Constraint Programming
– Searching : Part 2 –

Christophe Lecoutre
lecoutre@cril.fr

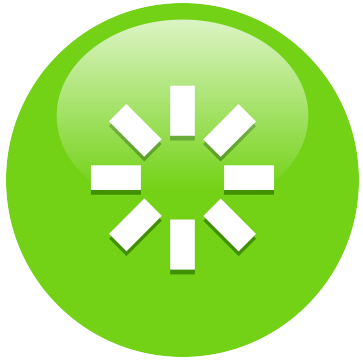
CRIL-CNRS UMR 8188
Universite d'Artois
Lens, France

January 2021

Outline

Outline

Restarts



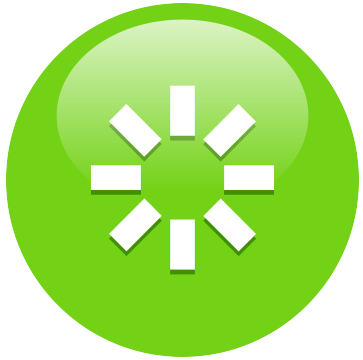
Remember that the first decisions are very important.

- Why? Because it can be very long to discard bad decisions performed at the top of the search tree.
- What to do? Regularly restarting search from the root node.

Restarting:

- avoids being stuck in large unsatisfiable portions of the search space
- brings diversification
- permits to test various heuristics
- permits to collect information (for adaptive heuristics) from run to run

Restarts



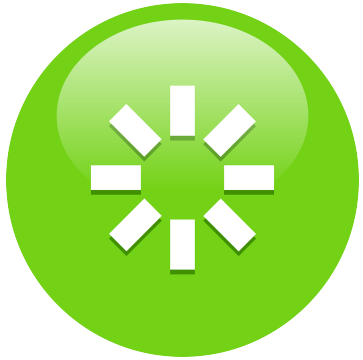
Remember that the first decisions are very important.

- Why? Because it can be very long to discard bad decisions performed at the top of the search tree.
- What to do? Regularly restarting search from the root node.

Restarting:

- avoids being stuck in large unsatisfiable portions of the search space
- brings diversification
- permits to test various heuristics
- permits to collect information (for adaptive heuristics) from run to run

Restarts



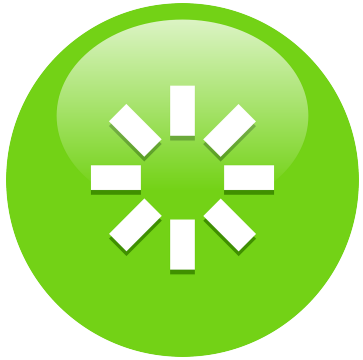
Remember that the first decisions are very important.

- Why? Because it can be very long to discard bad decisions performed at the top of the search tree.
- What to do? Regularly restarting search from the root node.

Restarting:

- avoids being stuck in large unsatisfiable portions of the search space
- brings diversification
- permits to test various heuristics
- permits to collect information (for adaptive heuristics) from run to run

Restarts



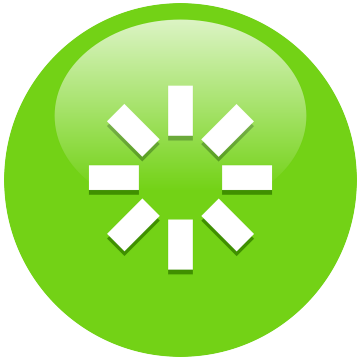
Remember that the first decisions are very important.

- Why? Because it can be very long to discard bad decisions performed at the top of the search tree.
- What to do? Regularly restarting search from the root node.

Restarting:

- avoids being stuck in large unsatisfiable portions of the search space
- brings diversification
- permits to test various heuristics
- permits to collect information (for adaptive heuristics) from run to run

Restarts



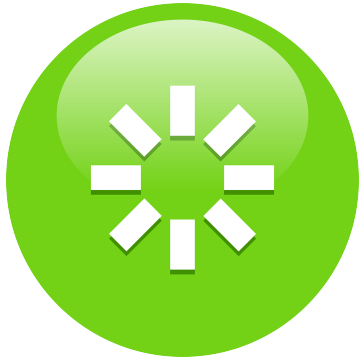
Remember that the first decisions are very important.

- Why? Because it can be very long to discard bad decisions performed at the top of the search tree.
- What to do? Regularly restarting search from the root node.

Restarting:

- avoids being stuck in large unsatisfiable portions of the search space
- brings diversification
- permits to test various heuristics
- permits to collect information (for adaptive heuristics) from run to run

Restarts



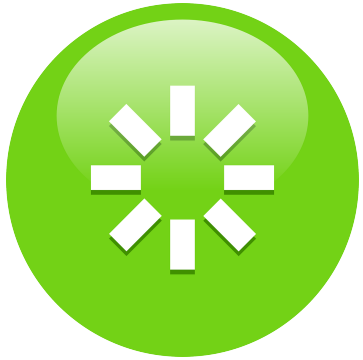
Remember that the first decisions are very important.

- Why? Because it can be very long to discard bad decisions performed at the top of the search tree.
- What to do? Regularly restarting search from the root node.

Restarting:

- avoids being stuck in large unsatisfiable portions of the search space
- brings diversification
- permits to test various heuristics
- permits to collect information (for adaptive heuristics) from run to run

Restarts



Remember that the first decisions are very important.

- Why? Because it can be very long to discard bad decisions performed at the top of the search tree.
- What to do? Regularly restarting search from the root node.

Restarting:

- avoids being stuck in large unsatisfiable portions of the search space
- brings diversification
- permits to test various heuristics
- permits to collect information (for adaptive heuristics) from run to run

Restarts

- When to restart? Search can be restarted after a certain limit:
  - number of decisions
  - number of backtracks
  - time
  - ...
- How to avoid exploring the same portions of the search space?
  - using randomization (partial answer)
  - recording nogoods (full answer)
- How to make search complete?
  - by regularly increasing the limit
  - by choosing a large enough limit

We focus on nogood recording from restarts (nrr)

Restarts

- When to restart? Search can be restarted after a certain limit:
  - number of decisions
  - number of backtracks
  - time
  - ...
- How to avoid exploring the same portions of the search space?
  - using randomization (partial answer)
  - recording nogoods (full answer)
- How to make search complete?
  - by regularly increasing the limit
  - by choosing a large enough limit

We focus on nogood recording from restarts (nrr)

Restarts

- When to restart? Search can be restarted after a certain limit:
  - number of decisions
  - number of backtracks
  - time
  - ...
- How to avoid exploring the same portions of the search space?
  - using randomization (partial answer)
  - recording nogoods (full answer)
- How to make search complete?
  - by regularly increasing the limit
  - by choosing a large enough limit

We focus on nogood recording from restarts (nrr)

Restarts

- When to restart? Search can be restarted after a certain limit:
  - number of decisions
  - number of backtracks
  - time
  - ...
- How to avoid exploring the same portions of the search space?
  - using randomization (partial answer)
  - recording nogoods (full answer)
- How to make search complete?
  - by regularly increasing the limit
  - by choosing a large enough limit

We focus on nogood recording from restarts (nrr)

Restarts

- When to restart? Search can be restarted after a certain limit:
  - number of decisions
  - number of backtracks
  - time
  - ...
- How to avoid exploring the same portions of the search space?
  - using randomization (partial answer)
  - recording nogoods (full answer)
- How to make search complete?
  - by regularly increasing the limit
  - by choosing a large enough limit

We focus on nogood recording from restarts (nrr)

Restarts

- When to restart? Search can be restarted after a certain limit:
  - number of decisions
  - number of backtracks
  - time
  - ...
- How to avoid exploring the same portions of the search space?
  - using randomization (partial answer)
  - recording nogoods (full answer)
- How to make search complete?
  - by regularly increasing the limit
  - by choosing a large enough limit

We focus on nogood recording from restarts (nrr)

Restarts

- When to restart? Search can be restarted after a certain limit:
  - number of decisions
  - number of backtracks
  - time
  - ...
- How to avoid exploring the same portions of the search space?
  - using randomization (partial answer)
  - recording nogoods (full answer)
- How to make search complete?
  - by regularly increasing the limit
  - by choosing a large enough limit

We focus on nogood recording from restarts (nrr)

Restarts

- When to restart? Search can be restarted after a certain limit:
  - number of decisions
  - number of backtracks
  - time
  - ...
- How to avoid exploring the same portions of the search space?
  - using randomization (partial answer)
  - recording nogoods (full answer)
- How to make search complete?
  - by regularly increasing the limit
  - by choosing a large enough limit

We focus on nogood recording from restarts (nrr)

nld-nogoods

Each branch of the search tree can be seen as a sequence of positive and negative decisions.

For any branch of the search tree starting from the root, a generalized nogood can be extracted from each negative decision. We call them nld-nogoods, nls standing for negative last decision.

Example

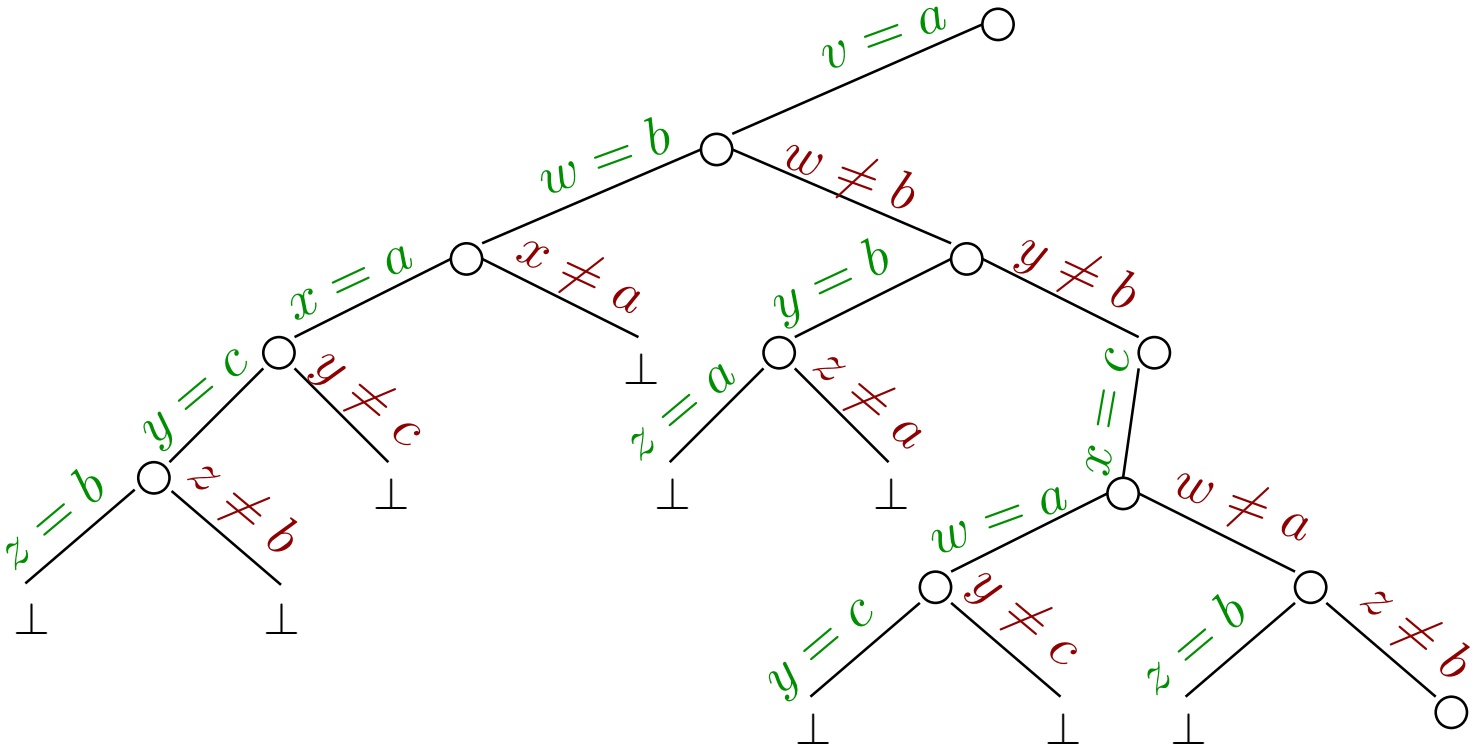


Figure: A partial search tree built by a backtracking search algorithm.

