Session 7
Set Constraints

Daniel Diaz
Salvador Abreu

Set constraints in Choco

A set variable (class **SetVar**) represents a *set of integers*

Domain is defined by an interval [LB,UB]

- lower bound (LB, an ISet object) contains integers that must be in every solution
- upper bound (UB) contains integers that may be in at least one solution

Initial values for LB and UB must satisfy LB $\subseteq$ UB.

Computation will remove values from UB and add to LB.

A set variable is instantiated if and only if LB = UB.

Set variable creation

```
// Constant SetVar equal to {2,3,12}
SetVar x = model.setVar("x", new int[]{2,3,12})

// SetVar representing a subset of {1,2,3,5,12}
SetVar y = model.setVar("y",
                        new int[]{},
                        new int[]{1,2,3,5,12});

// possible values: {}, {2}, {1,3,5} …


// SetVar for a superset of {2,3} and subset of {1,2,3,5,12}
SetVar z = model.setVar("z", new int[]{2,3},
                        new int[]{1,2,3,5,12});

// possible values: {2,3}, {2,3,5}, {1,2,3,5} ...
```

Constraints on set variables

`IntVar  getCard()`

get the constrained cardinality variable of this set.

`ISet  getLB()`
`ISet  getUB()`

Get SetVar Lower/Upper Bound : the set of integers that must belong to every/may be in any solution. To iterate over this set, use the following loop: **`ISet lbSet = getLB(); for(int i : lbSet){ ... }`** *This object is read-only*.

`ISet  getValue()`

Retrieves the current value of the variable if instantiated, otherwise the lower bound.

`boolean instantiateTo(int[] value, ICause cause)`

Enforces the set variable to contain exactly the set of integers given in parameter

Constraints on set variables

`allDifferent(SetVar... sets)`

Creates a constraint stating that sets should all be different (not necessarily disjoint) Note that there cannot be more than one empty set

`allDisjoint(SetVar... sets)`

Creates a constraint stating that the intersection of sets should be empty Note that there can be multiple empty sets

`allEqual(SetVar... sets)`

Creates a constraint stating that sets should be all equal

`disjoint(SetVar set1, SetVar set2)`

Creates a constraint stating that the intersection of set1 and set2 should be empty Note that they can be both empty

Constraints on set variables

`element(IntVar index, SetVar[] sets, int offset, SetVar set)`

Creates a constraint enabling to retrieve an element set in sets: sets[index-offset] = set

`element(IntVar index, SetVar[] sets, SetVar set)`

Creates a constraint enabling to retrieve an element set in sets: sets[index] = set

`intersection(SetVar[] sets, SetVar intersectionSet)`

Creates a constraint which ensures that the intersection of sets is equal to intersectionSet

Constraints on set variables

`union(IntVar[] ints, SetVar union)`

Creates a constraint ensuring that union is exactly the union of values taken by ints,

`union(SetVar[] sets, SetVar unionSet)`

Creates a constraint which ensures that the union of sets is equal to unionSet

`partition(SetVar[] sets, SetVar universe)`

Creates a constraint stating that partitions universe into sets: union(sets) = universe intersection(sets) = {}

Constraints on set variables

`member(int cst, SetVar set)`

Creates a member constraint stating that the constant cst is in set

`member(IntVar intVar, SetVar set)`

Creates a member constraint stating that the value of intVar is in set

`member(SetVar[] sets, SetVar set)`

Creates a member constraint stating that set belongs to sets

Exercises

- Encode the Sudoku of rank N problem.
  - Vary the search strategy
- Design a Sudoku generator
  - Provide means for assessing the difficulty of the problem
  - You may use the previous solver as a helper