An Introduction to Constraint Programming

Hugues Wattez

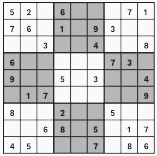
March 31, 2022

Tutorial at LIX

Demotivation

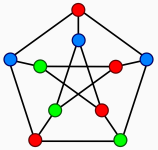
Constraints are everywhere...

Demotivation



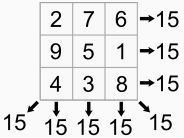
Sudoku...

Demotivation



Graph Coloring...

Demotivation



Magicsquare...

Demotivation

Crossword...

Demotivation

Scheduling...

Demotivation

(Car) Sequencing...

Demotivation

(Vehicle) Routing...

Demotivation

(Container Ship) Loading...

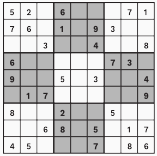
Demotivation

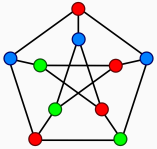
Bioinformatic (sequence analysis, protein structure prediction, ...)...

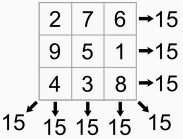
Demotivation

Music Composition...

Demotivation







... and many others!!

Motivation

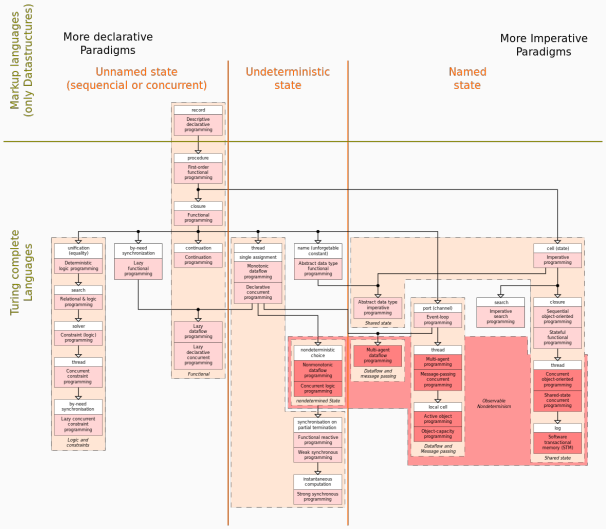
This fully constrained world could create a cognitive overload that Constraint Programming (CP) tries to tackle:

> *Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.*
>
> *- Eugene Freuder*

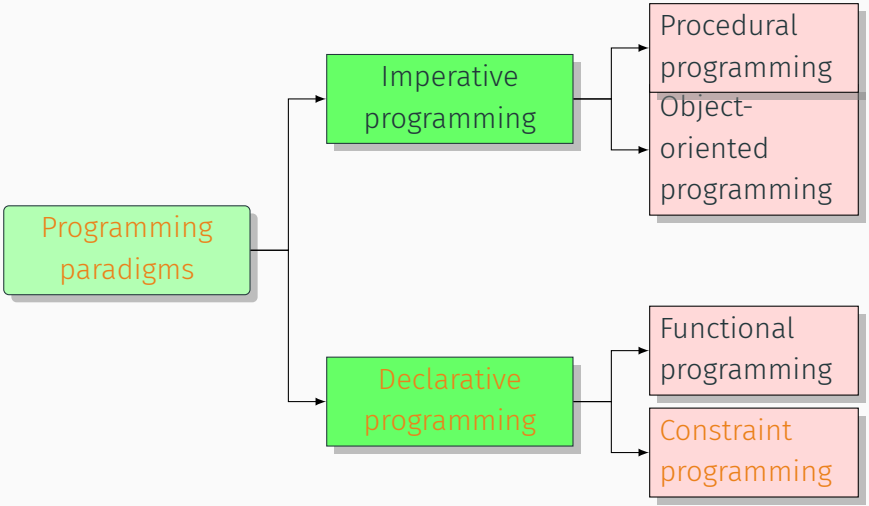
Outlook

Introduction

Overview of the various programming paradigms (Peter Van Roy)



A simplified overview of the programming paradigms



Which roots for the constraint programming?

Constraint Programming (CP) has several mechanisms commonly belonging to the Operations Research (OR) or/and Artificial Intelligence (AI):

- branching procedure
- inference
- heuristic
- learning

Constraint Satisfaction Problem

Variable

Definition (Variable)

A *variable* x is an entity associated to a value. This value belongs to its *domain*, denoted $\texttt{dom}(x)$.

Remark

In this introduction, we deal only with integer variables.

Constraint

Definition (Constraint)

A *constraint* c is defined by a set of variables, called *scope* of c and denoted $\texttt{scp}(c)$, and by a mathematical relation which describes the set of tuples allowed by c for the variables of its scope.

Many kind of constraints:

- intension: $x + y < z$ or $a \wedge b \Rightarrow c$
- extension: $(x, y) \in \{ (1, 2), (1, 3), (2, 4) \}$
- global: $\texttt{all} - \texttt{diff}(t)$ où t est un tableau de variables

Constraint

Definition (Constraint)

A *constraint* c is defined by a set of variables, called *scope* of c and denoted $\texttt{scp}(c)$, and by a mathematical relation which describes the set of tuples allowed by c for the variables of its scope.

Many kind of constraints:

- intension: $x + y < z$ or $a \wedge b \Rightarrow c$
- extension: $(x, y) \in \{ (1, 2), (1, 3), (2, 4) \}$
- global: $\texttt{all} - \texttt{diff}(t)$ où t est un tableau de variables

Constraint

Definition (Constraint)

A *constraint* c is defined by a set of variables, called *scope* of c and denoted $\texttt{scp}(c)$, and by a mathematical relation which describes the set of tuples allowed by c for the variables of its scope.