Example

On the righmost branch, there are four negative decisions. So, we obtain:

nld-subsequences	nld-nogoods
$\langle v = a, w \neq b \rangle$	$\{v = a, w = b\}$
$\langle v = a, w \neq b, y \neq b \rangle$	$\{v = a, w \neq b, y = b\}$
$\langle v = a, w \neq b, y \neq b, x = c, w \neq a \rangle$	$\{v = a, w \neq b, y \neq b, x = c, w = a\}$
$\langle v = a, w \neq b, y \neq b, x = c, w \neq a, z \neq b \rangle$	$\{v = a, w \neq b, y \neq b, x = c, w \neq a, z = b\}$

Example

On the righmost branch, there are four negative decisions. So, we obtain:

nld-subsequences	nld-nogoods
$\langle v = a, w \neq b \rangle$	$\{v = a, w = b\}$
$\langle v = a, w \neq b, y \neq b \rangle$	$\{v = a, w \neq b, y = b\}$
$\langle v = a, w \neq b, y \neq b, x = c, w \neq a \rangle$	$\{v = a, w \neq b, y \neq b, x = c, w = a\}$
$\langle v = a, w \neq b, y \neq b, x = c, w \neq a, z \neq b \rangle$	$\{v = a, w \neq b, y \neq b, x = c, w \neq a, z = b\}$

Example

Actually, it is proved that we can discard negative decisions from nld-nogoods, obtaining classical nogoods (reduced nld-nogoods) with a stronger filtering capability.

nld-nogoods	reduced nld-nogoods
$\{v = a, w = b\}$	$\{v = a, w = b\}$
$\{v = a, w \neq b, y = b\}$	$\{v = a, y = b\}$
$\{v = a, w \neq b, y \neq b, x = c, w = a\}$	$\{v = a, x = c, w = a\}$
$\{v = a, w \neq b, y \neq b, x = c, w \neq a, z = b\}$	$\{v = a, x = c, z = b\}$

Example

Actually, it is proved that we can discard negative decisions from nld-nogoods, obtaining classical nogoods (reduced nld-nogoods) with a stronger filtering capability.

nld-nogoods	reduced nld-nogoods
$\{v = a, w = b\}$	$\{v = a, w = b\}$
$\{v = a, w \neq b, y = b\}$	$\{v = a, y = b\}$
$\{v = a, w \neq b, y \neq b, x = c, w = a\}$	$\{v = a, x = c, w = a\}$
$\{v = a, w \neq b, y \neq b, x = c, w \neq a, z = b\}$	$\{v = a, x = c, z = b\}$

Example

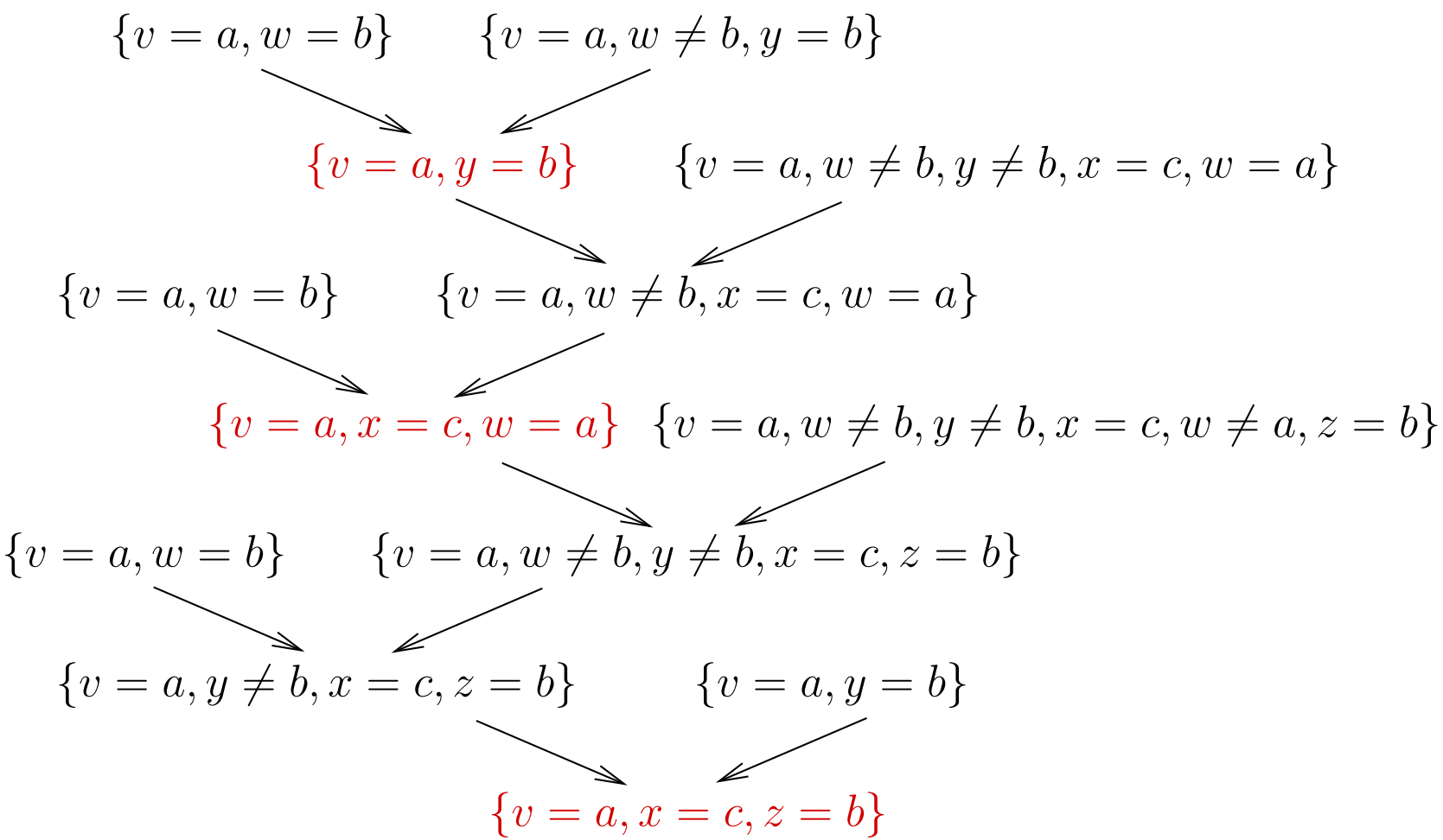


Figure: Derivation of reduced nld-nogoods using resolution.

Filtering with Watched Decisions

Suppose that we have the nogood:

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

which is equivalently represented by:

$$w \neq c \vee x \neq b \vee y \neq b \vee z \neq a$$

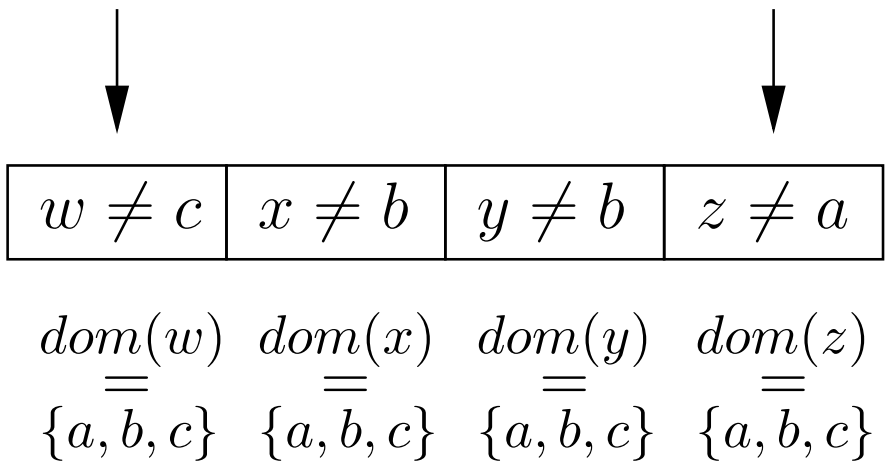


Figure: Initially (level 0), the first and last decision of the nogood (constraint) are watched.

Watched Decisions

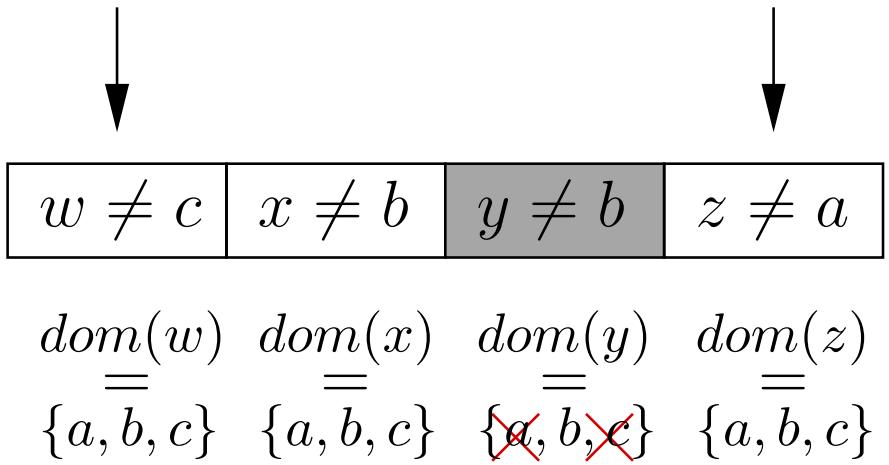


Figure: After setting $y = b$ (level 1). As $y \neq b$ was not watched, the nogood has not been handled.

