Session 3
Constraint Solvers

Daniel Diaz
Salvador Abreu

Many available solvers

Open-source solvers:

- C++: Gecode, Google or-tools (native)
- C#: or-tools (using its C# interface)
- Java: **Choco**, JaCoP, or-tools (using its **Java** interface)
- Python: or-tools (using its Python interface), Numberjack
- Scala: OscaR, JaCoP (using its Scala interface)
- Prolog: GNU Prolog, SICStus Prolog, ECLiPSe CLP,...
- CHR systems (Prolog-based)

Proprietary solvers:

IBM CPLEX Studio which includes CP optimizer (formerly ILOG solver) in addition to IP and MIP

Some solvers

Besides the mentioned solvers, there are tools / frameworks which make use of several different solvers

- Independently or
- Together, as a portfolio

These are the modeling language systems, which we'll come back to.

Beware: some solvers may be sort of slow...

A constraint solver for Java: Choco (version 4)

- Developed at Ecole des Mines de Nantes (France)
- Open (online source repository, BSD license)
- Readable and flexible (designed for teaching and research)
- Efficient and reliable (solves real world problems)
- Industrial companies: Safran, Dassault, PSA
- Research agencies: ONERA, NASA
- Software and Integrators: Kls-Optim, Easyvirt, alfaplan GmbH, Hedera Technology, etc

http://www.choco-solver.org/

The Choco Solver

Several variable paradigms:

Integer, **Boolean**, **Set**, Real, Graph

Several (~100) built-in constraints:

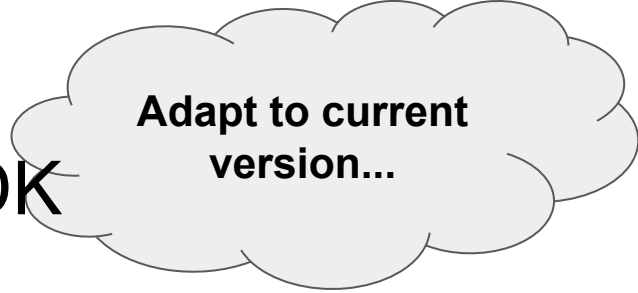
- Classical arithmetic constraints: =, ≠, <, ≤, >, ≥
- Global constraints: allDifferent, element, globalCardinality, nValue, cumulative, diffN, occurrence, regular, circuit, ...
- Reified constraints: any constraint can be reified

Choco 4 user guide: http://choco-solver.readthedocs.io/en/latest

Choco 4 api: http://www.choco-solver.org/apidocs/index.html

Choch 4 tutorial: http://choco-tuto.readthedocs.io/en/latest/

Using Choco 4 (4.10.6) with plain JDK



Using a JDK (version 8 or later), grab choco-4.10.6.zip from https://github.com/chocoteam/choco-solver/releases/latest

It contains several files, of interest:

choco-4.10.6.jar	*The classes, to link to apps*
4.10.6.tar.gz	*the source code*
User guide	*documentation*

Include **choco-solver-4.10.6.jar** in the CLASSPATH variable, when compiling and running your program.

Using Choco 4 with **NetBeans** (4.10.2)

Download the Choco zip file and unzip it (e.g. in your home).

Under NetBeans (NB), add the choco Library:

- In Tools|Library choose “New Library” then type “Choco 4”.
- In the left Library list, select “Choco 4”, In the right window:
  - In the Classpath tab, select Add JAR/Folder and select the file: choco-solver-4.10.2.jar
  - Similarly you can add the sources and the javadoc (apidocs-4.10.2.zip) in corresponding tabs.

To create a project using Choco:

- In the Project view, right-click on Libraries, select Add Library and then select “Choco”
  (a line “Choco 4.10.2 - choco-solver-4.10.2.jar” should appear in the list of libraries)

Using Choco with another IDE

Basically include the **choco-solver-4.10.2.jar** in the project (or just in the CLASSPATH).

Should work out-of-the-box with IntelliJ IDEA, Eclipse, Netbeans and others.