Many kind of constraints:

- intension: $x + y < z$ or $a \wedge b \Rightarrow c$
- extension: $(x, y) \in \{ (1, 2), (1, 3), (2, 4) \}$
- global: $\texttt{all} - \texttt{diff}(t)$ où t est un tableau de variables

Constraint

Definition (Constraint)

A *constraint* c is defined by a set of variables, called *scope* of c and denoted $\texttt{scp}(c)$, and by a mathematical relation which describes the set of tuples allowed by c for the variables of its scope.

Many kind of constraints:

- intension: $x + y < z$ or $a \wedge b \Rightarrow c$
- extension: $(x, y) \in \{ (1, 2), (1, 3), (2, 4) \}$
- global: $\texttt{all} - \texttt{diff}(t)$ où t est un tableau de variables

Constraint

Definition (Constraint)

A *constraint* c is defined by a set of variables, called *scope* of c and denoted $\texttt{scp}(c)$, and by a mathematical relation which describes the set of tuples allowed by c for the variables of its scope.

Many kind of constraints:

- intension: $x + y < z$ or $a \wedge b \Rightarrow c$
- extension: $(x, y) \in \{ (1, 2), (1, 3), (2, 4) \}$
- global: $\texttt{all} - \texttt{diff}(t)$ où t est un tableau de variables

Constraint Satisfaction Problem (CSP)

Definition (CSP)

A *Constraint Satisfaction Problem* (or *Constraint Network*) $\mathcal{P}$ is defined by:

- a finite set of variables, denoted $\mathcal{X}$
- a finite set of constraints, denoted $\mathcal{C}$, such that $\forall c \in \mathcal{C}, \mathbf{scp}(c) \subseteq \mathcal{X}$

The objective of a CSP is responding to this simple question: “Is there a satisfiable solution?” (which is NP-complete).

A solution for a CSP

Definition (Solution)

A *solution* of a CSP instance $\mathcal{P}$ corresponds to the assignment of a value to each variable of $\mathcal{X}$ such that all the constraints of $\mathcal{C}$ are satisfied.

Study problem: Sudoku (rules)

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

The objective is to fill a 9 × 9 grid with digits so that

- each column,
- each row, and
- each of the nine 3 × 3 subgrids

that compose the grid contain all of the digits from 1 to 9.

The puzzle setter provides a partially completed grid (clues), which for a well-posed puzzle has a single solution.

- Wikipedia

Study problem: Sudoku (CSP)

Data: Let D is the set of clues where each clue $d \in D$ corresponds to a fixed value d_v in the cell $(d_i, d_j) \in (1..9, 1..9)$.

Problem: Let $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ where:

Study problem: Sudoku (CSP)

Data: Let D is the set of clues where each clue $d \in D$ corresponds to a fixed value d_v in the cell $(d_i, d_j) \in (1..9, 1..9)$.

Problem: Let $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ where:

$$\mathcal{X} = \left\{ \begin{array}{cccc} x_{1,1} & x_{1,2} & \dots & x_{1,9} \\ x_{2,1} & x_{2,2} & \dots & x_{2,9} \\ \vdots & \vdots & \vdots & \vdots \\ x_{9,1} & x_{9,2} & \dots & x_{9,9} \end{array} \right\} \text{ and } \texttt{dom}(x_{i,j}) = \{1, \dots, 9\} \ \forall \ i, j \in 1..9$$

Study problem: Sudoku (CSP)

Data: Let D is the set of clues where each clue $d \in D$ corresponds to a fixed value d_v in the cell $(d_i, d_j) \in (1..9, 1..9)$.

Problem: Let $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ where:

$$\mathcal{C} = \left\{ \begin{array}{ll} \texttt{all\_diff}(x_{i,1}, \ldots, x_{i,9}) & \forall\ i \in 1..9 \\ \texttt{all\_diff}(x_{1,j}, \ldots, x_{9,j}) & \forall\ j \in 1..9 \\ \texttt{all\_diff} \begin{pmatrix} x_{k,l} & x_{k,l+1} & x_{k,l+2} \\ x_{k+1,l} & x_{k+1,l+1} & x_{k+1,l+2} \\ x_{k+2,l} & x_{k+2,l+1} & x_{k+2,l+2} \end{pmatrix} & \forall\ k, l \in \{1, 4, 7\} \\ x_{d_i,d_j} = d_v & \forall\ d \in D \end{array} \right\}$$

Study problem: Sudoku (PyCSP)

A Sudoku implementation example with the PyCSP modeling language:
`http://pycsp.org/documentation/models/CSP/Sudoku/`

... and many others through this documentation.

Solving Principles of a Constraint Solver

Greedy solving

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

A greedy way of solving this CSP could be an exploration of the overall set of complete assignations:

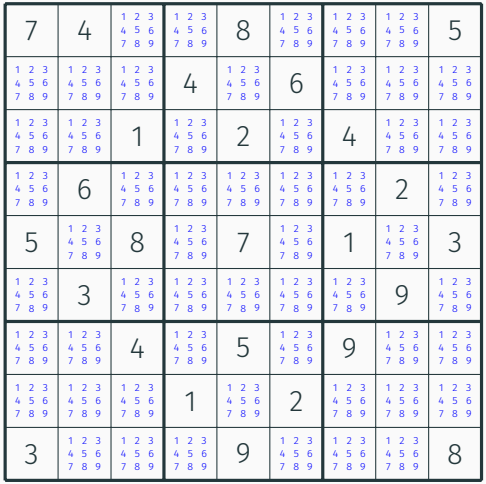
> $9^{81-|D|}$ *(*$> 3.0 \times 10^{52}$*) possible assignations for this current example* 🤯.