Watched Decisions

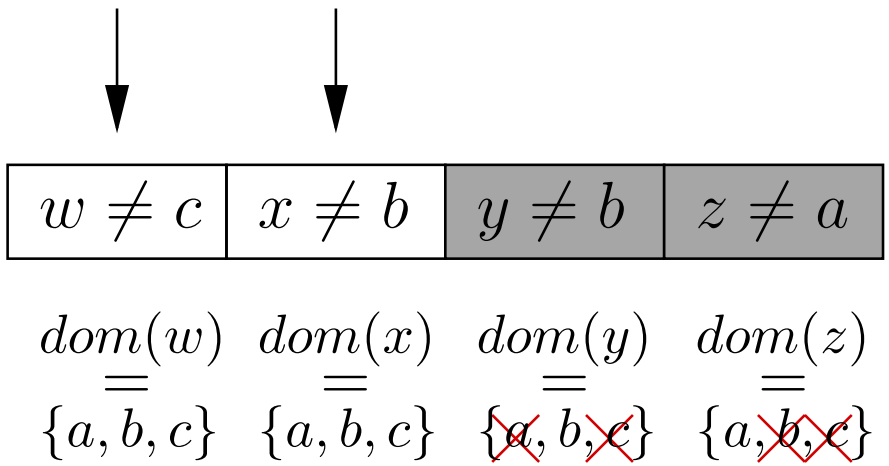


Figure: After setting $z = a$ (level 2). As $z \neq a$ was watched, a search for another watch occured. Here, $x \neq b$ has been found.

Watched Decisions

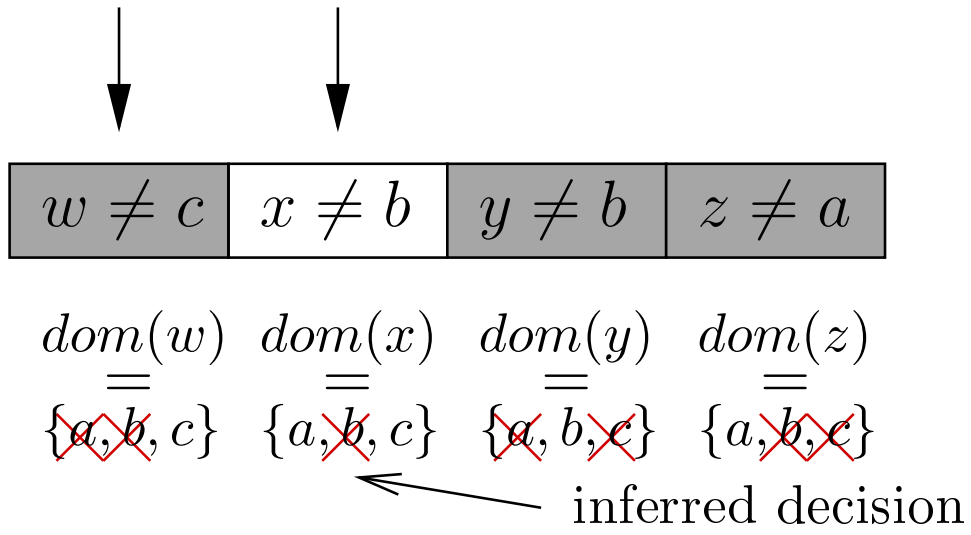


Figure: After setting $w = c$ (level 3). As $w \neq c$ was watched, a search for another watch occured. Here, no possibility remained. Consequently, $x \neq b$ has been inferred. The watched decisions are not modified.

Watched Decisions

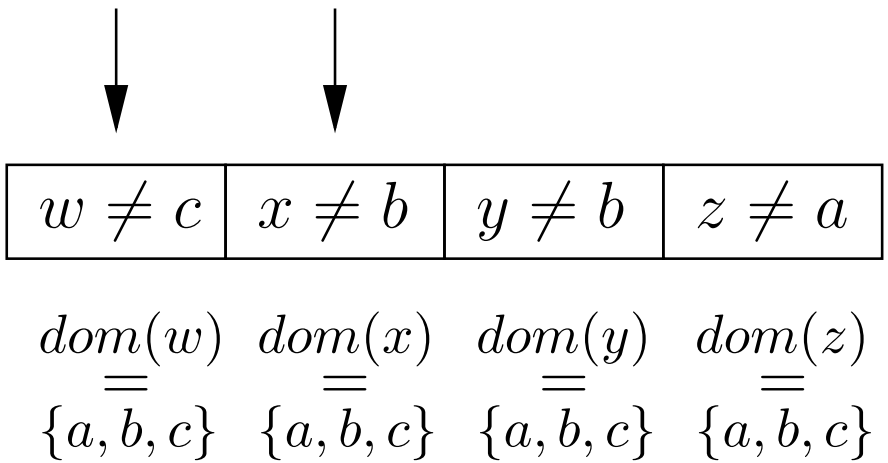
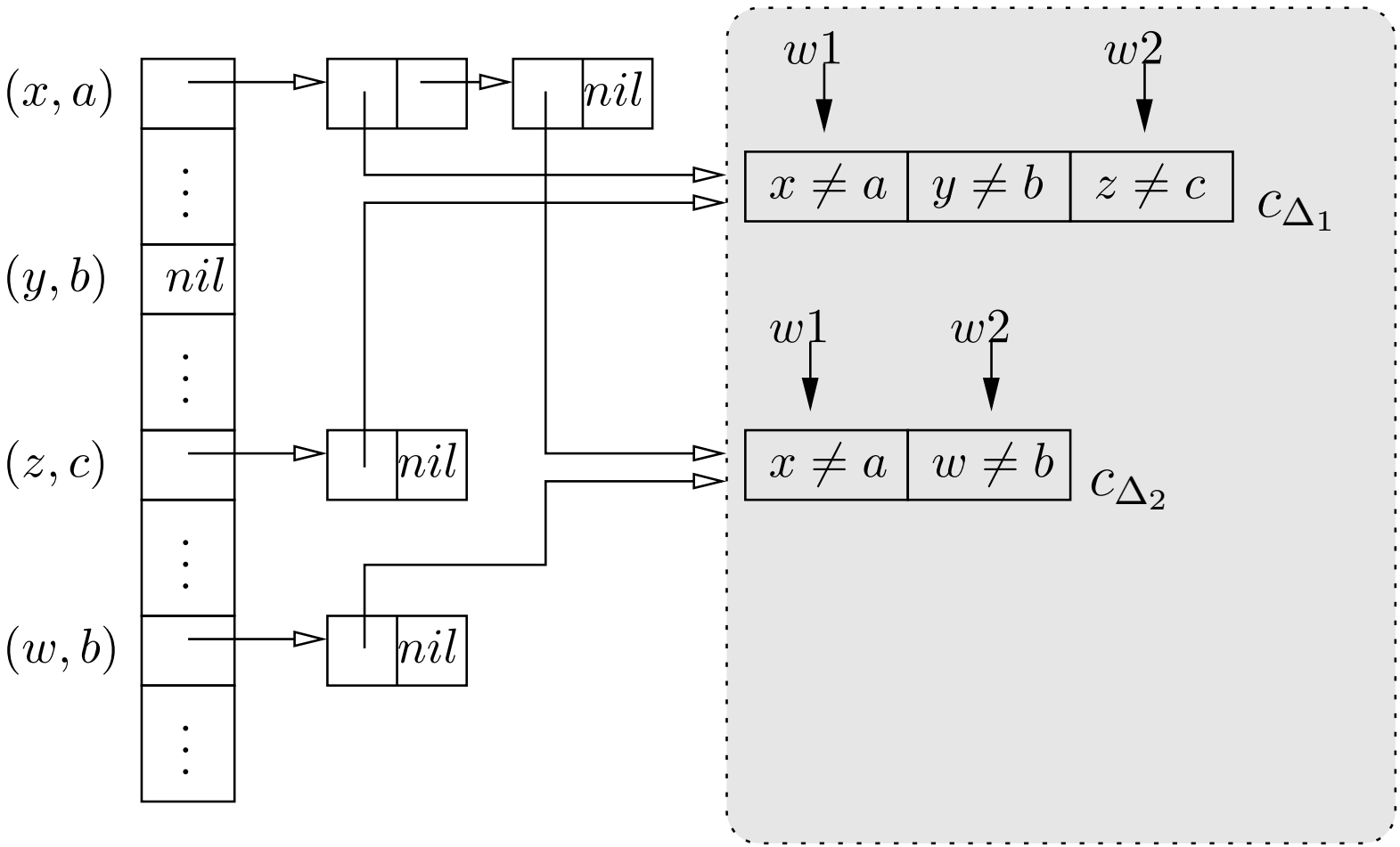
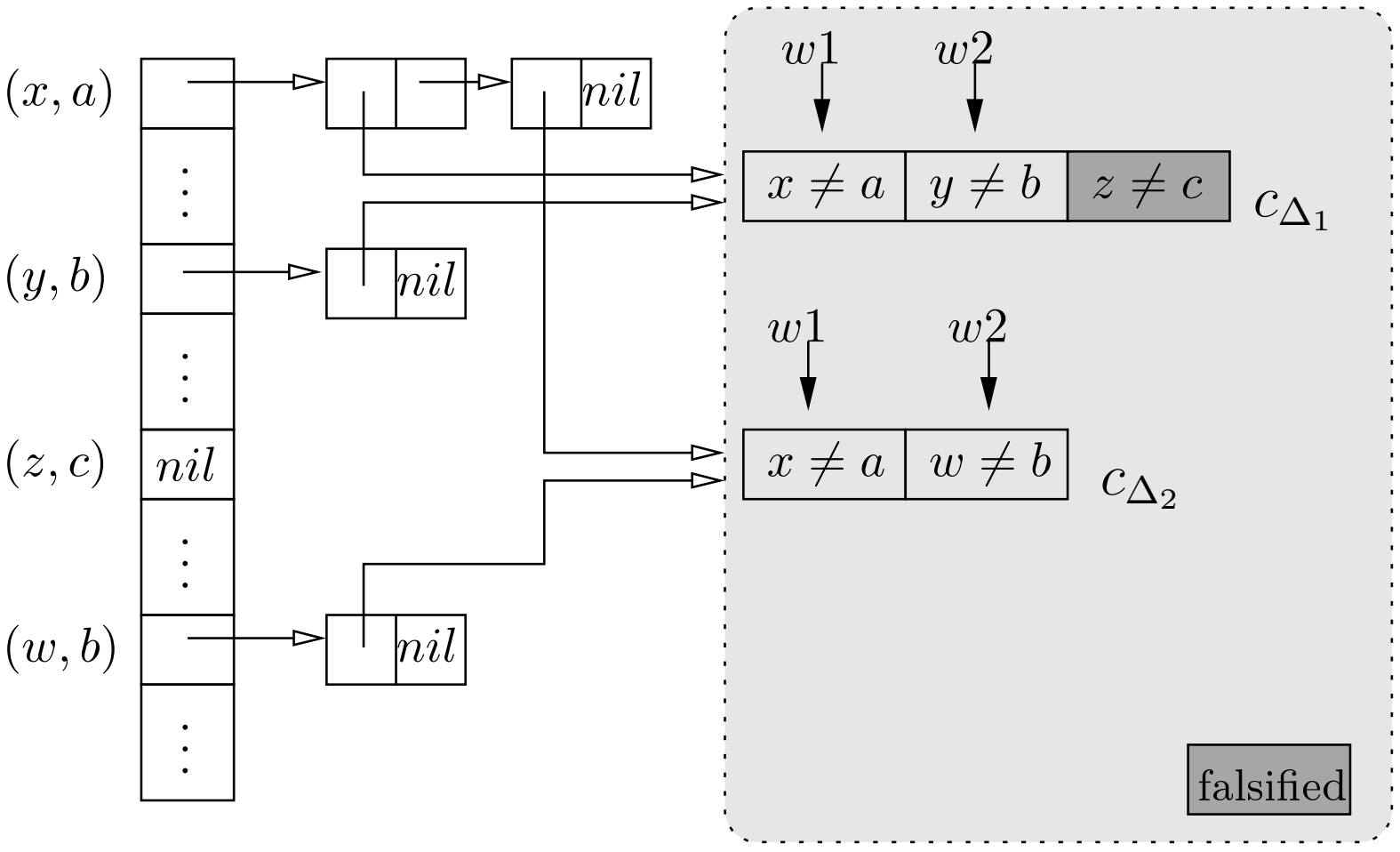


