

ICEBERG, a library of global constraints

This document describes the interface of the ICEBERG library of the OCRE project. It offers a few global constraints implemented on top of the CHOCO kernel. It cannot be used without its father module CHOCO. ICEBERG is the property of Bouygues e-lab and available for research purposes to all participants of the OCRE project.

1 Assignment constraints

The ICEBERG library provides an implementation of flow based propagation algorithms for bipartite assignment constraints, following the works of Régis.

The semantics of the various flavors of the constraint are the following: given two lists of variables v_i and w_i , `completeAllDiff(list(v1,..., vn))` holds iff

for all i, j , $i \neq j$, $v_i \neq v_j$

`permutation(list(v1,...,vn), list(w1,...,wn))` holds iff

for all i, j , $v_i = j \Leftrightarrow i = w_j$

`gcc(list(v1,...,vn), list((a1..b1),..., (an..bn)))` holds iff

for all i , $1 \leq i \leq n$, the number of variables v_j that take value i is between a_i and b_i

`gcc(list(v1,...,vn), m, p, list((a1..b1),..., (an..bn)))` holds iff

for all i , $1 \leq i \leq p$, the number of variables v_j that take value $m + i - 1$ is between a_i and b_i

1.1 Constructors

With $lv1: list[IntVar]$, $lv2: list[IntVar]$, $lab: list[Interval]$ and $m: integer$, $p: integer$:

`choco/completeAllDiff(lv1)` : CompleteAllDiff

`choco/permutation(lv1, lv2)` : Permutation

`choco/gcc(lv1, lab)` : GlobalCardinality

`choco/gcc(lv1, m, p, lab)` : GlobalCardinality

1.2 Propagation

This constraint propagates generalized arc consistency for all these flavors of the assignment constraints: the propagation phase removes all values not belonging to one feasible assignment (from the perspective of the sole assignment constraint, ignoring the other constraints in the program).

1.3 Examples

Valid situations

For $x:[0,2]$, $y:[0,2]$, $z:[0,2]$

`completeAllDiff(list(x, y, z))` (it is possible to assign pairwise different values to all variables)

For $x:[1,3]$, $y:[2]$, $z:[1,2]$, $t[2,3]$, $u:[1,3]$, $v:[2,3]$

`permutation(list(x, y, z), list(t, u, v))` (contains the solution $x=3$, $y=2$, $z=1$, $t=3$, $u=2$, $v=1$)

For $x:[1,3]$, $y:[2]$, $z:[1,3]$, $t[2,3]$, $u:[4,5]$, $v:[2,5]$

`gcc(list(x, y, z), list((1..2), (1.. 2), (0.. 2), (1.. 5), (1.. 5)))`
(contains the solution $x=1$, $y=2$, $z=3$, $t=3$, $u=4$, $v=5$)

Invalid situations (the minimal cost of the WCSP is 2)

For $x:[0,1]$, $y:[0,1]$, $z:[0,1]$

`completeAllDiff(list(x, y, z))` (not enough values (0 and 1) for 3 variables that must be different)

For $x:[1,3]$, $y:[2]$, $z:[1,2]$, $t[2,3]$, $u:[1]$, $v:[2,3]$

`permutation(list(x, y, z), list(t, u, v))` (impossible to have $y=2$ without having $u=2$)

For $x:[1,3]$, $y:[2]$, $z:[1,3]$, $t[2,3]$, $u:[4,5]$, $v:[2,3]$

`gcc(list(x, y, z), list((1..2), (1.. 2), (0.. 2), (1.. 5), (1.. 5)))`
(at least 2 variables take value within {4,5}, so at most 3 variables take values within {1,2,3}. But x, y, z and v have their domain within {1,2,3}).

2 Weighted Constraint Satisfaction Problems

The ICEBERG library provides an implementation of simple mechanisms for weighted constraint satisfaction, following the works of Verfaillie, Lemaître, Schiex and Fargier.

The semantics of the constraint is the following: given a list of constraints c_i and a list of integer weights w_i and a domain variable $Cost$,

`additiveWCSP(list(c1,..., cn), list(w1, ..., wn), Cost)` holds iff

$Cost \geq W$, where

W is the sum of w_i for all i such that c_i is satisfied

2.1 Constructor

With `lconst:list[AbstractConstraint]`, `lweight:list[integer]` being two lists of same length and `Cost:IntVar`:

```
choco/additiveWCSP(lconst, lweight, Cost)
```

2.2 Propagation

This constraint propagates a lower bound of the global penalty of the weighted CSP. This lower bound is obtained through a forward checking algorithm. It also discards those assignments that would cause the forward checking lower bound to rise above the maximal value allowed for `Cost`.