In practice, as we are human, we use inference and heuristic mechanisms to solve the sudoku...

... constraint solvers too!

Inference

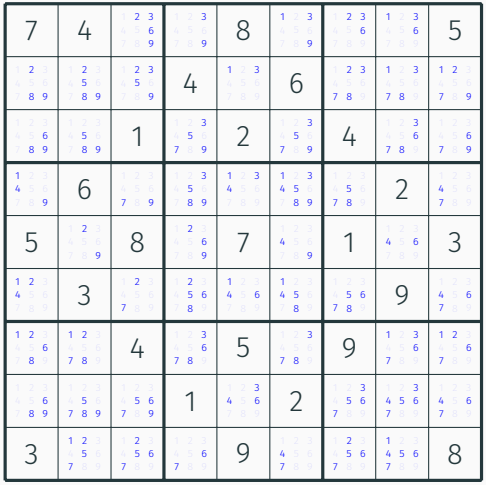
In modern contraint solvers, it exists many inference properties.



Inference

In modern contraint solvers, it exists many inference properties.

Usually, the *arc-consistency* (`AC`) property is applied after each variable assignation: we check the consistency of each value of each variable domains and the inconsistent values are removed.

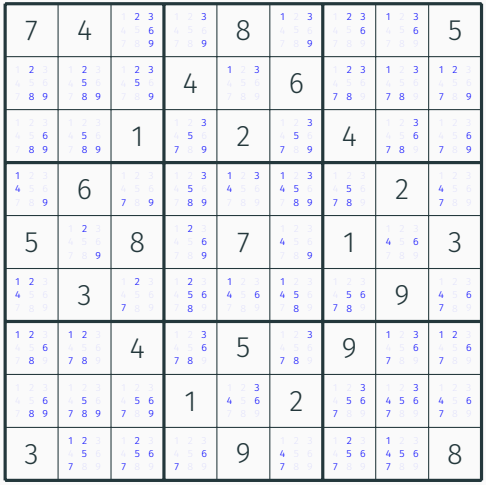


Inference

In modern contraint solvers, it exists many inference properties.

Usually, the *arc-consistency* (`AC`) property is applied after each variable assignation: we check the consistency of each value of each variable domains and the inconsistent values are removed.

Lighter and stronger properties exist: the choice of one them is the balance between number of filtered values and time consumed.

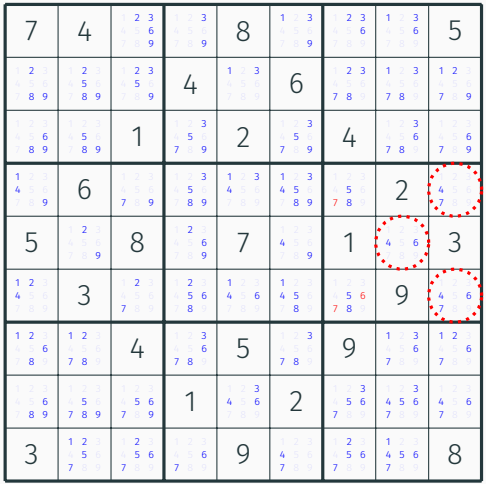


Inference

In modern contraint solvers, it exists many inference properties.

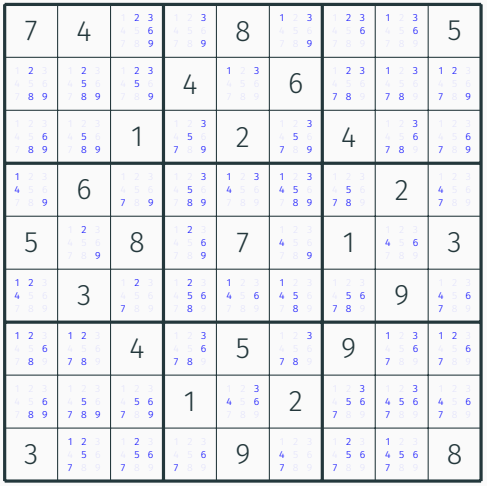
Usually, the *arc-consistency* (`AC`) property is applied after each variable assignation: we check the consistency of each value of each variable domains and the inconsistent values are removed.

Lighter and stronger properties exist: the choice of one them is the balance between number of filtered values and time consumed.



A stronger inference property could be able to remove these values! But sometimes it could be more efficient to assign, fail and backtrack.

Branching heuristics



Static variable ordering heuristics:

- `deg`: number of constraints where a variable is involved (max)
- `dom`: domaine size of a variable (min)
- `dom/deg`: combination (min)

Dynamic variable ordering heuristics:

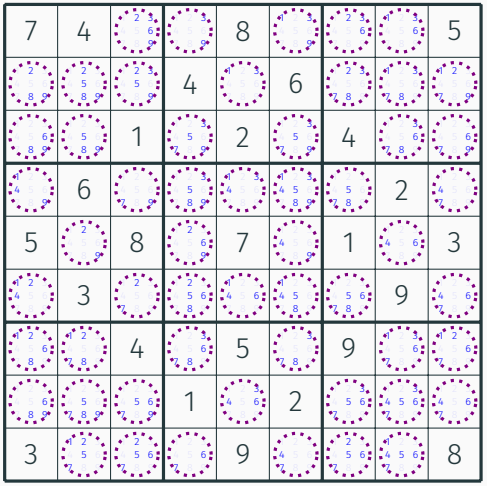
- `ddeg`, `ddom`, `ddeg/ddom`: the dynamic version of the previous ones

Adaptive variable ordering heuristics:

- `wdeg`: weighting the conflictual constraints (max)

Finally, usually `min-val` is used as the value ordering heuristic.

