Session 4
Global Constraints

Daniel Diaz
Salvador Abreu

Abstraction in Problem Solving

Problems may be modeled using

- Few concepts, and
- Compositional mechanisms

Good

- Low implementation complexity
- Easier to prove correct

Bad

- Models may become very large (eg. SAT)
- High-level concepts are “compiled” to RISC-like ones

An approach to Constraint Programming

Modeling expressed as relations over variables

Complex relations may be

- Decomposed into simpler ones, or
- Directly implemented (i.e. by defining their propagators)

Some of these “complex” relations are called **global constraints**, because they operate simultaneously on several variables

Large number of global constraints, there’s even a **catalog**!

http://sofdem.github.io/gccat/

We present some important global constraints (with the names used in the catalog) and their Choco binding.

Global constraints

Informal definition

- Relation involving variables and parameters
- Context independent (supposed not to be application-specific)
- Compact (i.e. conceptually simple to understand and to express)

Global constraints

Some important global constraints (alphabetic order, names used in the catalog):

- alldifferent (catalog)
- circuit (catalog)
- cumulative (catalog)
- diffn (catalog)
- element (catalog)
- exactly (catalog)
- global_cardinality (catalog)
- nvalue (catalog)
- path
- regular (catalog)
- sort (catalog)
- table

Many others exist: allEqual, among, binPacking, knapsack, inverseChanneling,…
Refer to the Choco IIntConstraintFactory doc

Global constraint: alldifferent

Meaning: alldifferent(vars)

- Enforce all variables of vars to take distinct values (vars[i] ≠ vars[j] for all i ≠ j).

Use: occurs in many practical problems directly or indirectly.

Choco:

```
allDifferent(IntVar vars...)
allDifferent(IntVar[] vars)
allDifferent(IntVar[] vars, String CONSISTENCY)
```

CONSISTENCY: **"BC"** for bound consistency, **"AC"** for domain consistency

Variants: apply to a subset of vars (those ≠ 0 or satisfying a given condition c)

```
allDifferentExcept0(IntVar[] vars)
allDifferentUnderCondition(IntVar[] vars, Condition c,
                           boolean singleCondition)
```

Global constraint: table

Meaning: table(vars, tuples)

- Enforce the sequence of variables vars to belong to the set of tuples.

Use: it is sometimes easier and/or more efficient to define a constraint by "extension", as a set of all possible tuples over the variables.

Choco: a class **Tuples** to store the set of tuples (internally with a **List<int[]>**. The class specifies the *allowed* (default) or *forbidden* tuples (using the boolean **feasible** of the constructor).

```
table(IntVar[] vars, Tuples tuples)
table(IntVar[] vars, Tuples tuples, String algo)
```

`algo`: various different consistency algorithms (see doc)

Variants: specialized versions for binary constraints.

```
table(IntVar v1, IntVar v2, Tuples tuples)
table(IntVar v1, IntVar v2, Tuples tuples, String algo)
```

Global constraint: nvalues

Meaning: nvalues(vars, n)

- Enforce the number of distinct values taken by vars to be = n.

Use: timetabling (limit the the maximum number of activity types it is possible to perform), frequency allocation problems (minimize the number of distinct frequencies used over the entire network).

Choco:

```
nValues(IntVar[] vars, IntVar nValues)
```

Variants: the number of distinct values taken by vars to be ≤ or ≥ n

```
 atMostNValues(IntVar[] vars, IntVar n, boolean AC)
atLeastNValues(IntVar[] vars, IntVar n, boolean AC)
```

`AC`: **`true`** for domain consistency (**`false`** = light propagation)
NB nvalues(vars, n) ⇔ atmost_nvalues(vars, n) AND atleast_nalues(vars, n)

Global constraint: exactly

Meaning: exactly(n, vars, value)

- Enforce the number of variables of vars assigned to value = n.

Use: in many practical applications.

Choco: NB: the order of parameters is inverted.

```
count(int    value, IntVar[] vars, IntVar n)
count(IntVar value, IntVar[] vars, IntVar n)
```

Global constraint: global cardinality (GCC)

Meaning: global_cardinality(vars, values, occurrences)

- Enforce: each value values[i] should be taken by exactly occurrences[i] variables in vars.

Use: the global cardinality constraint (GCC) is often used in assignments.