2.3 Examples

Valid situations

For $x:[0,1]$, $y:[0,1]$, $z:[0,1]$, $c:[0,10]$

`AdditiveWCSP(list(x<>y, y<>z, z<>x), list(2,5,20), c)`

Invalid situations (the minimal cost of the WCSP is 2)

For $x:[0,1]$, $y:[0,1]$, $z:[0,1]$, $c:[0,1]$

`AdditiveWCSP(list(x<>y, y<>z, z<>x), list(2,5,20), c)`

2.4 Implementation

As for all global constraints, propagation occurs in three phases:

- `awake` is responsible for initializing all internal data structures (invariants)
- `awakeOnInf`, `awakeOnSup`, `awakeOnRem`, `awakeOnInst` update the internal state (invariants) of the constraint after each domain update event. These methods mark the `needToAwake` flag of the constraint
- `propagate` performs the necessary propagation (bound recomputation and filtering) when the `needToAwake` flag has been raised.

The constraint is stored with all other delayed constraints of priority 2. Therefore, `propagate` is only called when all propagation events have been treated and all delayed constraints of priority 1 have been propagated.

The constraint stores a sub-constraint network (consisting of those constraints that may be violated). All variables and constraints in this network are numbered. As for the overall Choco structure, the following cross references are available in an `AdditiveWCSP` object `c`:

- `c.subConsts` : the array of all constraints in the WCSP network
- `c.vars` : the array of all variables involved in some of the above constraints
- `c.constIndices` : an array of integer cross references. For a variable v of index `varIdx` (ie: `c.vars[varIdx]=v`), `c.constIndices[varIdx]` is the array of the indices of all subconstraints that involve v .
- `c.varIndices` : an array of integer cross references. For a constraint c of index `subIdx` (ie: `c.subConsts[subIdx]=c`), `c.varIndices[subIdx]` is the array of the indices of all variables spanned by c .

The constraint maintains the following invariants:

- `nbUnInstantiatedVars` : an array, indexed by subconstraints such that `c.nbUnInstantiatedVars[subIdx]` is the number of yet uninstantiated variables spanned by constraint `subc` (of index `subIdx`).
- `ic`: a 2 dimensional matrix of inconsistency counts, such that

$$c.ic[var,val] = \text{Sum}(\text{weight}(subc) \mid subc \text{ in } c.nbUnInstantiatedVars[subc]=1 \\ \text{such that } (var=val) \text{ would imply } (subc \text{ is violated}))$$

Note that `ic` is indexed by two integers: the usual variable index `varIdx` for the first dimension and value `val` up to an offset (minimal value of the initial domain).

- *varPenalty*: an array, indexed by variables, storing the minimum inconsistency count over all possible values of a variable.

$$c.varPenalty[varIdx] = \text{Min}(c.ic[varIdx, val] \mid val \text{ in } \text{domain}(var))$$

These invariants are used to compute lower bounds on the overall cost of the WCSP:

- *backwardCheckingBound*: is the sum of the weights of all constraints that are instantiated and violated (a trivial lower bound)
- *forwardCheckingBound*: is the addition to *backwardCheckingBound* of the sum of the penalties of all uninstantiated variables.

The constraints is propagated by means of two propagation rules:

1. *c.forwardCheckingBound* is maintained as a lower bound of the *Cost* variable,
2. whenever $c.forwardCheckingBound - c.varPenalty[varIdx] + c.ic[varIdx, val]$ rises above $\text{sup}(Cost)$, then the value *val* is removed from the domain of *var*.

The constraint is propagated with the following patterns:

1. upon an instantiation, ($x = v$),
 - $c.ic[x, v]$ is added to *c.backwardCheckingBound*,
 - *c.nbUninstantiatedVars[subcIdx]* is decremented for all constraints *subc* involving *x*
 - whenever some *c.nbUninstantiatedVars[subcIdx]* reaches 1, let *x2* be the only uninstantiated variable of *c*, the weight of *c* is added to $c.ic[x2, val2]$ for all values *val2* in the domain of *x2* such that $x2=val2$ implies that *c* is violated.
 - *c.varPenalty[varIdx]* is updated for all *var* whose inconsistency counts have been changed
 - the constraint is posted in the queue of pending constraint awakenings.
2. Upon any other event ($x <> v, x \geq v, x \leq v$)
 - $c.varPenalty[x]$ is recomputed
 - the constraint is posted in the queue of pending constraint awakenings.

When popping the constraint from the queue of pending constraint awakenings, it is awoken by the call to *propagate(c)*:

- *c.forwardCheckingBound* is recomputed and propagated onto *Cost*
- The look-ahead analysis is performed
- This method returns a Boolean indicating whether some events have been generated or not.

This constraint has been implemented from the description in Lionel Lobjois' PhD thesis, page 50-56. The only adaptation is the reaction to external events¹.

¹ There is a subtlety for accessing the "last uninstantiated variable of a constraint": we do not check that the variable is indeed uninstantiated, but rather that the constraint has not been informed of the instantiation of any other variable concerning that constraints (but such events may be waiting in the propagation queue).