Figure: After backtracking to level 0. There is no need to modify watched decisions as found before backtracking.

Nogood Base 1/2



Nogood Base 2/2



Complexities

Proposition

The worst-case time complexity of extracting nogoods at each restart is $O(n^2 d)$.

Proposition

The worst-case time complexity of exploiting reduced nld-nogoods at each node of the search is $O(n|\mathscr{B}|)$.

Complexities

Proposition

The worst-case time complexity of extracting nogoods at each restart is $O(n^2 d)$.

Proposition

The worst-case time complexity of exploiting reduced nld-nogoods at each node of the search is $O(n|\mathscr{B}|)$.

Some Results

		MAC			
			+RST	*+RST+NG*	*+RST+NGm*
qk-12-5-*mul*	*dom/ddeg*	265.1	408.9	256.2	2.1
	brelaz	255.7	377.9	250.8	2.1
	dom/wdeg	3.1	1.8	2.6	1.6
qk-25-5-*mul*	*dom/ddeg*	timeout	timeout	timeout	4.9
	brelaz	timeout	timeout	timeout	5.1
	dom/wdeg	timeout	4.2	4.8	4.3
qk-50-5-*mul*	*dom/ddeg*	timeout	timeout	timeout	67.3
	brelaz	timeout	timeout	timeout	65.3
	dom/wdeg	timeout	59.5	44.6	43.9

Table: Cpu time to solve some instances of the queens-knights problem, given 20 minutes.

Outline

Group

Definition

A group is a pair $(G, \star)$ composed of a set G and a binary operation $\star$ defined on G, such that the following requirements are satisfied:

- Closure: $\forall f \in G, \forall g \in G,\ f \star g \in G$
- Identity element: $\exists e \in G \mid \forall f \in G,\ e \star f = f \star e = f$
- Inverse element: $\forall f \in G, \exists g \in G \mid f \star g = g \star f = e$; g is noted f^{-1}
- Associativity: $\forall f \in G, \forall g \in G, \forall h \in G,\ (f \star g) \star h = f \star (g \star h)$

Definition

- A subgroup of a group $(G, \star)$ is a group $(H, \star)$ such that $H \subseteq G$.
- The order of a group $(G, \star)$ is the number $|G|$ of elements in G.

Example

$(\mathbb{Z}, +)$ is a group, where $\mathbb{Z}$ denotes the set of integers, and $+$ denotes ordinary addition

Generating Set

A group can be represented by means of a subset of its elements, called *generating set*.

Definition

A generating set of a group $(G, \star)$ is a subset H of G such that each element of G can be expressed by composition (using $\star$) of elements of H, called *generators*, and their inverses.

Generating sets allow compact representations of groups :

Proposition

For any group $(G, \star)$, there exists a generating set of size $log_2(|G|)$ or smaller.

Permutation Groups

We are interested in permutation groups because we will be concerned by symmetries that are permutations.

Definition

A permutation on a set D is a bijection σ defined from D onto D. The image of an element $a \in D$ by σ is denoted by a^σ.

Example

For example, let $D = \{1, 2, 3, 4\}$ be a set of four integers. A possible permutation σ on D is: $1^\sigma = 2, 2^\sigma = 3, 3^\sigma = 1$ and $4^\sigma = 4$.

Remark.

A permutation can be represented by a set of *cycles* of the form $(a_1, a_2, \ldots, a_k)$ which means that:

- a_i is mapped to a_{i+1} for $i \in 1..k-1$
- and a_k is mapped to a_1.

Symmetry

In the following general definition of symmetry, a set of subsets of a set D is called a structure on D.

Definition

Let D be a set and $R \subseteq 2^D$ be a structure on D. A permutation σ on D is a symmetry on D for R iff $R^\sigma = R$.

A symmetry can be regarded as a permutation that preserves the structure R of a set D. Equivalently, the pair (D, R) can be regarded as a hypergraph H; a symmetry on D for R is then an automorphism of the hypergraph H.

Proposition

Let D be a set and $R \subseteq 2^D$ be a structure on D. The set of symmetries on D for R constitutes a group.

Dihedral Group

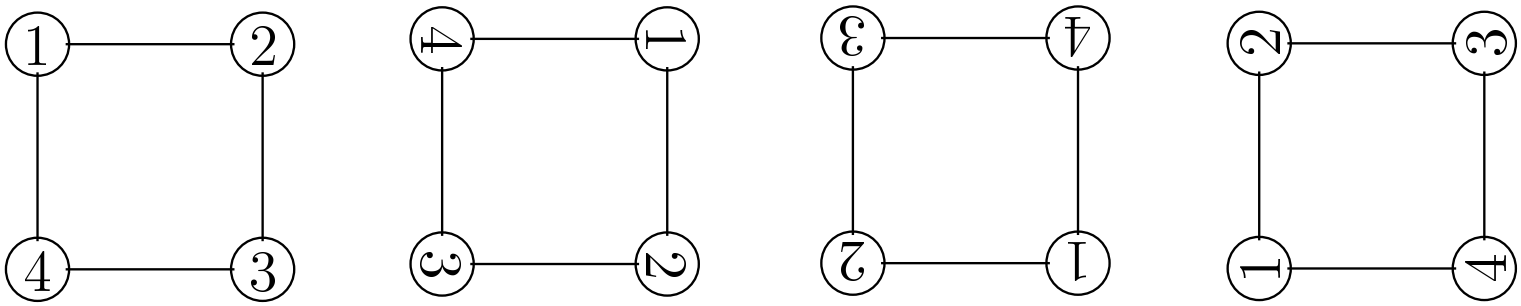
The symmetries (i.e., rotations and reflections) of a square form a group called a dihedral group. A square can be represented by:

- a set of four vertices $D = \{1, 2, 3, 4\}$
- and a set of four edges $R = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}\}$.

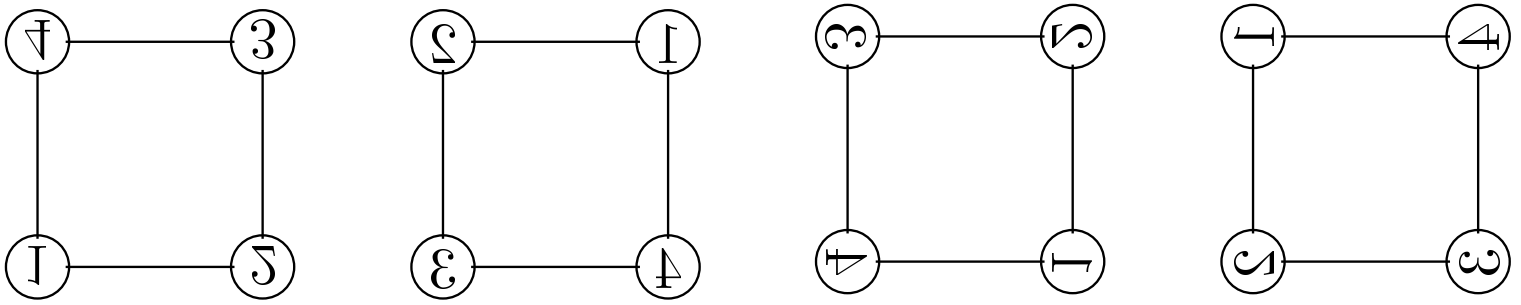
There are 8 symmetries for a square. In cyclic form, this gives:

- $r^0 = \{\}$
- $r^{90} = \{(1, 4, 3, 2)\}$
- $r^{180} = \{(1, 3), (2, 4)\}$
- $r^{270} = \{(1, 2, 3, 4)\}$
- $f^v = \{(1, 4), (2, 3)\}$
- $f^h = \{(1, 2), (3, 4)\}$
- $f^d = \{(1, 3)\}$
- $f^c = \{(2, 4)\}$

Dihedral Group



(a) r^0 (b) r^{90} (c) r^{180} (d) r^{270}



(e) f^v (f) f^h (g) f^d (h) f^c

Figure: The eight symmetries of the symmetry group of the square. r^0 is the identity permutation.

Symmetries on Constraint Networks

In the context of a CN P, we consider that:

- D is the set $vals(P) = \{(x, a) : x \in vars(P) \wedge a \in dom(x)\}$
- R is a set of instantiations on P (i.e., built from $vals(P)$)

Definition

Let P be a CN and R be a set of instantiations on P. A symmetry on P for R is a permutation σ of $vals(P)$ such that $R^\sigma = R$.

Sometimes symmetries are restricted to variables or values.

Definition

Let P be a CN and R be a set of instantiations on P.

- A variable symmetry on P for R is a symmetry σ on P for R such that for every $(x, a) \in vals(P)$, $(x, a)^\sigma = (x^{\sigma_{vars}}, a)$ where σ_{vars} is a permutation of $vars(P)$.
- A value symmetry on P for R is a symmetry σ on P for R such that for every $(x, a) \in vals(P)$, $(x, a)^\sigma = (x, a^{\sigma_x})$ where σ_x is a permutation of $dom(x)$.

Solution and Constraint Symmetries

We now define solution and constraint symmetries simply by considering two special structures (sets of instantiations).

Definition

A solution symmetry on a constraint network P is a symmetry on P for $sols(P)$, i.e. a permutation σ of $vals(P)$ such that $sols(P)^\sigma = sols(P)$.