Choco:

```
globalCardinality(IntVar[] vars, int[] values,
                  IntVar[] occurrences, boolean closed)
```

`closed`: if **`true`** restricts the domains of **`vars`** to **`values`**

Global constraint: cumulative

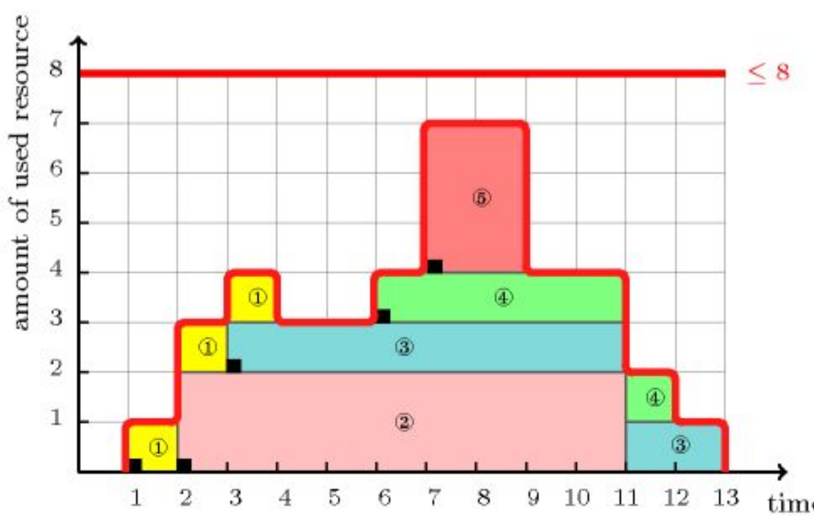
Meaning: cumulative(tasks, limit)

- Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit.
- A task is defined by 4 attributes: (start, duration, end, height) (with start + duration = end).
- The height often corresponds to the amount of used resources by the task. A task can be seen as a "vertically spreadable" rectangle.

Use: cumulative scheduling constraint or scheduling under resource constraints

Example with instantiated variables:

cumulative({ (1 3 4 1),(2 9 11 2), (3 10 13 1),(6 6 12 1), (7 2 9 3) }, 8)



Global constraint: cumulative

`cumulative(Task[] tasks, IntVar[] heights, IntVar limit)`

Task duration and height should be ≥ 0
(tasks whose duration or height = 0 are discarded)

Task is a container representing a task: It ensures that: start + duration = end.
Constructor: **`Task(IntVar start, IntVar duration, IntVar end)`**

Variant:

```
cumulative(Task[] tasks, IntVar[] heights, IntVar limit,
           boolean incremental)
```

`incremental`: **`true`**: apply to a subset of task (faster but less filtering),
`false`: apply to all tasks (**`true`** is the default for the above main constraint)

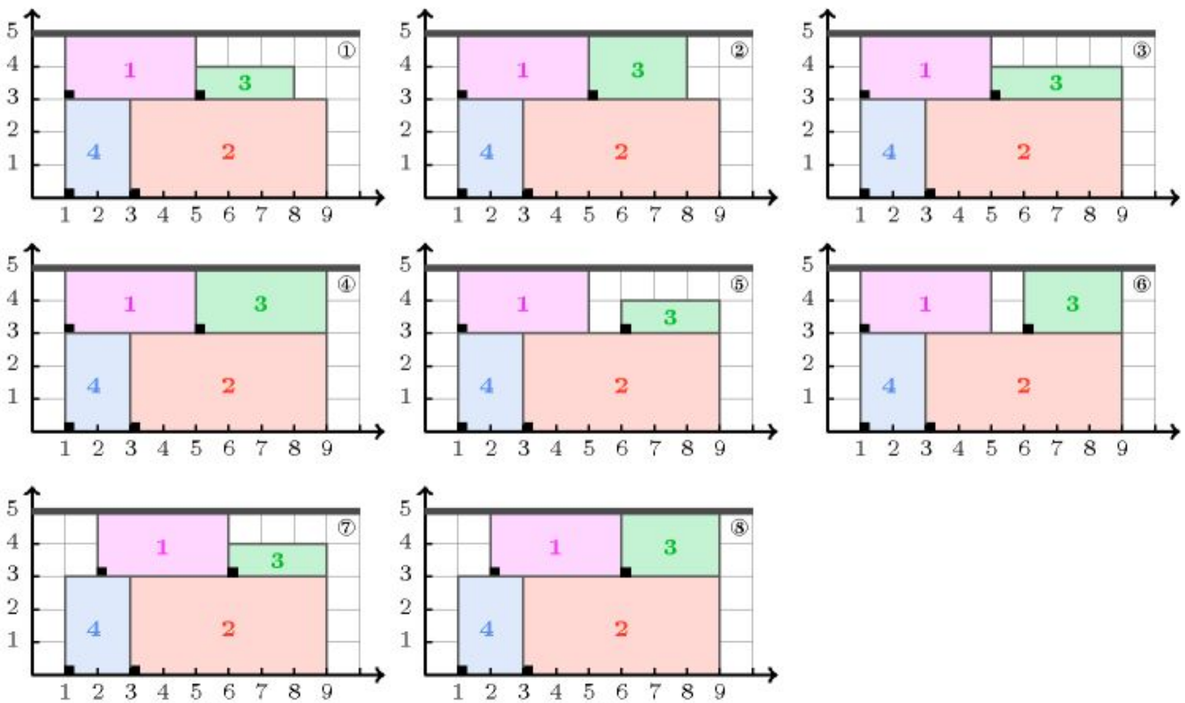
`cumulative({ (start, duration, end, height)... }, limit)`

