# Session 2
# Constraint Programming

Daniel Diaz
Salvador Abreu

# Modeling for Constraint Programming

*“Constraint Programming represents the Holy Grail of programming: the user states the problem, the computer solves it”*

Eugene Freuder, Director of the Cork Constraint Computation Centre

# Constraint Programming (CP)

- Family of methods for problem-solving.
- Allows the user to model a problem ***declaratively*** in terms of a state described with **variables** and **constraints**.
- Provides efficient ***constraint solvers*** to find a solution to the stated problem.
- In general, the solver is *complete*. Theoretically:
  - CP solvers are able to find all solutions of a problem (and thus the best ones in case of an optimisation problem)
  - If the problem is over-constrained, they discover it and report “no solution”

NB: in practice, this depends on problem size and complexity...

# Constraint Programming (CP)

Provides tools for modeling & solving, based on the concepts:

- **Variables** (the unknowns of the problem)
- **Domains** (values they may take)
- **Relations** among variables (aka *constraints*)

Variables take values which collectively specify a **state** of the system being described.

Some variables may be **defined** (have a specific value) while others may have an as yet unknown value, albeit restricted to a set of candidate values (their **initial domain**).

Some states are **admissible** while others are not.

# What is a constraint?

- A constraint is a relation applied to a subset of the variables of the problem (each taking a value in its own domain).
- The role of a constraint is to **restrict** the values these variables can simultaneously take. It represents a partial information about some variables. Ex: paper sheet: Height / Width = $\sqrt{2}$
- It can be expressed **extensionally** as a set of possible tuples.

  and(X,Y,Z) = { <0,0,0>, <0,1,0>, <1,0,0>, <1,1,1> }
- It is more often expressed **intentionally** as a formula:

  $X + Y \leq 2$

  which is more concise than the set of possible tuples for <X,Y>:

  { <0,0>, <0,1>, <0,2>, <1,0>, <1,1>, <2,0> }

# Properties of constraints

Constraints enjoy several interesting properties:

- Specify **partial information**: a constraint needs not uniquely specify the values of its variables
- **Non-directional**: a constraint on X and Y can be used to infer a value for X given a value for Y and vice versa
- **Declarative**: a constraint specifies *what* relationship must hold without specifying *how* to enforce that relationship.
- **Additive**: the order of imposition of constraints does not matter (at the end, the conjunction of all constraints is in effect).

# Different Constraint Systems

Constraint Programming applies to a wide variety of constraint systems. For instance it is possible to solve problems expressed with constraints on:

- **Finite integers (discrete problems)**
- **Booleans**
- **Sets**
- Character strings
- Real numbers (continuous problems)
- Graphs and trees

In this lecture we are mainly interested in **discrete problems**, often a good fit for Constraint Satisfaction Problems over **Finite Domains**, also known by the acronym **FD**.

# Some Applications of FD

- Planning (finding activities to achieve a given goal)
- Scheduling (allocating known activities to limited resources and time)
- Routing problems (TSP, VRP,...)
- Operations Research problems (optimisation problems)
- Circuit design (to compute layouts)
- Electrical engineering (to locate faults)
- Airline crew rostering
- Datacenter network layout

# Some Applications of FD

- DNA sequencing
- Protein docking
- Resource management (forests, HR, …)
- Virtual Reality (to express coherence in VR scenes)
- Natural language processing (construction of efficient parsers)
- Business applications (option trading)
- (Software) Product Line Engineering
- Air Traffic Control
- … and many others ...

# Companies using CP Technology

Several companies and public entities make use of the advantages of constraint technology:

- SNCF (French railway authority)
- RATP (Paris metropolitan subway authority)
- Renault
- Michelin
- Dassault
- Airports, including: Orly, CDG and Hong-Kong
- Airlines, including: British Airways, SAS, Swissair
- REN (Portugal electric power distribution)
- ...

# History

Prolog as a vehicle for Logic Programming.

Negation as failure: convenient but accident-prone :)

- Difficulties when variables still unbound

Early to mid-1980s: Prolog variants with “delay” or “freeze” constructs

- Prolog-II (Marseille, 1982)
- Mu-Prolog (Melbourne, 1985)
- ...

The part being “frozen” became **increasingly complex** (e.g. a goal). It always “hangs” on a variable (initially unbound).