Branching heuristics



Static variable ordering heuristics:

- `deg`: number of constraints where a variable is involved (max)
- `dom`: domaine size of a variable (min)
- `dom/deg`: combination (min)

Dynamic variable ordering heuristics:

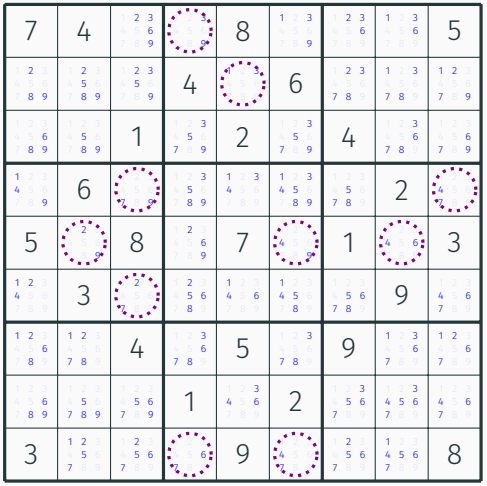
- `ddeg`, `ddom`, `ddeg/ddom`: the dynamic version of the previous ones

Adaptive variable ordering heuristics:

- `wdeg`: weighting the conflictual constraints (max)

Finally, usually `min-val` is used as the value ordering heuristic.

Branching heuristics



Static variable ordering heuristics:

- `deg`: number of constraints where a variable is involved (max)
- `dom`: domaine size of a variable (min)
- `dom/deg`: combination (min)

Dynamic variable ordering heuristics:

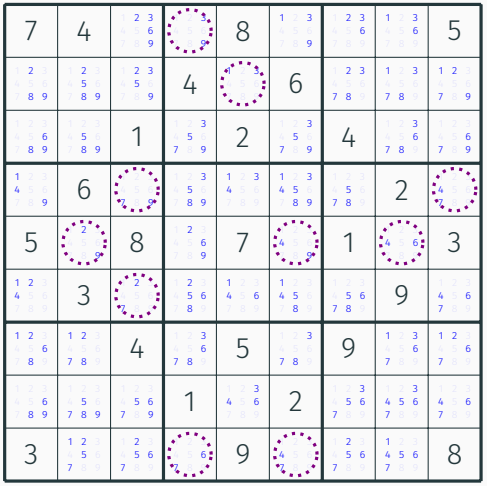
- `ddeg`, `ddom`, `ddeg/ddom`: the dynamic version of the previous ones

Adaptive variable ordering heuristics:

- `wdeg`: weighting the conflictual constraints (max)

Finally, usually `min-val` is used as the value ordering heuristic.

Branching heuristics



Static variable ordering heuristics:

- `deg`: number of constraints where a variable is involved (max)
- `dom`: domaine size of a variable (min)
- `dom/deg`: combination (min)

Dynamic variable ordering heuristics:

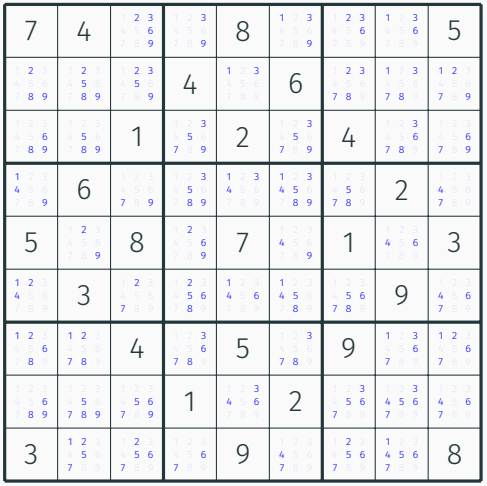
- `ddeg`, `ddom`, `ddeg/ddom`: the dynamic version of the previous ones

Adaptive variable ordering heuristics:

- `wdeg`: weighting the conflictual constraints (max)

Finally, usually `min-val` is used as the value ordering heuristic.

Branching heuristics



Static variable ordering heuristics:

- `deg`: number of constraints where a variable is involved (max)
- `dom`: domaine size of a variable (min)
- `dom/deg`: combination (min)

Dynamic variable ordering heuristics:

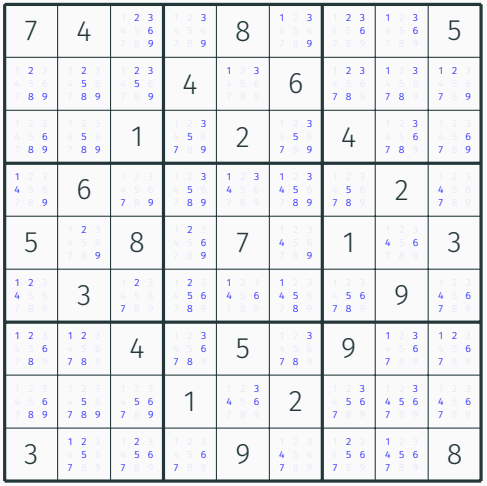
- `ddeg`, `ddom`, `ddeg/ddom`: the dynamic version of the previous ones

Adaptive variable ordering heuristics:

- `wdeg`: weighting the conflictual constraints (max)

Finally, usually `min-val` is used as the value ordering heuristic.