Global constraint: cumulative

Example with domain variables:

cumulative({
([1..5] [4] [1..9] [2..6])
([2..7] [6] [1..9] [3])
([3..6] [3..6] [1..9] [1..2])
([1..8] [2..3] [1..9] [3..4])}, 5)

Domain reduction:
([1..5] [4] [5..9] [2..4])
([3] [6] [9] [3])
([3..6] [3..6] [6..9] [1..2])
([1] [2] [3] [3..4])



All solutions
(1 4 5 2)(3 6 9 3)(5 3 8 1)(1 2 3 3)
(1 4 5 2)(3 6 9 3)(5 4 9 1)(1 2 3 3)
(1 4 5 2)(3 6 9 3)(6 3 9 1)(1 2 3 3)
(2 4 6 2)(3 6 9 3)(6 3 9 1)(1 2 3 3)
(1 4 5 2)(3 6 9 3)(5 3 8 2)(1 2 3 3)
(1 4 5 2)(3 6 9 3)(6 3 9 2)(1 2 3 3)
(2 4 6 2)(3 6 9 3)(6 3 9 2)(1 2 3 3)
(1 4 5 2)(3 6 9 3)(5 4 9 2)(1 2 3 3)

Exercises: cumulative global constraint

Exercise A: do these instances hold ?

```
1) cumulative({ (1 2 3 3), (2 2 4 2), (4 1 5 1) }, 4)
2) cumulative({ (1 2 3 1), (4 1 5 2) }, 1)
3) cumulative({ (1 2 3 0), (1 2 3 4), (4 1 6 1) }, 4)
```

`cumulative({ (start, duration, end, height)... }, limit)`

Exercice B: find all possible solutions

```
cumulative({ ([1..9] [1] [1..8] [1])
             ([1..9] [2] [1..8] [2])
             ([1..9] [3] [1..8] [5])
             ([1..9] [4] [1..8] [7])}, 7)
```

Do this with a Java program using Choco and the `cumulative` constraint.

Global constraint: element

Meaning: element(index, table, value)

> Enforce value to be the indexth of table (index numbered from 0)
> i.e. value = table[index].

Use: many applications. E.g. encode a discrete function y=f(x). Scheduling: the duration of a task depends on the machine (the table associates a duration to each machine number).

Choco: NB: the order of parameters is inverted.

```
element(IntVar value, int[] table, IntVar index)
```

Variants: allow an offset (to subtract) and/or a table of variables

```
element(IntVar value, int[] table, IntVar index,
        Int offset)
```

```
element(IntVar value, IntVar[] table, IntVar index,
        int offset)
```

> Enforce **`value = table[index-offset]`**

Global constraint: diffn

Meaning: diffn(rectangles)

- Enforce a set of rectangles not to overlap.
- A rectangle is defined by its position (x,y) and size (width, height)
- Can be generalized for the multidimensional case (*orthotopes*).

Use: placement (e.g. design of memory-dominated embedded systems), scheduling (in particular timetabling). E.g. assign each non-preemptive task to a resource and fix its origin so that two tasks, which are assigned to the same resource, do not overlap.