Definition

A constraint symmetry on a constraint network P is a symmetry on P for $\mu(P)$, i.e. a permutation σ of $vals(P)$ such that $\mu(P)^\sigma = \mu(P)$ where $\mu(P)$ is the set of hyperedges in the micro-structure of P.

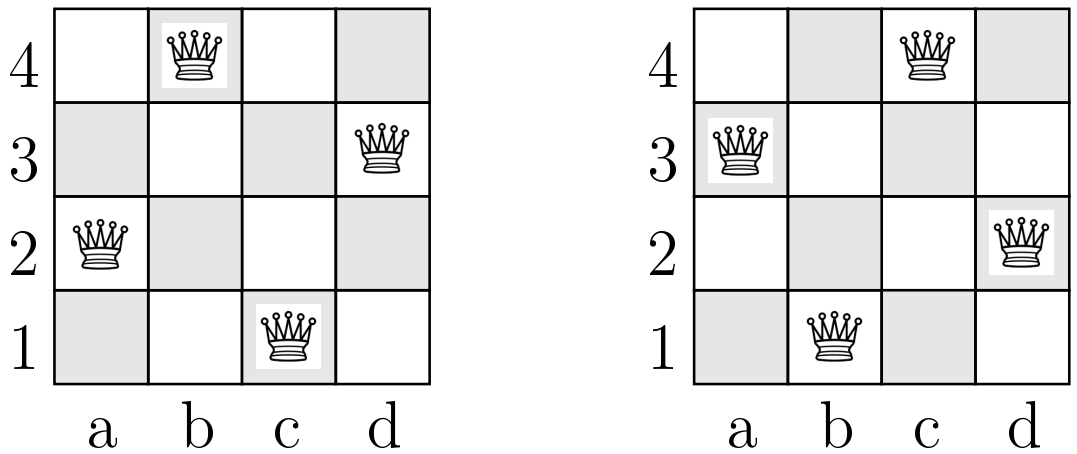
4-queens Instance

This instance has exactly eight constraint symmetries, which are the geometrical ones. For r^{90},

- a first cycle is $((x_a, 1), (x_a, 4), (x_d, 4), (x_d, 1))$,
- a second cycle is $((x_a, 2), (x_b, 4), (x_d, 3), (x_c, 1))$, etc.

Among these symmetries,

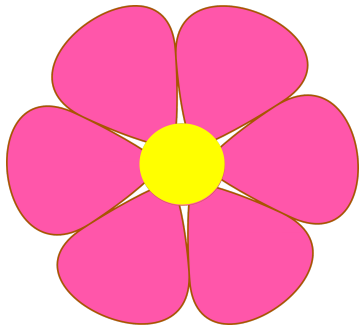
- only r^0 and f^h are variable symmetries
- only r^0 and f^v are value symmetries.



(a) First Solution (b) Second Solution

Figure: The two solutions of the `4-queens` instance.

Symmetry-breaking Methods

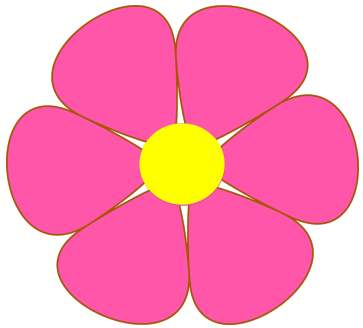


Three main classical ways to tackle symmetries:

- reformulate the model to reduce the number of symmetries
- add symmetry-breaking constraints to the model
- avoid during search exporing nodes that are symmetric to nodes already explored

We only focus on (automatic) variable symmetry-breaking

Symmetry-breaking Methods

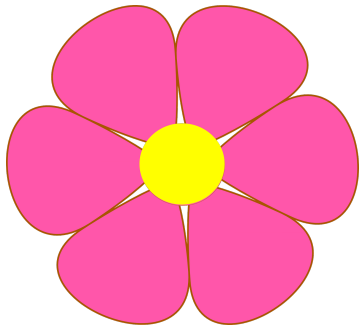


Three main classical ways to tackle symmetries:

- reformulate the model to reduce the number of symmetries
- add symmetry-breaking constraints to the model
- avoid during search exporing nodes that are symmetric to nodes already explored

We only focus on (automatic) variable symmetry-breaking

Symmetry-breaking Methods

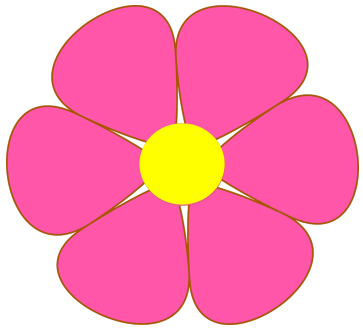


Three main classical ways to tackle symmetries:

- reformulate the model to reduce the number of symmetries
- add symmetry-breaking constraints to the model
- avoid during search exporing nodes that are symmetric to nodes already explored

We only focus on (automatic) variable symmetry-breaking

Symmetry-breaking Methods



Three main classical ways to tackle symmetries:

- reformulate the model to reduce the number of symmetries
- add symmetry-breaking constraints to the model
- avoid during search exporing nodes that are symmetric to nodes already explored

We only focus on (automatic) variable symmetry-breaking

Symmetry-breaking Methods

All variables symmetries can be broken by adding the following lexicographic ordering constraints to the constraint network P:

$$\langle x_1, x_2, \ldots, x_n \rangle \leq_{lex} \langle x_1, x_2, \ldots, x_n \rangle^{\sigma}, \forall \sigma \in Sym$$

which is equivalent to:

$$\langle x_1, x_2, \ldots, x_n \rangle \leq_{lex} \langle x_1^{\sigma}, x_2^{\sigma}, \ldots, x_n^{\sigma} \rangle, \forall \sigma \in Sym$$

Example

If we consider the variable symmetry f^h on the `4-queens` instance, we obtain:

$$\langle x_a, x_b, x_c, x_d \rangle \leq_{lex} \langle x_a, x_b, x_c, x_d \rangle^{f^h}$$

which gives:

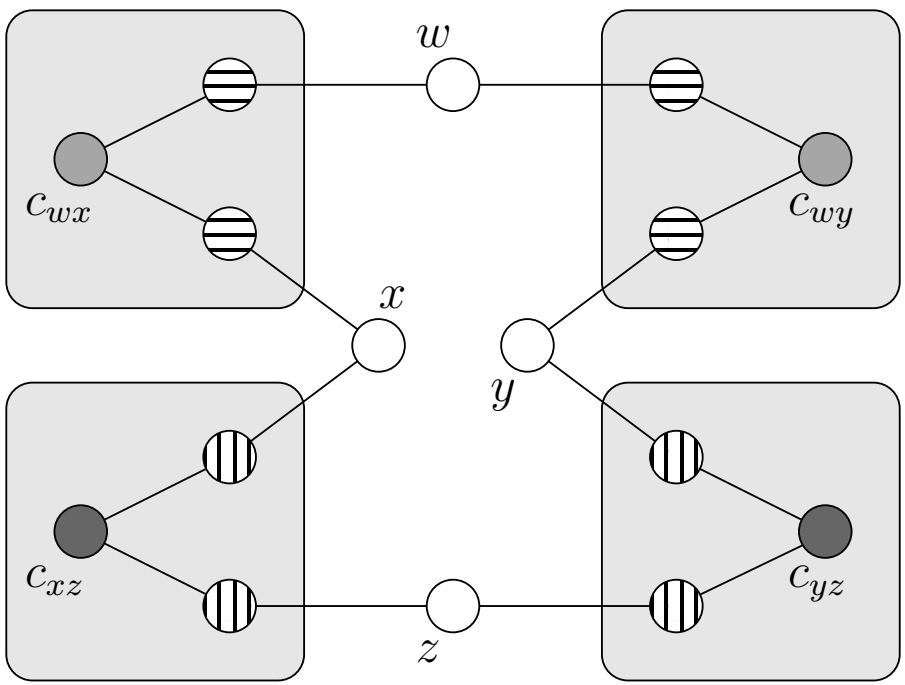
$$\langle x_a, x_b, x_c, x_d \rangle \leq_{lex} \langle x_d, x_c, x_b, x_a \rangle$$

Automatic Symmetry Detection

Definition

Two variables x and y are locally symmetrical for a constraint c such that $\{x, y\} \subseteq scp(c)$ iff $\forall \tau \in rel(c), \tau_{x \leftrightarrow y} \in rel(c)$.

We can build a colored graph from locally symmetrical variables.



Automatic Symmetry Detection

Each automorphism of the colored graph identifies a variable symmetry.

We can run a graph automorphism algorithms such as :

- Saucy
- Nauty

For each generator of the automorphism group, identified by Saucy or Nauty, we post a symmetry-breaking constraint.

Remark.

The size of the group of symmetries is often exponential. Only exploiting generators of a generating set is reasonable because they represent maximally independent symmetries.

BIBD

A BIBD is a v by b binary matrix with exactly r ones per row, k ones per column, and with a scalar product of λ between any pair of distinct rows. A BIBD is therefore specified by its parameters (v, b, r, k, λ). An example of a solution for $(7, 7, 3, 3, 1)$:

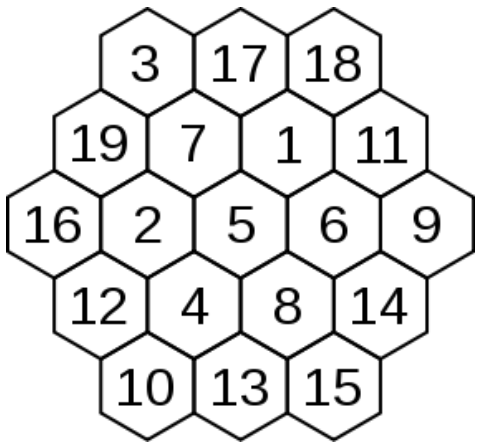
```
0 1 1 0 0 1 0
1 0 1 0 1 0 0
0 0 1 1 0 0 1
1 1 0 0 0 0 1
0 0 0 0 1 1 1
1 0 0 1 0 1 0
0 1 0 1 1 0 0
```

With x denoting the 0/1 matrix, we can add the following constraint in PyCSP^3:

```
LexIncreasing(x, matrix=True)
```

Magic Hexagon

A (normal) magic hexagon of order n is an arrangement of numbers 1 to $3n^2 - 3n + 1$ in an hexagon with n cells on each edge, in such a way that the numbers in each row, in all three directions, sum to the same magic constant M. Example for order 3:

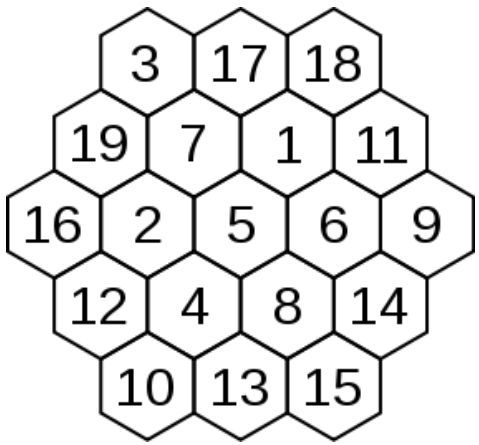


With x denoting the matrix for the hexagon (with row indices starting all at 0), and with $d = n + n - 1$, we can break symmetries by posting:

$$x_{0,0} < x_{0,n-1} \qquad x_{0,0} < x_{n-1,d-1}$$
$$x_{0,0} < x_{d-1,n-1} \qquad x_{0,0} < x_{d-1,0}$$
$$x_{0,0} < x_{n-1,0} \qquad x_{0,n-1} < x_{n-1,0}$$

