1977.
Consistency in networks of relations.
Artificial Intelligence, **8**(1), 99–118.

Mohr, R., & Henderson, T.C. 1986.
Arc and path consistency revisited.
Artificial Intelligence, **28**, 225–233.

Pesant, G. 2001.
A Filtering Algorithm for the Stretch Constraint.
Pages 183–195 of: Proceedings of CP'01.

Pesant, G. 2004.
A Regular Language Membership Constraint for Finite Sequences of Variables.
Pages 482–495 of: Proceedings of CP'04.

Régin, J.C. 1994.
A filtering algorithm for constraints of difference in CSPs.
Pages 362–367 of: Proceedings of AAAI'94.

Simonis, H. 2008.
Kakuro as a constraint problem.
In: Proceedings of the workshop on modelling and reformulating constraint satisfaction problems held with CP'08.

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

van Dongen, M.R.C. 2002.
$AC3_d$ an efficient arc consistency algorithm with a low space complexity.
Pages 755–760 of: Proceedings of CP'02.