# History

1987: Jaffar and Lassez propose the Constraint Logic Programming (CLP) paradigm which extends LP (Prolog) by replacing unification with constraint solving over some domain X.

1988-1991: several CLP systems appear:

- BNR-Prolog: interval arithmetic constraints
- Prolog III: constraints over strings, booleans and reals (linear).
- CLP(R): real arithmetic
- CHIP: constraints over Finite Domains (integers)

CHIP successfully solves several hard real-life problems (e.g. car-sequencing). However, CHIP is a commercial product and the authors enshroud the solver in opaque mystery

# History

CHIP: the solver is a **black box**

1992: Pascal Van Hentenryck proposes a "**glass-box** approach" to develop solvers. This theoretical framework uses simple primitives on which complex constraints can be built.

1993: **clp(FD)**: first implementation to adopt the glass-box approach. Proves that an efficient solver can be built with little effort.

From 1995: CP extends beyond LP: others languages propose solvers (e.g. via libraries). CP becomes an independent research area which is identified by ACM as a "strategic direction" in Computing Science (1998).

# Modeling as Constraint Satisfaction Problems

# Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is defined as a tuple

$$\mathbf{P = (V, D, C)}$$

where:

- **V** is a set of *variables* $V_1,...,V_n$ (the unknowns of the problem)
- **D** a set of *domains* $D_1,...,D_n$ (one domain per variable)
- **C** is a set of *constraints* $C_1,...,C_m$ (relations between variables)

Each variable $V_i$ can only take values in its domain $D_i$.

Such a problem can also be stated as a logic formula:

$$\mathbf{\exists\ (V_1,V_2,...,V_n) \in (D_1 \times D_2 \times ... \times D_n)\ /\ C_1 \wedge C_2 \wedge \ldots \wedge C_m}$$

# Constraint Satisfaction Problems over Finite Domains

When the domains of the variables are **finite** sets of values (constants), we say the CSP is over **Finite Domains** (FD, for short).

For convenience we shall only consider positive integer values.

$$D_i = \{0, 1, \ldots k\} \text{ where } k \in \mathbb{N}$$

This is done without loss of generality (other problems over FD may be mapped onto these).

# A simple example

Consider a tray with 5 slots, numbered 0 to 4 and laid out left-to-right.

Consider also that we have 5 numbered tokens, with labels in the range 1 to 13 (as integers).

Now let's consider that:

- In each slot we may place at most one token
- If the token labeled **i** is in slot **j**, then:
  - All tokens to its left (slots < j) must be labeled less than or equal to i
  - All tokens to its right (slots > j) must be labeled greater than or equal to i

This amounts to saying that the tokens are *sorted*.

These conditions may be construed as **constraints**.

# A simple example

We model this as follows:

- The token in slot i is represented by constraint variable $V_i$
- The domain of $V_i$ is {1, 2, … 13}, for all i in 0..4
- The following conditions must hold
  - $V_0 \leq V_1$
  - $V_1 \leq V_2$
  - $V_2 \leq V_3$
  - $V_3 \leq V_4$

  These are the **constraints** for our CSP.

The intention is to describe a card hand, with 5 cards, in which the cards are sorted in increasing order.

# A simple example

Solving the problem amounts to finding the tuples which satisfy the constraints (hence the term ***constraint satisfaction***)

We can go about this task in many ways, but usually it will be a **search process**, as we normally don't have an algorithm to produce the solutions.

In the previous case, the following are some solutions:

3, 3, 6, 10, 13 2, 5, 8, 11, 12

But the following are **not**:

8, 9, 10, 11, 3 7, 6, 5, 4, 0

# Solution of a CSP

A solution of a CSP P=(V,D,C) is a complete **consistent** *instantiation* of variables, ie. an *assignment* of the variables s.t.:

- Each variable $V_i$ is assigned a value in $D_i$ (its domain)
- All constraints in C are satisfied (for each constraint $C_j$, the values of involved variables form an admissible tuple of $C_j$)

Example: CSP = ({X,Y, Z}, {1..3, 1..3, 1..3}, { X+Y = Z })

admits 3 solutions :

X = 1, Y = 1, Z = 2
X = 1, Y = 2, Z = 3
X = 2, Y = 1, Z = 3

# Constraints on integers

In practice, wide variety of constraints over Finite Domains:

- Arithmetic (linear and NOT linear):

  $=, \neq, <, \leq, >, \geq$ (using +, -, *, /, min(), max(), abs(), ...)