Programming with Choco

1) Create a **model**:

```
Model model = new Model("my model name");
```

2) Declare the variables and their domain:

```
IntVar x = model.intVar("name", min, max); ...
```

3) Create and ***post*** (activate) the constraints:

```
model.<constraint>.post(); ...
```

4) Create a **solver**:

```
Solver solver = model.getSolver();
```

5) **Search** for a solution (repeat to obtain several solutions):

```
solver.solve();          // returns a boolean
Or solver.findSolution(); // (next) Solution
```

Choco API: some basic arithmetic constraints

`arithm(IntVar x, String op1, IntVar z)`
`arithm(IntVar x, String op1, IntVar y, String op2, IntVar z)`

- One **`op`** ∈ {"=", "!=", ">", "<", ">=", "<="} the other **`op`** ∈ {"+", "-"}
- Parameter **`z`** can be an integer
- Enforce: **`x op1 z`** **`x op1 y op2 z`**
- e.g. **`arithm(x, "!=", y)`**, **`arithm(x, "<=", y, "+", 3)`**

`allDifferent(IntVar[] vars)` (i.e. also accepts **`IntVar... vars`**)

- Enforce **`vars[i]`** ≠ **`vars[j]`** for all **`i`** ≠ **`j`**

`scalar(IntVar[] vars, int[] coeff, String op, IntVar z)`

- **`op`** ∈ {"=", "!=", ">", "<", ">=", "<="}
- Parameter **`z`** can be an integer
- Enforce: **`coef[0]`** × **`vars[0]`** + **`coeff[1]`** × **`vars[1]`** + … +
  `coef[n-1]` × **`vars[n-1]`** **`op z`**

A simple Choco Example

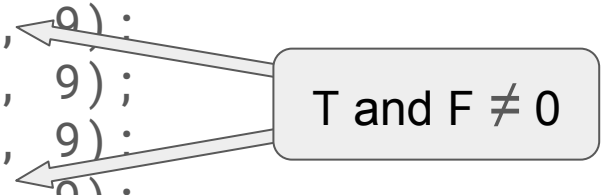
```
    T W O
+   T W O
-------
= F O U R
```

```
public static void main(String[] args) {

    Model model = new Model("TWO+TWO=FOUR");

    IntVar T = model.intVar("T", 1, 9);
    IntVar W = model.intVar("W", 0, 9);
    IntVar O = model.intVar("O", 0, 9);
    IntVar F = model.intVar("F", 1, 9);
    IntVar U = model.intVar("U", 0, 9);
    IntVar R = model.intVar("R", 0, 9);

    model.allDifferent(T, W, O, F, U, R).post();
```



A first Choco Example

```
    IntVar[] vars = new IntVar[]{
                  T,  W,  O,
                  T,  W,  O,
              F,  O,  U,  R};
    int[] coeffs = new int[]{
                100, 10,  1,
                100, 10,  1,
        -1000,-100,-10, -1};

    model.scalar(vars, coeffs, "=", 0).post();
    Solver solver = model.getSolver();
    System.out.println(solver.findSolution());
}
```

```
      T W O
+     T W O
-------------
= F O U R
```

TWO+TWO = FOUR Java code

```
public static void main(String[] args) {
    Model model = new Model("TWO+TWO=FOUR");

    IntVar T = model.intVar("T", 1, 9);  // T != 0
    IntVar W = model.intVar("W", 0, 9);
    IntVar O = model.intVar("O", 0, 9);
    IntVar F = model.intVar("F", 1, 9);  // F != 0
    IntVar U = model.intVar("U", 0, 9);
    IntVar R = model.intVar("R", 0, 9);

    model.allDifferent(T, W, O, F, U, R).post();

    IntVar[] vars = new IntVar[]{
        T, W, O,
        T, W, O,
        F, O, U, R};
    int[] coeffs = new int[]{
        100, 10, 1,
        100, 10, 1,
        -1000, -100, -10, -1};
    model.scalar(vars, coeffs, "=", 0).post();

    Solver solver = model.getSolver();

    // one solution
    System.out.println(solver.findSolution());
}
```

A first Choco Example

```
Solution: T=9, W=2, O=8, F=1, U=5, R=6,
```

```
    T W O
+   T W O
-------
= F O U R
```

```
    9 2 8
+   9 2 8
-------
= 1 8 5 6
```

Exercise:

- Find and display all 7 solutions

Helper: include the following in the Java code:

```
import org.chocosolver.solver.Model;
import org.chocosolver.solver.Solution;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.variables.IntVar;
```

A variant

Repeat the previous exercise but with the puzzle:

```
    S E N D
+   M O R E
-----------
= M O N E Y
```

- How does it compare with hand-written code for this instance?