Magic Hexagon

A (normal) magic hexagon of order n is an arrangement of numbers 1 to $3n^2 - 3n + 1$ in an hexagon with n cells on each edge, in such a way that the numbers in each row, in all three directions, sum to the same magic constant M. Example for order 3:



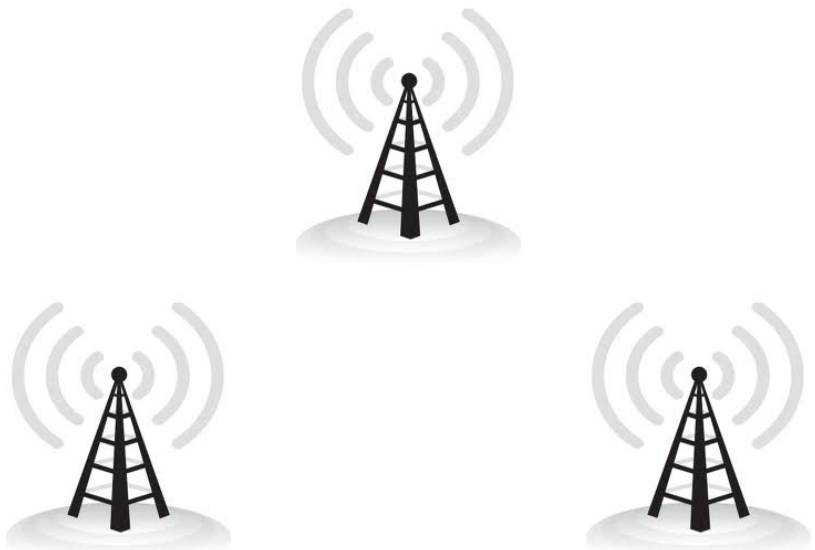
With x denoting the matrix for the hexagon (with row indices starting all at 0), and with $d = n + n - 1$, we can break symmetries by posting:

$$x_{0,0} < x_{0,n-1} \qquad x_{0,0} < x_{n-1,d-1}$$
$$x_{0,0} < x_{d-1,n-1} \qquad x_{0,0} < x_{d-1,0}$$
$$x_{0,0} < x_{n-1,0} \qquad x_{0,n-1} < x_{n-1,0}$$

Outline

Scenario

To summarize what we have seen so far, let us consider a solving scenario.

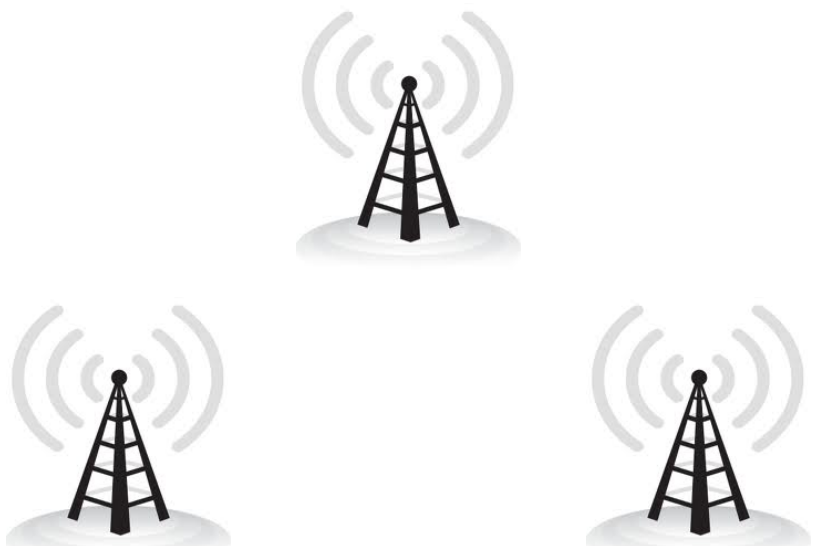


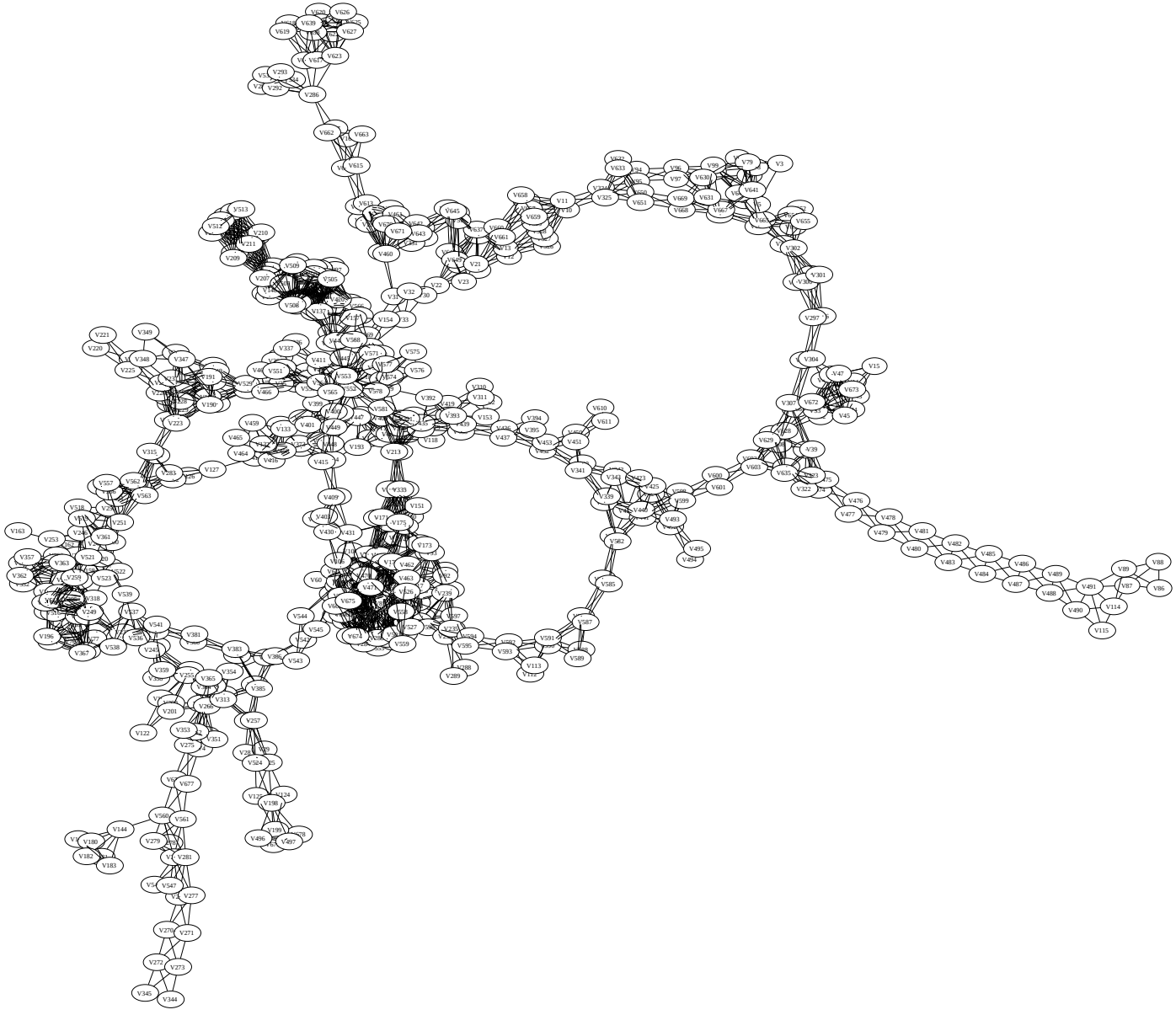
RLFAP (CELAR, project CALMA, Cabon et al. 1999)

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.

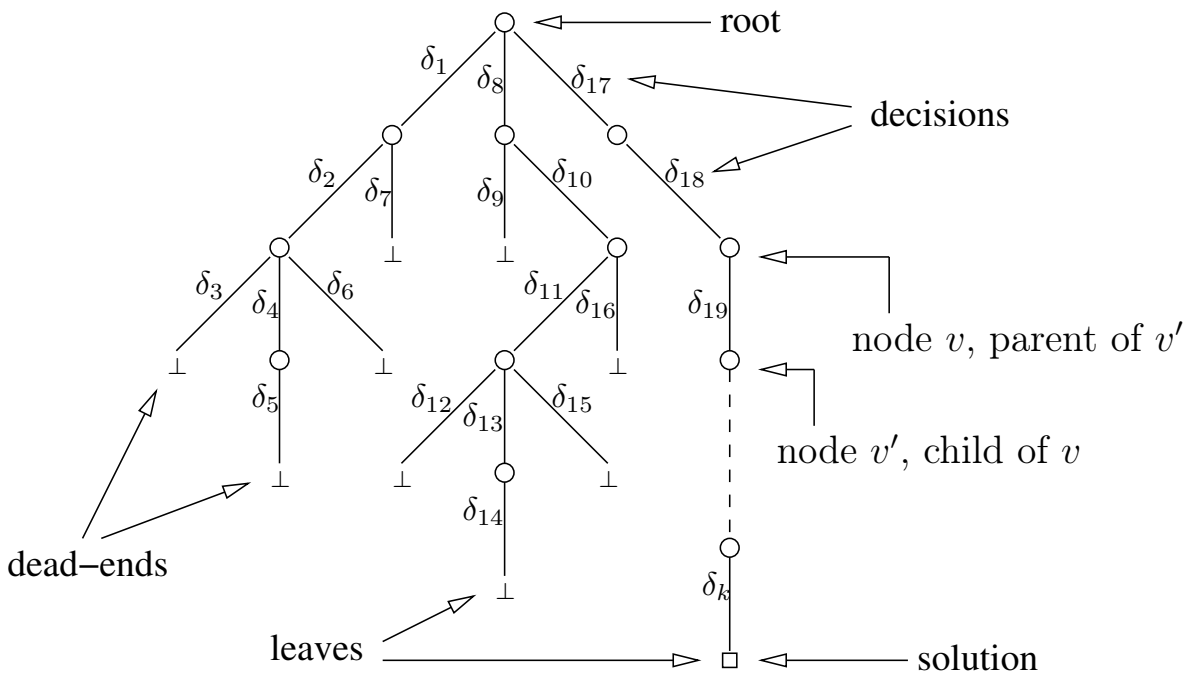




Structure of CSP instances `scen11`, `scen11-f12`, ...:
680 variables, 4,103 binary constraints

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*

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.