- Symbolic: (most are *global constraints*)

  `allDifferent`($X_1$,...,$X_n$) enforces all Xi are different

  `element`(I, {$X_1$,...,$X_n$}, V) enforces $X_I = V$

  `atmost`(N, {$X_1$,...,$X_n$}, V) enforces at most N variables = V

# Constraints on booleans

Booleans can be seen as a special case of integers:

- Consider false=0, 1=true

  $\wedge$ (and), $\vee$ (or)

  $\veebar$ (xor), $\Leftrightarrow$ (equiv)

  (imply) $\Rightarrow$

  $\neg$ (not), etc…

- A reified constraint “captures” its truth value in a (Boolean) variable (which detects when the store *entails* a constraint)

  $X > Y \Leftrightarrow B$  enforces B=1 iff X > Y

  (propag. 4 cases, e.g. as soon as X > Y is true B is set to 1)

# Combinatorial Optimisation Problems

A Combinatorial Optimisation (CO) consists of finding an *optimal* solution from a finite (discrete) set of candidate solutions.

The set of possible solutions can be describe with a CSP (V,D,C).

The optimality criterion is stated as an *objective function* $\mathcal{Z}$ over the variables of V which must be either *minimised* or *maximised*.

COP = CSP + $\mathcal{Z}$

Solving a COP consists in finding a solution to the underlying CSP which minimises (or maximises) $\mathcal{Z}$.

# Exercise: CSP **Modeling**

**Model** the following problems:

1) **TWO + TWO = FOUR**
2) **4-queens, generalize to N-queens**
3) **Magic squares 4**
4) **Vending machine**
5) ~~Find stable marriage(s)~~

# Searching for a solution

Sometimes, we just **know** how to get to it.

Typically this is because we have an algorithm that computes solutions to the problem.

Very often, though, we **don't know how to get** to a solution, but we do **know how to tell** whether a candidate configuration is indeed a solution.

Since the domain of each variable is finite, the set of all possible *solution candidates* is also finite. This set forms the "*search space*".

It is natural to consider an exhaustive exploration of the search space, e.g. using *brute-force algorithms*...

# Combinatorial Explosion

Unfortunately, in many problems, the size of the search space rapidly grows w.r.t. the size of the problem (e.g. exponentially)

⇒ naive exhaustive search **unusable**

Example: magic square

The search space of a Problem N is (NxN) !

*(Note that factorial grows faster than any exponential function)*

| N | Search space size |
|---|---|
| 10 | $100 ! \approx 10^{158}$ |
| 25 | $625 ! \approx 10^{1475}$ |
| 50 | $2\,500 ! \approx 10^{7412}$ |
| 100 | $10\,000 ! \approx 10^{35660}$ |
| 200 | $40\,000 ! \approx 10^{166714}$ |
| 400 | $160\,000 ! \approx 10^{763175}$ |

# How does CP work?

Hopefully, **constraint solvers** know how to solve the problem.

They include clever (?) algorithms which avoid an exhaustive search by **pruning** the search space.

Solvers combine:

- **inference**: removing values violating constraints each time a new constraint is added (consistency techniques + propagation).
- **search**: try possible combinations of values & follow depth-first search or another strategy

# Searching the (candidate) solution space

Every combination of values for the variables in a problem is a *candidate* solution.

Some will turn out to be **actual solutions** (i.e. they satisfy all constraints)

Most will **not be solutions** (i.e. they violate some constraints)

The size of the solution space tends to be very large, so…

… we need to be smart about how to search in it!

# Programming with constraints: posting constraints

The user program ***posts*** constraints to the ***constraint store***.

The result of this operation can be:

- **Fail**: in case the problem has no solution (overconstrained)
- **OK**: the store is consistent (other constraints can be posted)

For the programmer, a solver is an abstraction which collects the state of the exploration of a CSP.

Internally, the solver shrinks the domain of involved variables (removing impossible values) until a fix-point is reached. This phase is called *consistency*, which is based on *filtering+propagation* techniques.

If the domain of a variable becomes empty the solver returns Fail.

# Programming with constraints: consistency

Posting a constraint triggers a consistency checking algorithm whose goal is to remove impossible values from the domains of variables.