Arrays and Matrices

Useful methods to create variables (see IVariableFactory)

`IntVar[] intVarArray(String name, int size, int[] values)`
return an array of size variables (domain: values)

`IntVar[] intVarArray(String name, int size, int lb, int ub)`
return an array of size variables (domain: [lb, ub])

`IntVar[][] intVarMatrix(String name, int s1, int s2, int[] values)`
return a matrix of s1×s2 variables (domain: values)

`IntVar[][] intVarMatrix(String name, int s1, int s2, int lb, int ub)`
return a matrix of s1×s2 variables (domain: [lb, ub])

Exercise: vending machine

Work out the change C to be given out by a vending machine, knowing that the user inserts the amount A (in €cent, or just c) to pay for a drink that costs B 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[2€], C[1€]… C[5c])

Try with A=200 and B=135.

How many solutions? What is the “best”? How to compute it?

- (optional) extend the problem to a COP, i.e. minimise the number of coins

Bound vs Domain consistency

2 levels of consistency for a constraint:

- Domain consistency: the consistency (arc-consistency) is ensured for each value in the domain of each variable of the constraint. This can creates “holes” inside the domain of a variable.
- Bound consistency: the consistency is only ensured for the minimal and maximal value of each variable. (the domain of a variable is approximated by its bounds, i.e. the interval min..max).

Domain consistency removes more impossible values from domains (better pruning) but has higher computational cost.

Bound vs Domain consistency

Choco: consistency level depends on the IntVar type which can be:

- *bounded* : the domain is approximated by its bounds (interval)
- *enumerated*: the domain is stored as a set of values (“holes” can be created).

Methods which return IntVar can also take an ***optional*** last boolean argument “**`bounded`**”, which is true ⇔ the domain is bounded.

Default: Choco choses on the basis of the initial domain size: if less than 32768, it is enumerated, see **`getMaxDomSizeForEnumerated()`**.

Some constraints perform either bound or domain consistency. For some, the user can choose the consistency algorithm.

More constraints

Choco provides many predefined constraints. We here present a few very useful ones

member(IntVar v, int min, int max)
member(IntVar v, int[] values)
Enforce **v** ∈ **min..max** or **v** ∈ { **values…** }

notMember(IntVar v, int min, int max)
notMember(IntVar v, int[] values)
Enforce **v** ∉ **min..max** or **v** ∉ { **values…** }

More arithmetic constraints

`min/max(IntVar z, IntVar v1, IntVar v2)`
`min/max(IntVar z, IntVar[] vars)`

Enforce: **`z`** = minimum / maximum of 2 or more variables

`square(IntVar z, IntVar v)`

Enforce: **`z`** = **`v`**^{2}

`times/div/mod(IntVar x, IntVar y, IntVar z)`

Enforce: **`x`** × **`y`** = **`z`** **`x`** / **`y`** = **`z`** **`x`** % **`y`** = **`z`**

`sum(IntVar[] vars, String op, IntVar z)`

`op` ∈ {**`"="`**, **`"!="`**, **`">"`**, **`"<"`**, **`">="`**, **`"<="`**}

Parameter **`z`** can be an integer

Enforce : **`vars[0]`** + **`vars[1]`** + … + **`vars[n-1]`** **`op`** **`z`**

Constraints as variable views

Some binary constraints can be defined as views on a variable.

E.g. constraints of the form Y = X + C or Y = X × C where C is an integer (Java int) constant. A view is like a new variable (e.g. Y).

`IntVar intOffsetView(IntVar var, int c)`

Returns an **`IntVar`** = **`var + c`**

E.g. **`IntVar Y = model.intOffsetView(X, 10);`**

`IntVar intScaleView(IntVar var, int c)`

Returns an **`IntVar`** = **`var × c`**

Requires **`c`** > -2 (**`c`** = -1 is similar to **`intMinusView`**)

`IntVar intMinusView(IntVar var)`

Returns an **`IntVar`** = **`-var`**

`IntVar intAbsView(IntVar var)`

Returns an **`IntVar`** = **`|var|`**

Exercises

- Encode the N-queens problem.
  - Recall the possible models.
  - Which one do you choose?
  - How does this compare with your previous (pure Java) implementation?
- Encode the Magic Squares (N) problem
  - Can you solve N=3, 4, 5, 6 …
  - What is the runtime for each value of N?
  - What is the current limit under 5 minutes runtime?
  - How does this compare with your previous (pure Java) implementation?