Branching heuristics



Static variable ordering heuristics:

- `deg`: number of constraints where a variable is involved (max)
- `dom`: domaine size of a variable (min)
- `dom/deg`: combination (min)

Dynamic variable ordering heuristics:

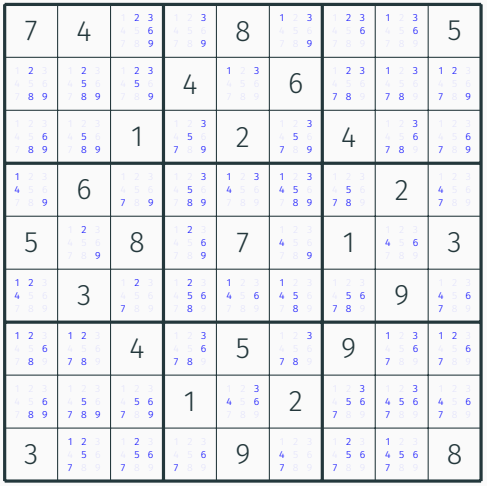
- `ddeg`, `ddom`, `ddeg/ddom`: the dynamic version of the previous ones

Adaptive variable ordering heuristics:

- `wdeg`: weighting the conflictual constraints (max)

Finally, usually `min-val` is used as the value ordering heuristic.

Branching heuristics



Static variable ordering heuristics:

- `deg`: number of constraints where a variable is involved (max)
- `dom`: domaine size of a variable (min)
- `dom/deg`: combination (min)

Dynamic variable ordering heuristics:

- `ddeg`, `ddom`, `ddeg/ddom`: the dynamic version of the previous ones

Adaptive variable ordering heuristics:

- `wdeg`: weighting the conflictual constraints (max)

Finally, usually `min-val` is used as the value ordering heuristic.

Restart mechanism and learning

Sometimes, the search needs to be restarted from the beginning. This is due to:

- the constraint network changes (evolving domains, ...)
- the heuristics learn
- the research stagnates without finding a solution (*Heavy-Tailed Phenomena*)

Recent constraint solvers adopt this mechanism and follow some known sequences to compute the cutoffs (number of decisions, number of failures, etc) of each run:

- exponential sequence
- Luby sequence

From a run to the next one, the solver extracts nogoods corresponding to the explored spaces.

An illustrated example

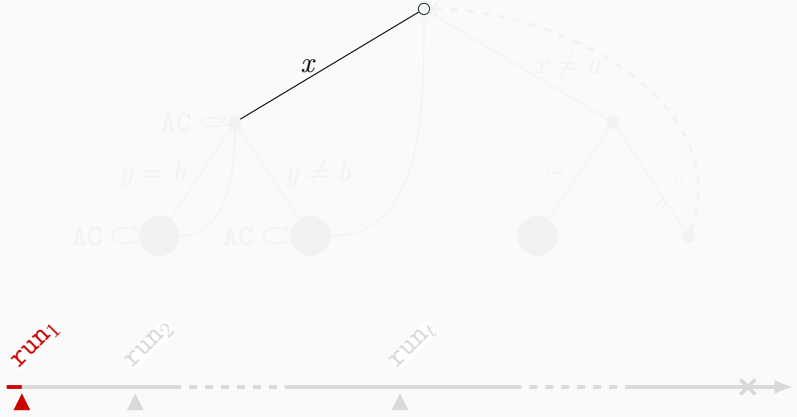
Global scheme : depth first binary tree search with backtracking