Ex1: X < Y, $D_X$=$D_Y$={1,2}, after consistency: X=1  Y=2 [solution]

Ex2: X < Y, $D_X$=$D_Y$={1,2,3}, after consistency: $D_X$={1,2,~~3~~}  $D_Y$={~~1~~,2,3}

Notice that reduced domains are an ***approximation*** of the set of solutions:

- All solutions are composed of values in the reduced domains.
- Not all tuples of values of the reduced domains form a solution.
- It often happens that a value in a domain **does not even belong to any** solution !

# Programming with constraints: consistency

Different consistency algorithms exist which vary w.r.t. their precision (i.e. removing more or less invalid values). In general, the more precise they are, the more time-demanding they are.

In any case, the consistency algorithms are **not complete**, i.e. they do not remove all invalid values.

Fully solving the consistency problem, to have a complete consistency mechanism would be NP-complete and thus unusable in practice!

We will now see a few important consistency algorithms.

# Programming with constraints: arc-consistency

When a constraint C is posted, the arc-consistency algorithm (AC) checks all variables involved in C. For each variable X, it considers each value V in $D_X$, and checks that a compatible value V' exists in the domain of all other variables (i.e. a combination of values satisfying C). If it is not the case, V is removed from $D_X$.

$D_X$ = {1,2,3,4,6,~~7~~}  posted constraint : X + 3 = Y
$D_Y$ = {~~1,2,3~~,4,5,6,7,~~8~~,9}

Arc-consistency can create and propagate "**holes**" in the domains.

When a value V is removed from $D_X$, all constraints C' which involve X must be reconsidered. This can lead to new value removals.

This is done until a fixpoint is reached. (i.e. no more domains get modified)

# Programming with constraints: bound-consistency

AC can be very costly in practice. With the bound-consistency algorithm (BC), when a variable X is reconsidered, only the consistency of its Lower (LB) and Upper Bound (UB) are checked.

This equates to approximating the domain $D_X$ of a variable X with the interval [LB..UB] (as if there were no holes).

$D_X$ = {1,2,3,4,6,~~7~~} posted constraint : X + 3 = Y

$D_Y$ = {~~1,2,3~~,4,5,6,7,**8**,9}

BC does not propagate holes.

In practice BC is often preferred to AC but in some cases a better pruning is desirable. Some systems allow the user to choose (for some constraints) the consistency algorithm to use.

# Programming with constraints: consistency (summary)

Posting a constraint triggers a consistency algorithm whose goal is to remove impossible values from the domain of variables.

**AC**: checks all values in the domains (propagate “holes”)

**BC**: only ensures the bounds are “consistent” (does not propagate “holes”)

**Fix-point procedure**: when a value is excluded from a variable, reconsider al constraints on this variable., etc...

**Incomplete procedure**: domains are approximations of actual solutions (not all values are part of a solution).

How to find actual solutions ? ...

**… by enumeration (a trial-and-error process).**

# Programming with constraints: search

When all constraints have been posted a *search phase* may be triggered (this phase is also called *labeling* or *enumeration*). The goal is to find a value for all variables.

- For each variable X, each value C in its (reduced) domain is tried (by **temporarily** posting a constraint X = C to the store)
- This is iterated over the remaining variables, so as to explore *all* remaining possible bindings.

Binding a variable X to a specific value will **trigger propagation**, to enforce consistency w-r-t other variables which are related to X.

# Programming with constraints: search

Different search strategies (with varying **performance** and resulting tordering) exist for:

- Several possibilities for the order in which variables are tried:
  chronological, first-fail,...
- For a given variable, the order in which its values are tried
  from min to max, from max to min, from middle, random,...

These strategies define the search tree which is explored by the labeling phase (usually depth-first, left-to-right).