Choco: currently limited to 2D (i.e. rectangles)

```
diffN(IntVar[] x, IntVar[] y,
      IntVar[] width, IntVar[] height,
      boolean cumul)
```

Where **`cumul`** = **`true`** to force additional cumulative (providing more consistency)

Global constraint: sort

Meaning: sort(vars, vars1)

> Enforce the variables of vars1 to be the ordered sequence of variables of vars.
>
> Can be used to obtain in vars a permutation of vars1 (opposite direction)

Use: many constraints involving collections of variables become much simpler to express when the variables of these collections are sorted.

Choco:

```
sort(IntVar[] vars, IntVar[] sortedVars)
```

Global constraint: circuit

Meaning: circuit(vars)

- The n variables of vars encode a digraph:
  vars[i] = j means that j is the successor of i (vars[i] ∈ 0 .. n-1)
- Enforce vars to form a circuit, i.e. to form a permutation of {0,1,...,n-1}

Choco: accepts an offset (e.g. to count from 1 to n). vars[i]=j means that j is the successor of i. Vars[i] ∈ offset .. offset+n-1.

`circuit(IntVar[] vars)`
`circuit(IntVar[] vars, int offset)`, version with an offset.
  i.e. **`vars`** must form a permutation of {**`offset`**, **`offset`**+1, …, **`offset`**+n-1}

Variants: find a circuit over a subset of the vertices. The (sub)circuit is encoded as follows: if vars[i] = offset+j then j is the successor of i (i is part of the circuit).
  if vars[i] = offset+i then i is not part of the circuit.

`subCircuit(IntVar[] vars, int offset, IntVar length)`
  `length`: the size of the subcircuit, i.e. |{**`vars[i]`** ≠ **`offset+i`**}| = **`length`**

Global constraint: circuit (variants)

Variants: see **`circuit`** for offset and for excluded vertices: **`vars[i]`** = **`offset+i`**

`path(IntVar[] vars, IntVar start, IntVar end)`
`path(IntVar[] vars, IntVar start, IntVar end, int offset)`

Enforce the elements of **`vars`** to define a covering path from **`start`** to **`end`** .

`vars` take values in [**`offset`**,**`offset`**+n] with n = |**`vars`**|

Moreover, **`vars[end-offset]`** = n+**`offset`**

`subPath(IntVar[] vars, IntVar start, IntVar end, int offset, IntVar length)`

Enforce the elements of **`vars`** define a path of **`length`** vertices, leading from **`start`** to **`end`**.

Global constraint: regular

Meaning: regular(vars, automaton)

Enforce the sequence of variables of vars to satisfy the automaton (a finite automaton).
It is often more practical to provide a Regular Expression (RE).
(enforce the variables of vars to form a “word” matched by the RE).

Choco:

regular(IntVar[] vars, new FiniteAutomaton(RE**))**
RE is a String containing a regular expression.
Note: use < and > to form numbers, more on RE syntax [here]()

Examples:

```
regular(vars, new FiniteAutomaton(“[1-5]*3*[8-<35>]+”))

regular(vars, new FiniteAutomaton(“[^0-2]*[0-2]+[^0-2]”))
```

Exercise: decompositions

Find all decompositions of an integer n >= 1. A decomposition of n is a tuple (a1,...,ak) s.t. ai >=1 and a1 + a2 + ... + ak = n. For n = 5:

(1 1 1 1 1) (1 1 1 2) (1 1 2 1) (1 1 3) (1 2 1 1) (1 2 2) (1 3 1) (1 4)
(2 1 1 1) (2 1 2) (2 2 1) (2 3)
(3 1 1) (3 2)
(4 1)
(5)

1) How many decompositions for 1? For 2, for 3? For a given n?
2) How would you solve this problem in Java without constraints?
3) Formalize as a CSP and write a Choco program to find all decompositions (respecting the above order if possible).

Hint: what about … a `regular` constraint to remove things unlike (1 2 2 0 0) or (5 0 0 0 0) or (3 2 0 0 0)? :)

Problem: square box packing

Place a set of square boxes in a rectangular container.

- The container is sized N*M
- There are K boxes of side s1, s2, …, sk

Programming notes:

- Use the provided classes for the problem instances (SquarePackingInstance.java) and the abstract framework (SquarePackingAbstract.java) to guide the constraint problem setup
- Use the (SquarePackingSkeleton.java) file as a basis.