An illustrated example

1st run : root of the tree

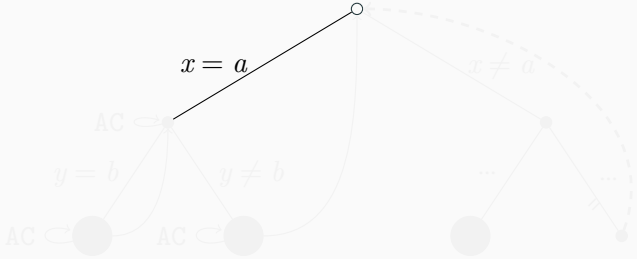
An illustrated example

Decision : the variable ordering heuristic selects x



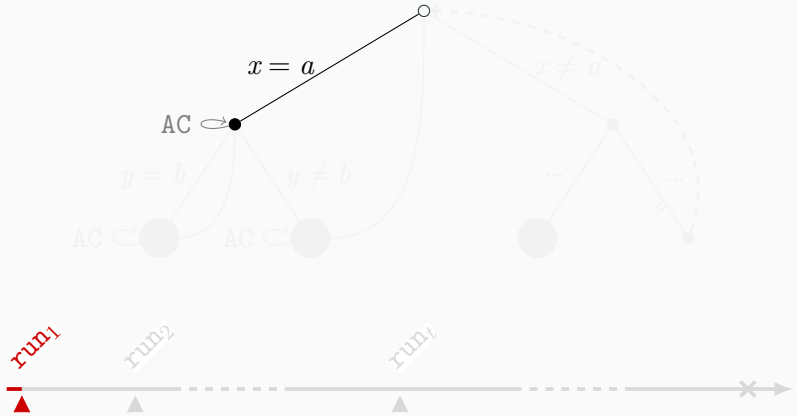
An illustrated example

Decision : the value ordering heuristic selects a



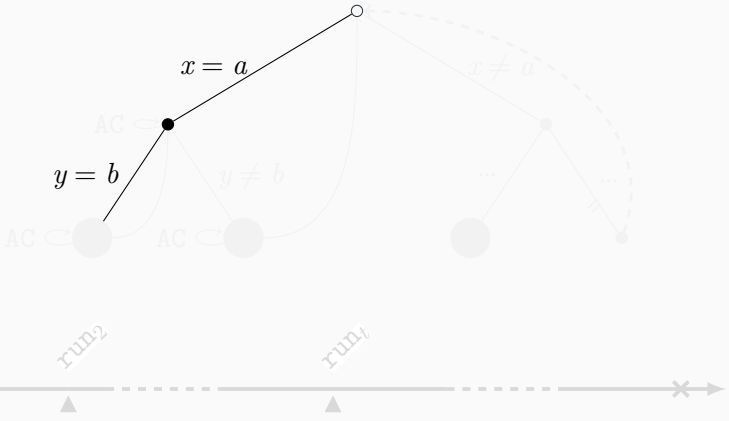
An illustrated example

Propagation : enforcing of the arc-consistency (AC) property



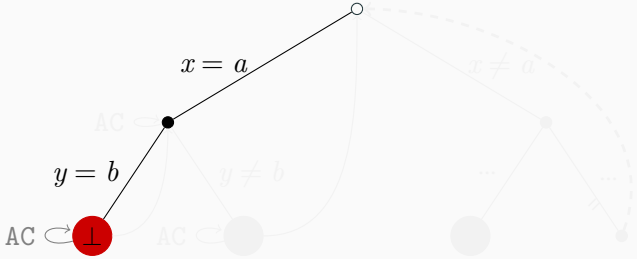
An illustrated example

Decision : next selection (variable, valeur)



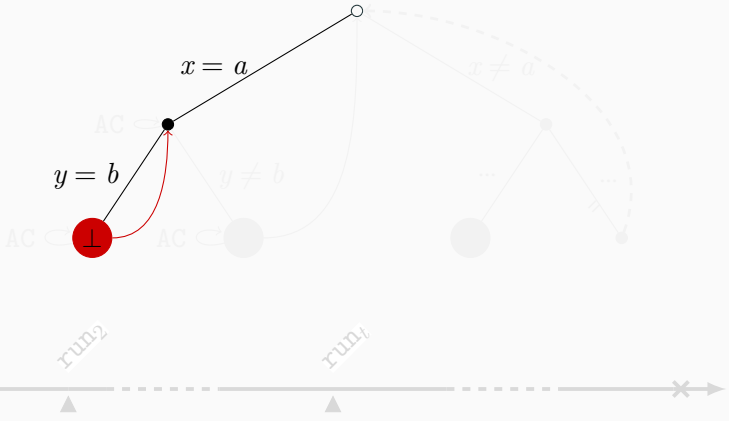
An illustrated example

Propagation : enforcing of the AC property and conflict



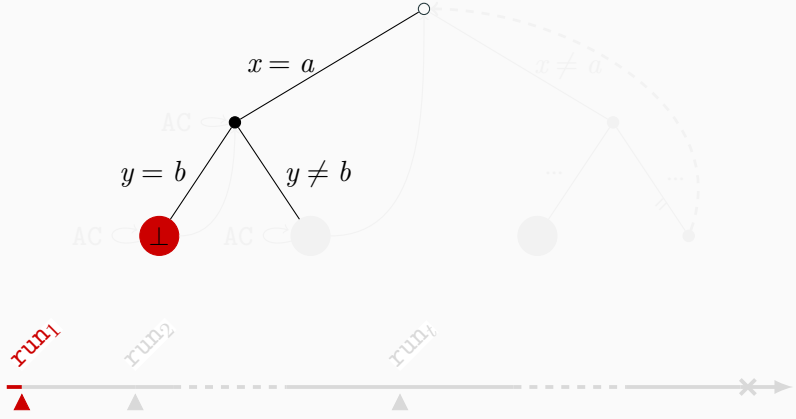
An illustrated example

Backtracking : parent node



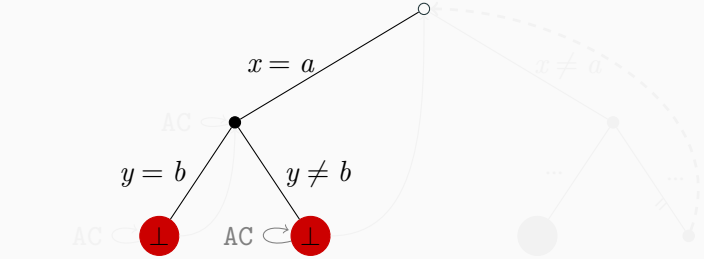
An illustrated example

Refutation : we consider $y \neq b$



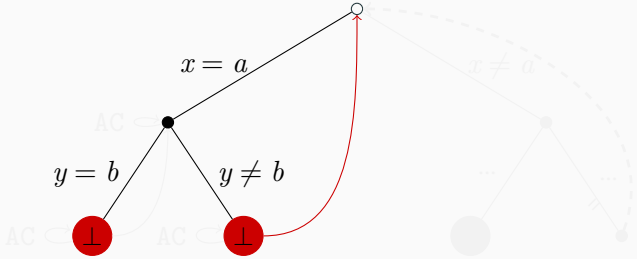
An illustrated example

Propagation : enforcing of the AC property and conflict



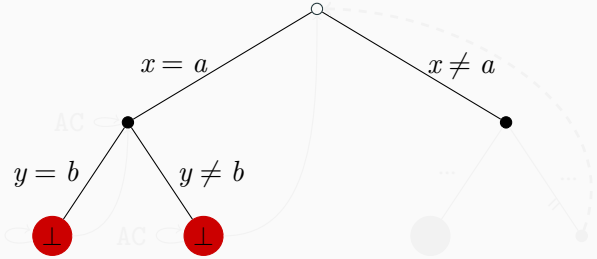
An illustrated example

Backtracking : parent node (root node)