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.

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

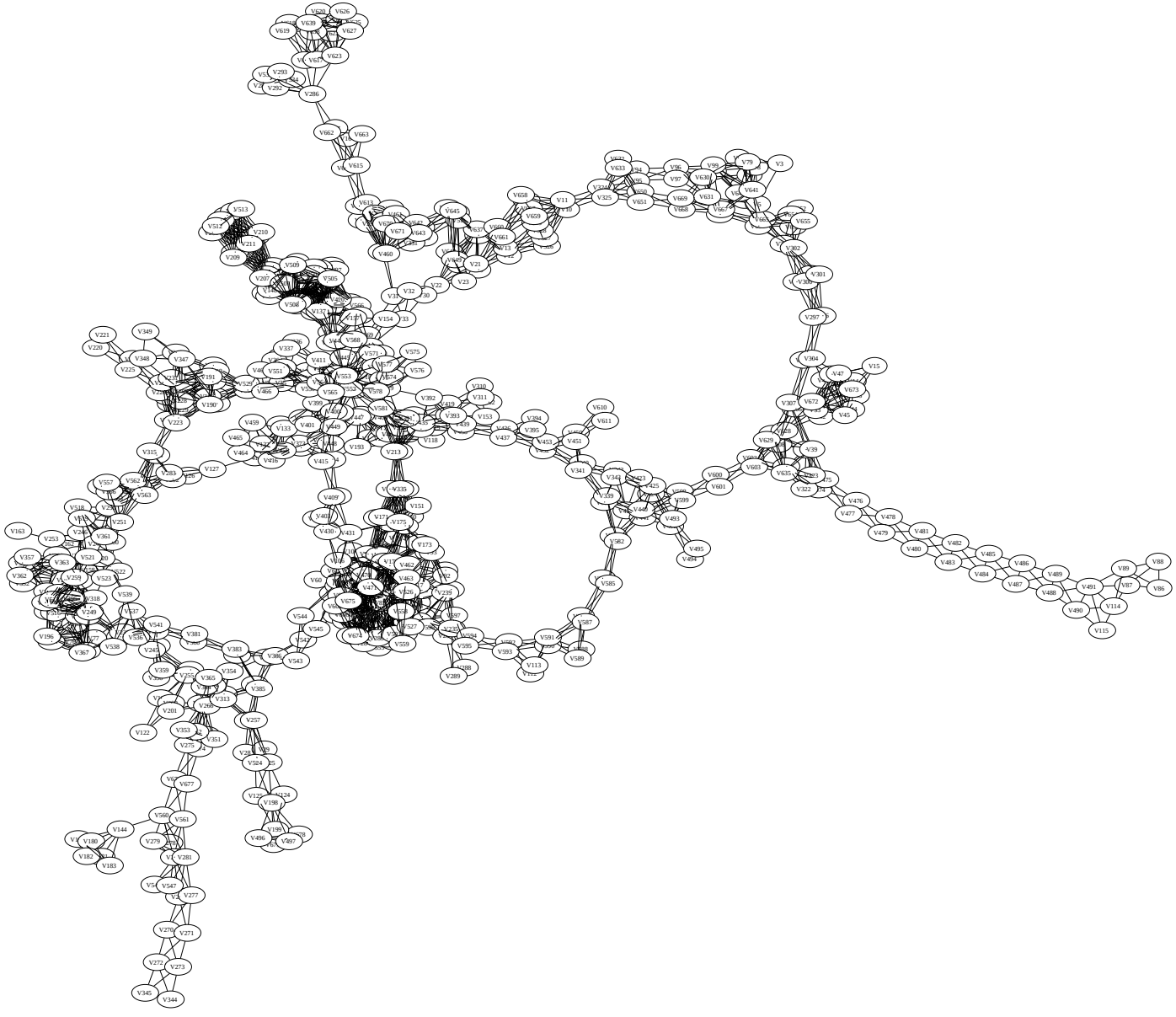


Illustration of Constraint Weighting with scen11-f6

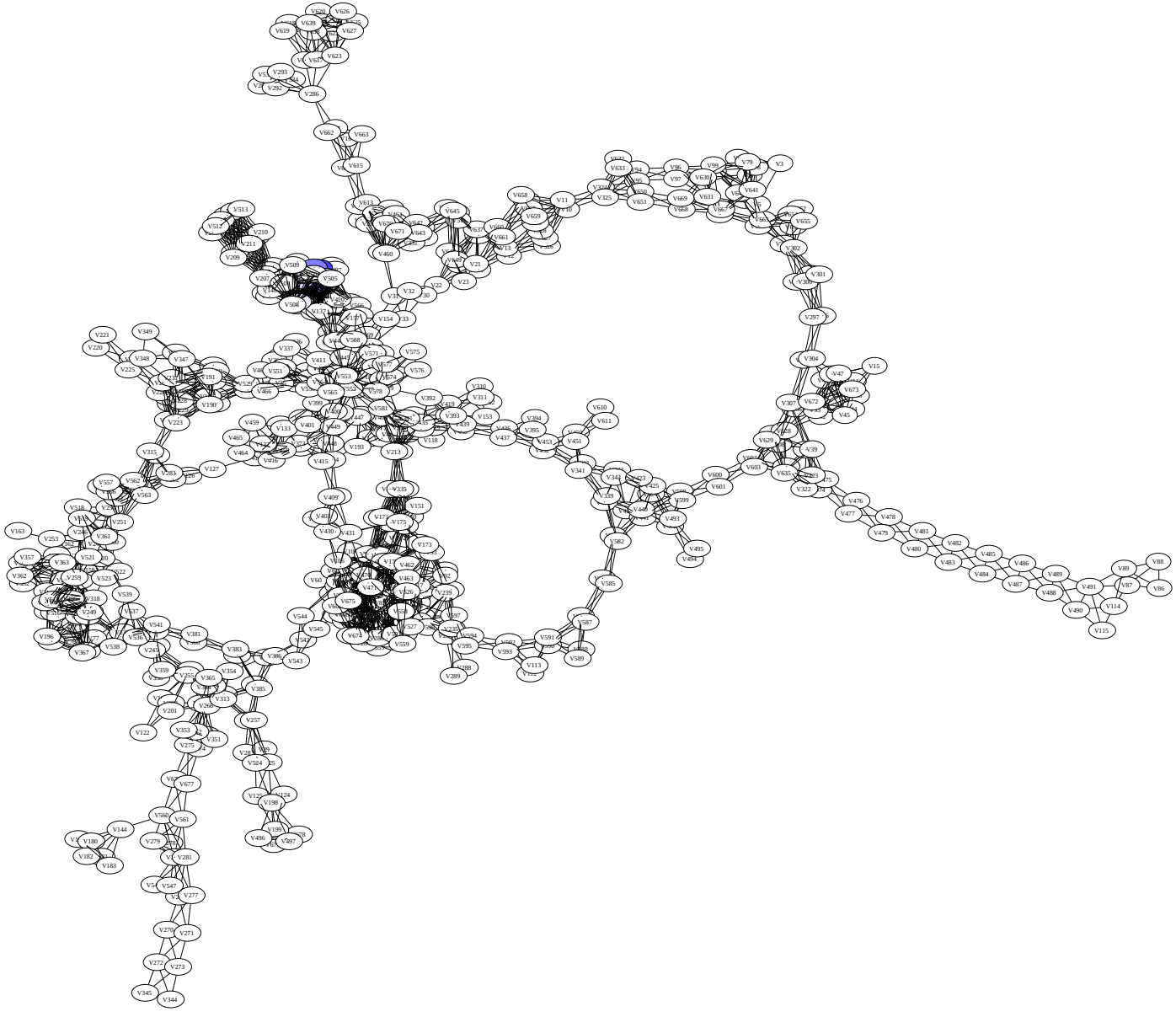


Illustration of Constraint Weighting with scen11-f6

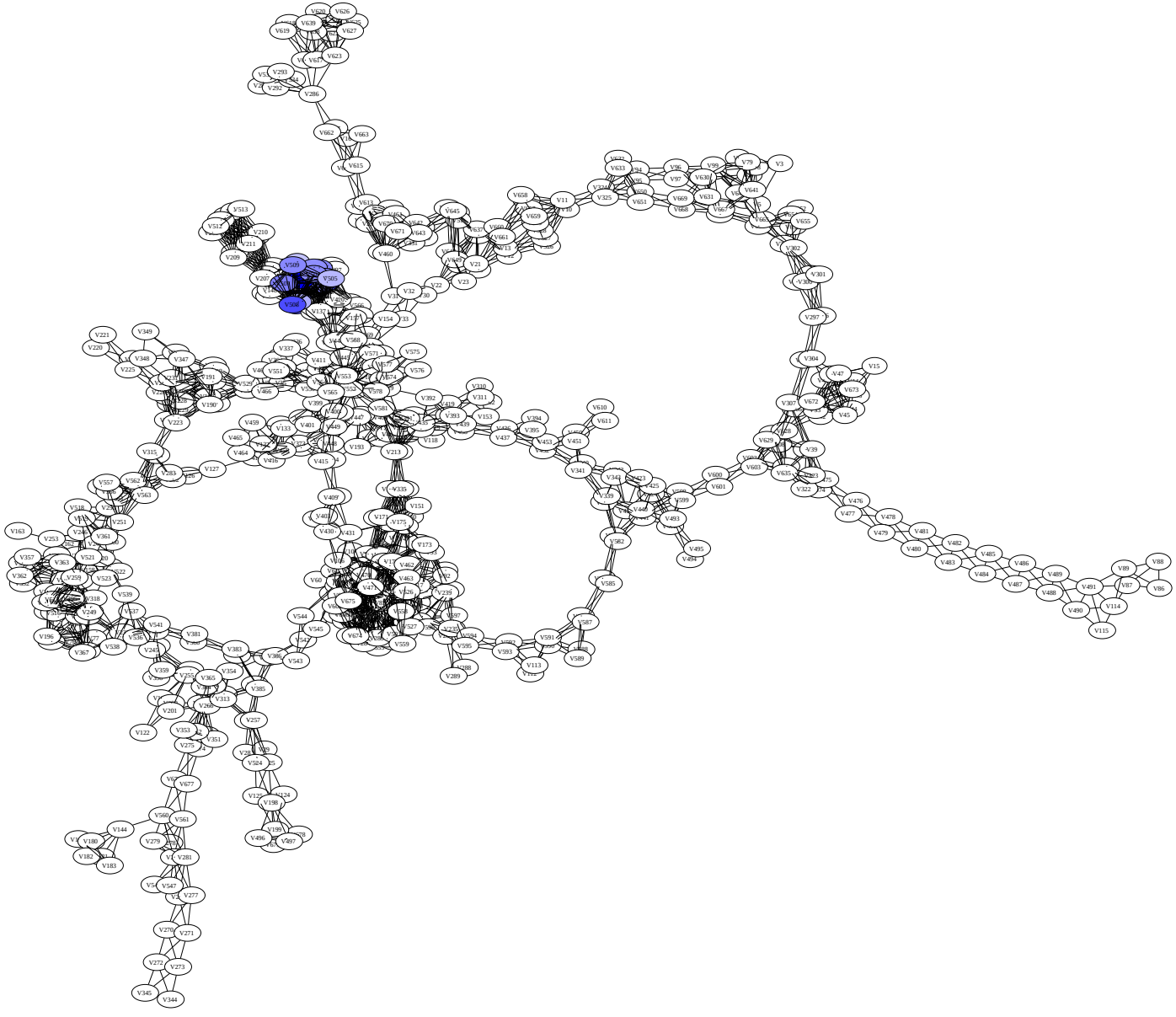
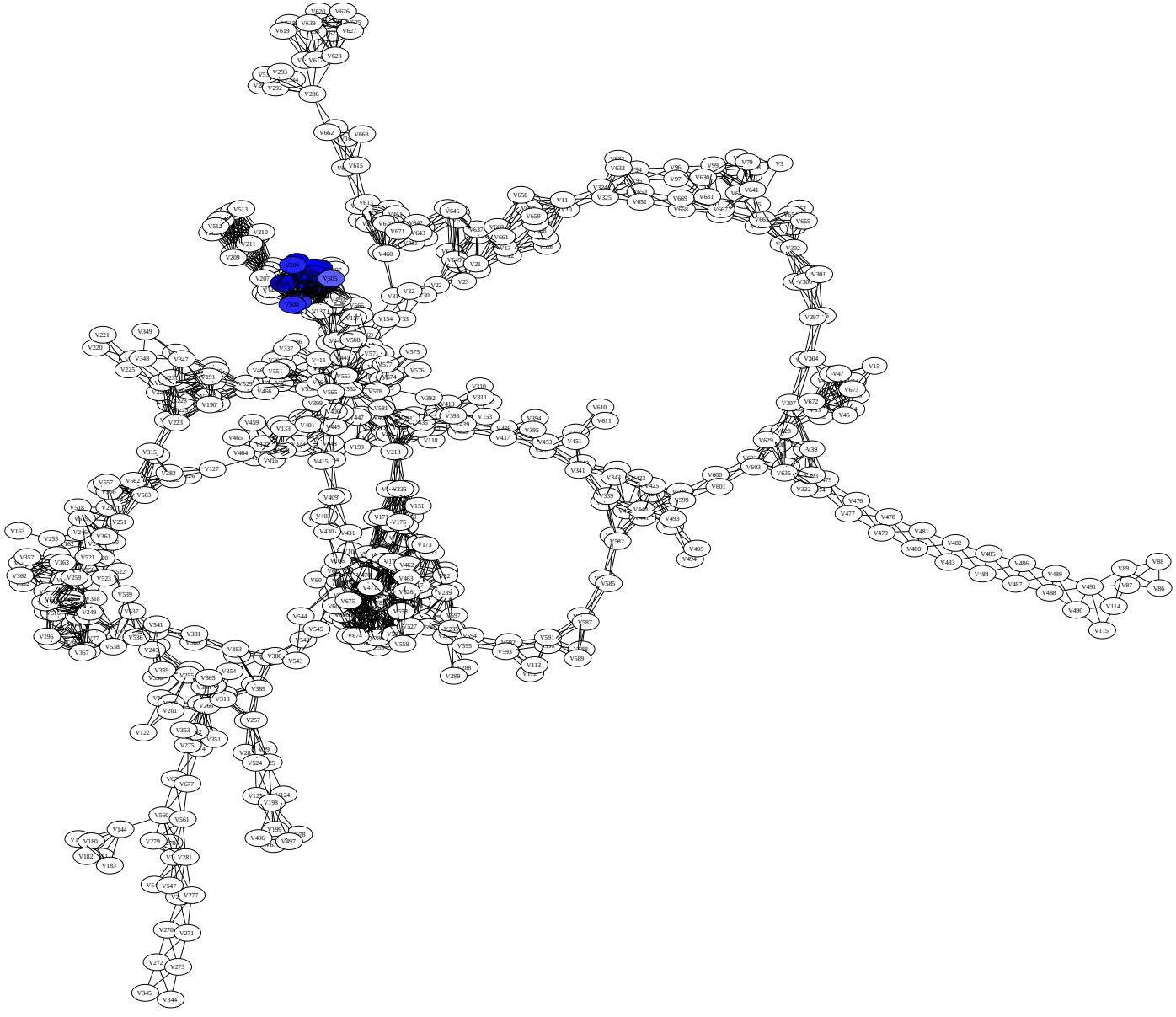


Illustration of Constraint Weighting with scen11-f6



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

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

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

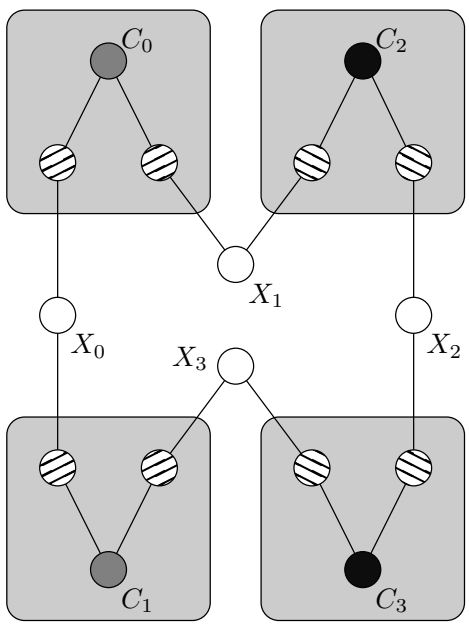
Instances	nodes	CPU (4)	CPU (3)
`scen11`	882	1.48	1.47
`scen11-f12`	353	1.39	1.49
`scen11-f8`	1,264	1.56	2.80
`scen11-f6`	33,542	4.45	25.20
`scen11-f4`	421,097	37.2	292.0
`scen11-f2`	4,310,576	356	3,158
`scen11-f1`	11,096,549	921	7,805

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

