DM87
SCHEDULING,
TIMETABLING AND ROUTING

Lecture 7

# Constraint Programming in Practice

Marco Chiarandini

## Outline





## Outline





## Constraint Programming Systems

CP systems must provide reusable services for:

- Variable domains
  finite domain integer, finite sets, multisets, intervals, ...
- Constraints
  distinct, arithmetic, scheduling, graphs, ...
- Solving
  propagation, branching, exploration, ...
- Modelling
  variables, values, constraints, heuristics, symmetries, ...

## CP modelling

Greater expressive power than mathematical programming

- constraints involving disjunction can be represented directly
- constraints can be encapsulated (as predicates) and used in the definition of further constrains

However, CP models can often be translated into MIP model by

- eliminating disjunctions in favor of auxiliary Boolean variables
- unfolding predicates into their definitions



## CP System Interfaces

Two possible interfaces:

- host language
- libraries



## Modelling Language

- Fundamental difference to LP
  - language has structure (global constraints)
  - different solvers support different constraints
- In its infancy
- Key questions:
  - what level of abstraction?
    - solving approach independent: LP, CP, ...?
    - how to map to different systems?
  - Modelling is very difficult for CP
    - requires lots of knowledge and tinkering



## Modelling Languages

- Prolog
  - B-Prolog (Prolog based, proprietary)
  - CHIP V5 (Prolog based, also includes C++ and C libraries, proprietary)
  - Ciao Prolog (Prolog based, Free software: GPL/LGPL)
  - ECLiPSe (Prolog based, open source)
  - SICStus (Prolog based, proprietary)
  - GNU Prolog
- OPL
- Zinc, MiniZinc, FlatZinc

## CP Systems

- Library-based
  - CHOCO (free) `http://choco.sourceforge.net/`
  - Kaolog (commercial) `http://www.koalog.com/php/index.php`
  - Gecode (free) `www.gecode.org`
    Programming interfaces Java and MiniZinc, library C++



## CP Systems

- Language-based
  - SICStus Prolog (commericial) `www.sics.se/sicstus`
    Prolog language, library
  - ECLiPSe (free) `www.eclipse-clp.org`
    Prolog language, library
  - Mozart (free) `http://www.mozart-oz.org`
    Oz language
  - ILOG CP Optimizer `http://www.ilog.com/products/`
    OPL Language, libraries C/C++/
  - CHIP (commercial) `http://www.cosytec.com`
    Prolog language, library C/C++
  - G12 Project `http://www.g12.cs.mu.oz.au/`



## Outline





## Solving CP

- Compute with possible values
  rather than enumerating assignments
- Prune inconsistent values
  constraint propagation
- Search
  branch: define search tree
  explore: explore search tree for solution
  branching heuristics
  best solution search (in optimization)

## Propagators

CP Systems do not compute constraints extensionally (as a collection of assignments):

- impractical (space)
- would make difficult to take advantage of structure

A Constraint $c$ is implemented by a set of propagators (also known as filtering algorithms and narrowing operators).

A propagator $p$ is a function that maps domains to domains. They are decreasing and monotonic.

A set of propagators implements a constraint $c$ if all $p \in P$ are correct for $c$ and $P$ is checking for $c$. Notation: $P = \text{prop}(c)$



## Execution of Propagators

- Execution of propagator $p$
  - narrows domains of variables in var(p)
  - signals failure
- Execution computes largest simultaneous fixpoint
  - fixpoint: propagators cannot narrow any further
  - largest: no solutions lost
- Propagator is either
  fix: has reached fixpoint
  runnable: not known to have reached fixpoint
- Propagation execution maintains propagator sets
- Propagators know their variables
  - to perform domain modifications
  - passed as parameters to propagator creation
- Variables know dependent propagators
  - to perform efficient computation of dependent propagators



## Global Constraints

- Classic example: $x, y, z \in \{1, 2\}, \quad x \neq y, x \neq z, y \neq z$
- No solution!
- But: each individual constraint still satisfiable!
  no propagation possible!
- Solution: look at several constraints at once
  distinct(x,y,z)
- Specialization



## Kinds of symmetries

- Variable symmetry:
  permuting variables keeps solutions invariant
  $\{x_i \to v_i\} \in sol(P) \Leftrightarrow \{x_{\pi(i)} \to v_i\} \in sol(P)$
- Value symmetry: permuting values keeps solutions invariant
  $\{x_i \to v_i\} \in sol(P) \Leftrightarrow \{x_i \to \pi(v_i)\} \in sol(P)$
- Variable/value symmetry:
  permute both variables and values
  $\{x_i \to v_i\} \in sol(P) \Leftrightarrow \{x_{\pi(i)} \to \pi(v_i)\} \in sol(P)$

## Symmetry

- inherent in the problem (sudoku, queens)
- artefact of the model (order of groups)

How can we avoid it?

- ... by model reformulation (eg, use set variables,
- ... by adding constraints to the model (ruling out symmetric solutions)
- ... during search
- ... by dominance detection



## Reified constraints

- Constraints are in a big conjunction
- How about disjunctive constraints?

$$A + B = C \quad \vee \quad C = 0$$

- Solution: reify the constraints:

$$\begin{array}{l} (A + B = C \quad \Leftrightarrow \quad b_0) \quad \wedge \\ (C = 0 \quad \Leftrightarrow \quad b_1) \quad \wedge \\ (b_0 \quad \vee \quad b_1 \quad \Leftrightarrow \quad true) \end{array}$$



## Scheduling Models

- Variable for start-time of task a start(a)
- Precedence constraint:
  $start(a) + dur(a) \leq start(b)$ (a before b)
- Disjunctive constraint:
  $start(a) + dur(a) \leq start(b)$ (a before b)
  or
  $start(b) + dur(b) \leq start(a)$ (b before a)
  Solved by reification
- Cumulative Constraints (renewable resources)
  For tasks a and b on resource R
  $use(a) + use(b) \leq cap(R)$
  or $start(a) + dur(a) \leq start(b)$
  or $start(b) + dur(b) \leq start(a)$



## Propagators for Scheduling

Serialization: ordering of tasks on one machine

- Consider all tasks on one resource
- Deduce their order as much as possible
- Propagators:
  - Timetabling: look at free/used time slots
  - Edge-finding: which task first/last?
  - Not-first / not-last

## Job Shop Problem

- Hard problem!
- 6x6 instance solvable using Gecode
  - disjunction by reification
  - normal branching
- Classic 10x10 instance not solvable using Gecode!
  - specialized propagators (edge-finding) and branchings needed



## References

- Lecture notes by Christian Schulte for courses at KTH, Sweden
- Lecture notes by Marco Kuhlmann and Guido Tack for courses at Saarland University



## Outline





## Exercise

Write a MiniZinc model for the instance of Resource Constraint Project Scheduling Problem and solve the instance made available.

An installation of `minizinc-0.7` might be sufficient (uses G12 to solve the problem)

```
> mzn2fzn --data rcpsp.data rcpsp.mzn
> flatzinc jobshop.fzn
```

Otherwise, it is possible to use the interface `gecode-flatzinc-1.1` for `gecode-2.0.1`

```
> mzn2fzn --data rcpsp.data rcpsp.mzn
> fz jobshop.fzn
```