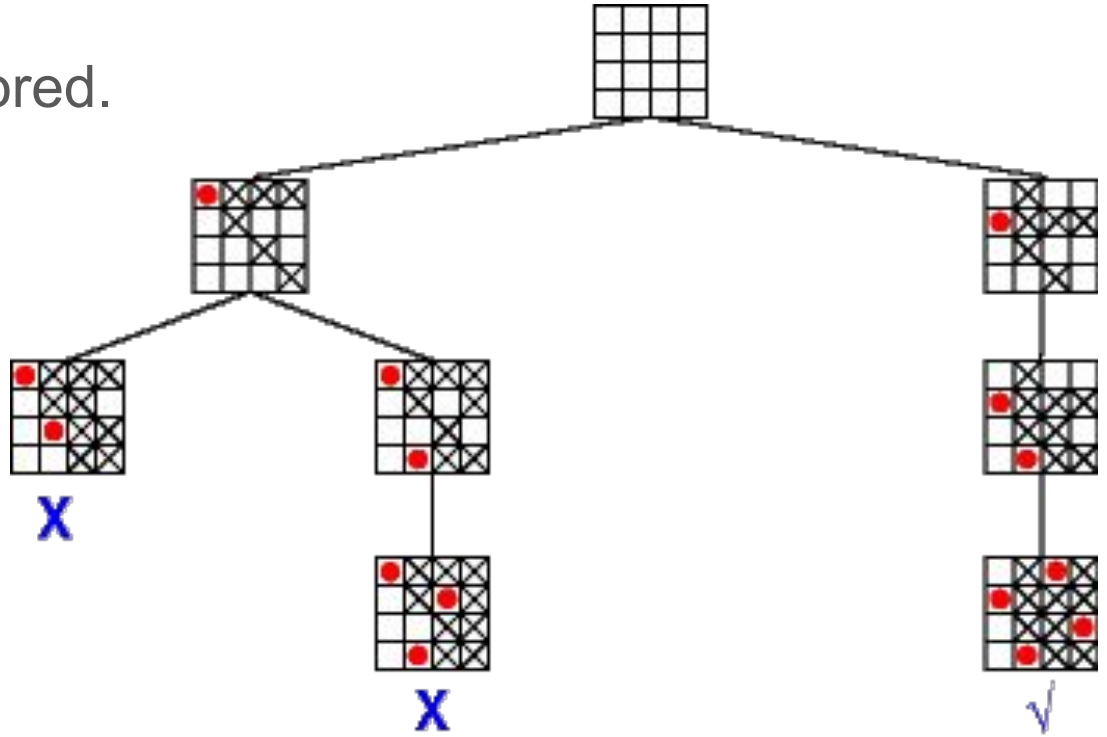
# Programming with constraints: search

The (remaining) search space can be seen as a tree: the ***search tree***.

This tree must be explored.

e.g.: depth-first search

Example: 4-queens



# Exercise: CSP **Modeling**

**Model** the following problems:

1) SEND + MORE = MONEY
2) 4-queens, generalize to N-queens
3) Magic squares N
4) Vending machine
5) Stable matching

For all of the above problems, you must model them as a CSP (or a COP if applicable)

# Exercise: vending machine

Problem statement:

Work out the change C to be given out by a vending machine, knowing that the user inserts the amount A (in Eurocent, or just c) to pay for a drink that costs D c.

Problem information:

- The machine takes and returns Euro coins, no less than 5c, i.e. 2€, 1€, 50c, 20c,10c and 5c.
- The machine is loaded with a number N of each type of coin
- C is an array indexed by the coin types (i.e. we have C[2E], C[1E]… C[5c])
- Extend the problem to a COP, i.e. minimise the number of coins

# Exercise: Stable Matching

5 men and 5 women with their preferences:

| Men's preferences | Women's preferences |
|---|---|
| 1: 1, 2, 4, 3, 5 | 1: 2, 3, 5, 4, 1 |
| 2: 3, 4, 1, 5, 2 | 2: 2, 4, 3, 1, 5 |
| 3: 4, 3, 5, 2, 1 | 3: 5, 3, 2, 4, 1 |
| 4: 1, 5, 2, 4, 3 | 4: 1, 5, 4, 3, 2 |
| 5: 5, 2, 3, 1, 4 | 5: 4, 3, 2, 1, 5 |

Lower number indicates higher preference (i.e. it's a rank)

Find a ***stable*** marriage, i.e. a list of (m,w) pairs where no man-woman pair occurs in which **both** would rather marry each other than their assigned partner.

# Stable Matching: modeling hints

- Represent **Wi** as the wife of a given man **i** and **Hj** as the husband of woman **j**. All **Wi** and **Hj** will range over **1..n**.
- Define a ranking which tells us the rank, for man **i**, of woman **j**. Conversely indicate the ranking for woman **i**, of man **j**. Do this for all applicable **i** and **j**.
- Define “integrity constraints” (mutual exclusion, reciprocity, monogamy)
- State the stability condition, both from the point of view of the men and of the women.
    - Hint: if man m prefers other woman o to his wife, then o must prefer her husband to m (and conversely for woman w and other man o)