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

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

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

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

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

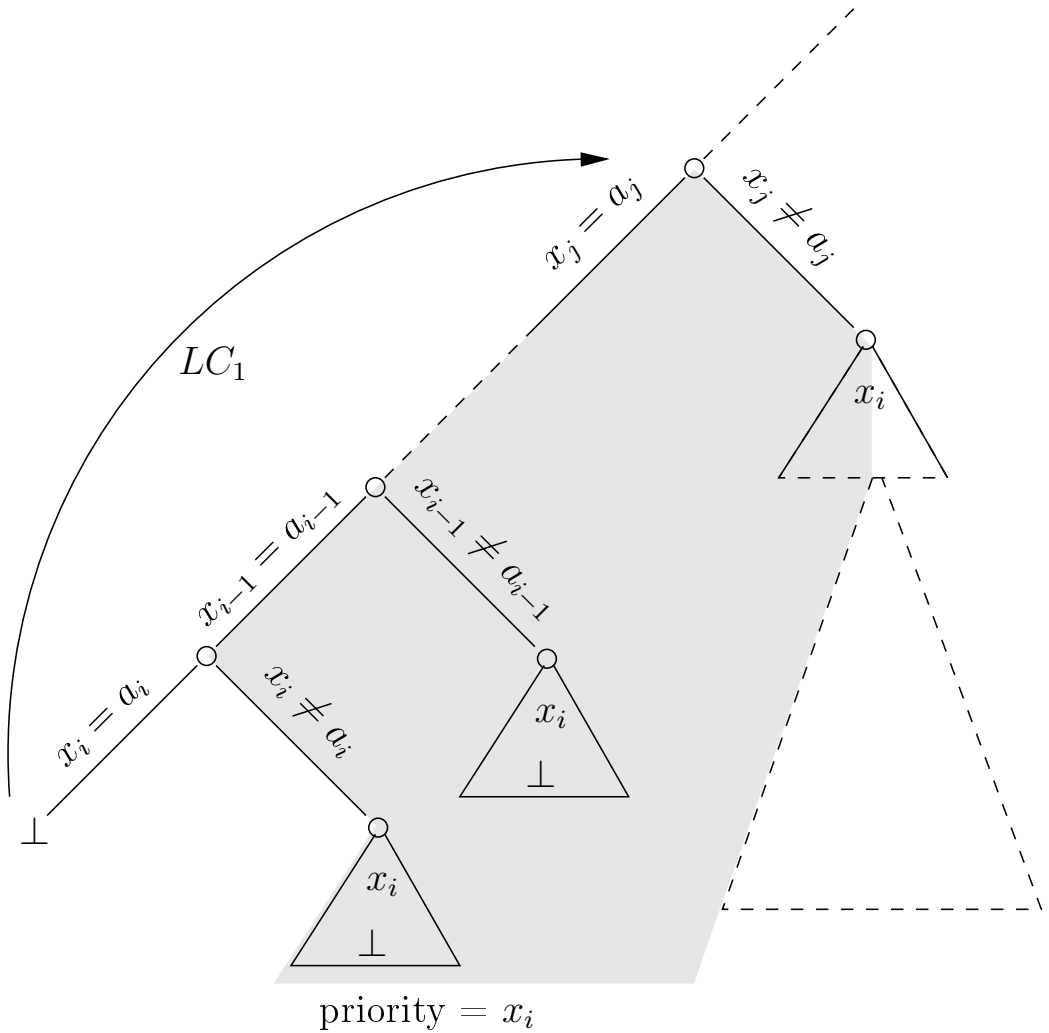
Results (5) – MAC-*dom/wdeg*-nrr-sb

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

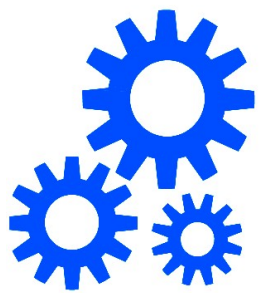
Instances	nodes	CPU (6)	CPU (5)
`scen11`	1,173	1.57	1.59
`scen11-f12`	187	1.48	1.51
`scen11-f8`	191	1.48	1.56
`scen11-f6`	273	1.51	1.69
`scen11-f4`	957	1.82	1.86
`scen11-f2`	5,101	2.19	2.96
`scen11-f1`	11,305	2.84	3.03

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**.

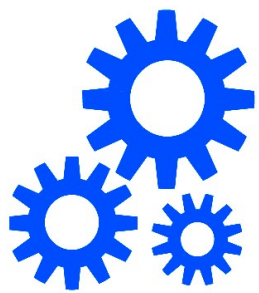


Strong Preprocessing

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**.

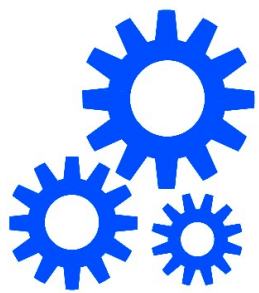


Strong Preprocessing

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