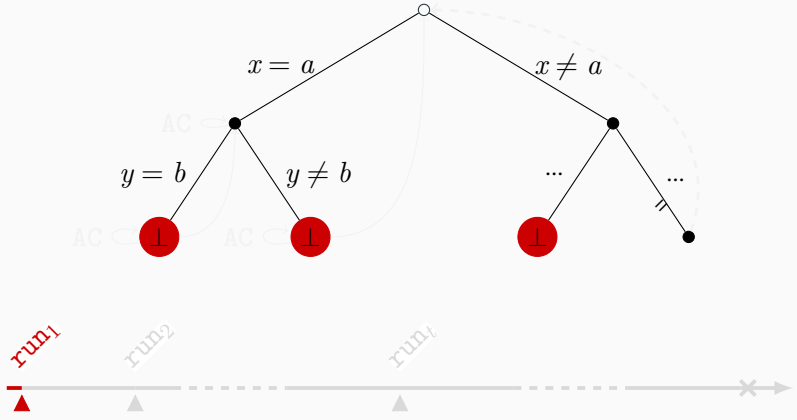
An illustrated example

Refutation : $x \neq a$



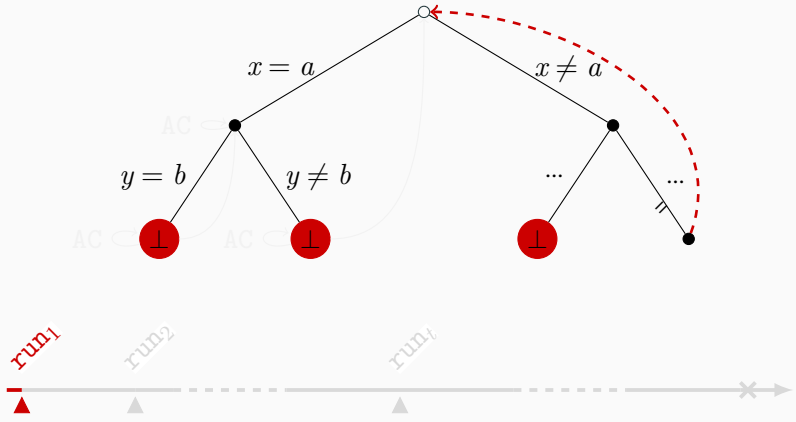
An illustrated example

Restarting : cutoff reached and nogood extraction



An illustrated example

Restarting : backtrack to the root node

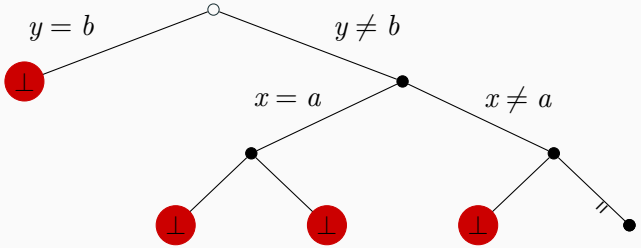


An illustrated example

2nd run : root node

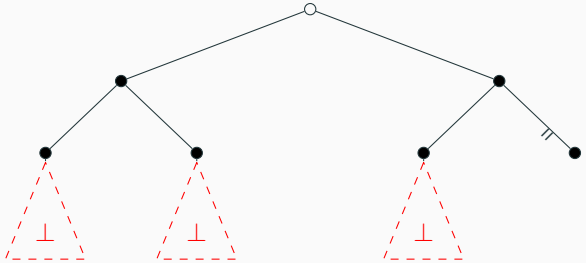
An illustrated example

2nd run : cutoff reached and restarting



An illustrated example

$\mathbf{t}^{\mathrm{th}}$ run : cutoff reached and restarting



An illustrated example

End of solving : satisfiability | unsatisfiability | timeout

Extension to the Optimization Problem

Constraint Optimization Problem (COP)

Definition (COP)

A *Constraint Optimization Problem* $\mathcal{P}$ is defined by :

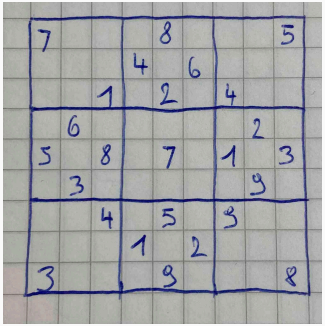
- a finite set of variables, denoted $\mathcal{X} = \mathtt{vars}(\mathcal{P})$;
- a finite set of constraints, denoted $\mathcal{C}$, such that $\forall c \in \mathcal{C}, \mathtt{scp}(c) \subseteq \mathcal{X}$;
- an objective function $\mathcal{O}$ to maximize or minimize

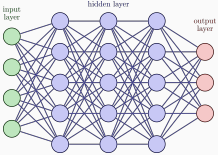
A solution for a COP

Definition (Solution)

A *solution* of a COP instance $\mathcal{P}$ corresponds to a solution of the underlying CSP problem. For minimisation (resp. maximisation), an *optimal solution* of $\mathcal{P}$ is a solution for which the objective value is less than or equal to (resp. greater than or equal to) the value of any other solution.

When the ML transforms the Sudoku to an optimization problem





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

Figure 1: NN identified a handwritten Sudoku with some errors (on 1 and 7 reading)

When the ML transforms the Sudoku to an optimization problem

The neural network produces some errors when translating 1 and 7 handwritten digits. Instead of asking the neural network to produce its preferred label for a given non-empty cell, we ask it to provide the confidence of each digit label for that cell. Thus, the new data are as follows:

Data: Let D is the set of clues where each clue $d \in D$ corresponds to a set of confidence percentages d_k (where $k \in 1..9$ and $d_k \in 0..100$) in the cell $(d_i, d_j) \in (1..9, 1..9)$.

Study problem: Sudoku under uncertainty (COP)

Data: Let D is the set of clues where each clue $d \in D$ corresponds to a set of confidence percentages d_k (where $k \in 1..9$ and $d_k \in 0..100$) in the cell $(d_i, d_j) \in (1..9, 1..9)$.

Problem: Let $\mathcal{P} = (\mathcal{X}, \mathcal{C}, \mathcal{O})$ where:

Study problem: Sudoku under uncertainty (COP)

Data: Let D is the set of clues where each clue $d \in D$ corresponds to a set of confidence percentages d_k (where $k \in 1..9$ and $d_k \in 0..100$) in the cell $(d_i, d_j) \in (1..9, 1..9)$.

Problem: Let $\mathcal{P} = (\mathcal{X}, \mathcal{C}, \mathcal{O})$ where:

$$\mathcal{X} = \begin{Bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,9} \\ x_{2,1} & x_{2,2} & \dots & x_{2,9} \\ \vdots & \vdots & \vdots & \vdots \\ x_{9,1} & x_{9,2} & \dots & x_{9,9} \end{Bmatrix} \text{ and } \texttt{dom}(x_{i,j}) = \{1, \dots, 9\} \ \forall \ i, j \in 1..9$$

Study problem: Sudoku under uncertainty (COP)

Data: Let D is the set of clues where each clue $d \in D$ corresponds to a set of confidence percentages d_k (where $k \in 1..9$ and $d_k \in 0..100$) in the cell $(d_i, d_j) \in (1..9, 1..9)$.

Problem: Let $\mathcal{P} = (\mathcal{X}, \mathcal{C}, \mathcal{O})$ where:

$$\mathcal{C} = \left\{ \begin{array}{ll} \texttt{all\_diff}(x_{i,1}, \ldots, x_{i,9}) & \forall\ i \in 1..9 \\ \texttt{all\_diff}(x_{1,j}, \ldots, x_{9,j}) & \forall\ j \in 1..9 \\ \texttt{all\_diff} \begin{pmatrix} x_{k,l} & x_{k,l+1} & x_{k,l+2} \\ x_{k+1,l} & x_{k+1,l+1} & x_{k+1,l+2} \\ x_{k+2,l} & x_{k+2,l+1} & x_{k+2,l+2} \end{pmatrix} & \forall\ k, l \in \{1, 4, 7\} \\ \cancel{x_{d_i,d_j} = d_v} & \cancel{\forall\ d \in D} \end{array} \right\}$$

Study problem: Sudoku under uncertainty (COP)

Data: Let D is the set of clues where each clue $d \in D$ corresponds to a set of confidence percentages d_k (where $k \in 1..9$ and $d_k \in 0..100$) in the cell $(d_i, d_j) \in (1..9, 1..9)$.

Problem: Let $\mathcal{P} = (\mathcal{X}, \mathcal{C}, \mathcal{O})$ where:

$$\mathcal{O} = \max \sum_{d \in D} \sum_{k \in 1..9} \mathbb{1}_{x_{d_i, d_j} = k} \times d_k$$

COP solving principle

An objective function could be divided into an objective o and a function f: $\mathcal{O} = (o, f)$.

A COP is interpreted as a CSP instance that is iteratively more constrained in function of the new solution s and the evaluated s bound B:

Data: COP $\mathcal{P} = (\mathcal{X}, \mathcal{C}, \mathcal{O})$ where $\mathcal{O} = (\max, f)$
Result: optimal solution s

1 $B \leftarrow -\infty$

2 do
3 | CSP $\mathcal{P}' = (\mathcal{X}, \mathcal{C} \cup (f > B))$
4 | $s \leftarrow \texttt{solve}(\mathcal{P}')$
5 | $B \leftarrow \texttt{eval}(s)$
6 while $s \neq \bot$

7 return s

Conclusion

What we retain

The constraint programming is divided into two parts

- modeling the CSP/COP problem (PyCSP, MiniZinc, ...)
- solving the instance problem (ACE, Choco, ...)

The constraint solvers are composed of many mechanisms:

- backtracking-search procedure
- inference/propagation algorithms
- branching heuristics
- restart mechanism
- ... and many others from the bibliography

Solving a COP instance corresponds to iteratively solve the subjacent CSP instances.

Some actual work...

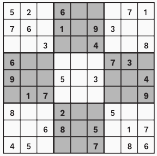
More efficient/autonomous CP solvers (due to wide combinations of their different mechanisms):

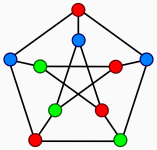
- portfolios: offline learning of the mapping between solver configuration and instance pattern
- multi-armed bandit: online learning of the best configuration
- ...

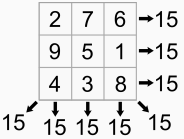
Hybridation:

- translate the instance to SAT/PB solvers with more specific solving mechanisms
- translate/share the instance to/with MILP solvers
- ...

~~Demotivation~~: not so much







... and many others!! 😁

An Introduction to Constraint Programming

Hugues Wattez

March 31, 2022

Tutorial at LIX

Abstract

Constraint programming is declarative: the user models the problem (in the form of constraints) without worrying about the solving part. The solving is delegated to a generic program called a constraint solver. For this introduction, we first present the modeling of a constraint satisfaction problem proposing the use of integer variables and a vast choice of constraints. Then, we describe the resolution mechanisms of modern constraint solvers: to guide their choices (heuristics) and to efficiently reduce the search space (inference). Finally, we see how this paradigm can be extended to optimization problems.