

CHOCO: user's guide

*version 1.324
last update, October 5th, 2002*

Contents of the user's guide

- 1: Principle of use
- 2: Getting started
- 3: Creating problems
- 4: Creating variables
- 5: Stating constraints
- 6: Solving
- 7: Tracing and debugging

1. Principle of use

CHOCO is a CLAIRE class library intended for modeling and solving combinatorial problems. The library is always used within the same scenario: a Problem object is created; Variable objects are created and related to the Problem. Constraints are stated and posted to the problem. Once all variables and constraints have been declared, several algorithms can be applied to the problem: explicit propagation algorithms, global tree search for finding one or all solutions and local search methods for improving a (possibly infeasible) solution, according to some optimization criterion.

Several problems can be created in parallel, variables and constraints may be attached to them. These problems can then be solved one after another, without needing to re-initialize the system.

This document describes the methods that are available for building problems, creating variables, posting constraints and applying various solving procedures.

2. Getting started

The installation package comes as a compressed archive (.zip package). It creates the following structure on your disk under a host directory (say, "d:\user\myself\greatProjects\choco"):

- a directory, say v1.318, containing the sources of the latest release,
- a test directory, containing simple regression tests (small programs testing simple the behavior of CHOCO on simple cases),
- a bench directory, containing simple programs used as benchmarks for the solver,
- an `init.cl` file used for defining the choco module. This file is read by Claire, when the Claire interpreter is launched.

CHOCO can be used as any CLAIRE library and applications using CHOCO can be developed as any other CLAIRE applications. For users unfamiliar with CLAIRE, we propose the following steps for writing your first program.

1. create a new directory, named `apps` under the `choco` host directory, say "C:\user\myself\greatProjects\choco". You will use this directory as a repository for the files of your application.
2. edit the `init.cl` file from the Choco package and add a new module declaration, such as


```
mychocoapp :: module(
    source = ChocoInstallationDir / "apps",
    uses = list(choco),
    made_of = list("myappfile"))
```

You have just declared a new Claire module, made of one single file (`apps/myappfile.cl`), and declared that this module may use all primitives defined in the Choco library.

3. create the `myappfile.cl` file in the `apps` directory. Write in this file your first code. For instance, write a function that solves a silly CSP such as


```
[sillyCSP() : void
-> let pb := choco/makeProblem("Silly CSP",3),
    x := choco/makeIntVar(pb, "x", 1, 3),
    y := choco/makeIntVar(pb, "y", 1, 3),
    z := choco/makeIntVar(pb, "z", 1, 3) in
(choco/post(pb, x + y == z),
choco/post(pb, x > y),
choco/propagate(pb),
printf("~S ~S ~S\n",x,y,z) )]
```

You have just declared a new function, with no parameters and `void` result; the function uses four local variables, `pb`, `x`, `y` and `z` (created in that order). These local variables are assigned the result of the evaluation of a Choco primitive (`makeProblem`, `makeIntVar`) for creating a CSP or a finite domain variable. Note the use of the `choco/` prefix for function calls (this is the syntax for calling a function defined in another module). The main block of the function consists in a sequence of calls to `choco/post` (for posting constraints to a problem). Constraints are created on the fly, by using arithmetic operators (here, `+`, `==`, `>`). Then, the `propagate` method is called and the variables are printed out through a `printf` expression.

Note that there is no mention of memory management: the Claire garbage collector will dispose of the memory used by objects that are no longer referenced.

4. Call the Claire interpreter with the `mychocoapp` module:


```
claire -s 3 3 -m mychocoapp
```

```

You then get a prompt, waiting for commands
[C:\user\myself\greatProjects\choco]claire -s 3 3 -m mychocoapp
increasing memory size by 2^3 and 2^3
-- CLAIRe run-time library v 2.5.64 [os: ntv, C++:gcc ] --
-- CLAIRe interpreter - Copyright (C) 1994-00 Y. Caseau (see about())
---- note: choco is a root module !
---- note: chocotest is a root module !
---- note: chocobench is a root module !
---- note: mychocoapp is a root module !
---- Loading the module choco.
---- [load CLAIRe file: .\v1.08\model.c]
Choco version 1.08, Copyright (C) 1999-2001 F. Laburthe
Choco comes with ABSOLUTELY NO WARRANTY; for details read licence.txt
This is free software, and you are welcome to redistribute it
under certain conditions; read licence.txt for details.
---- [load CLAIRe file: .\v1.08\iprop.c]
---- [load CLAIRe file: .\v1.08\const.c]
---- [load CLAIRe file: .\v1.08\const2.c]
---- [load CLAIRe file: .\v1.08\bool.c]
---- [load CLAIRe file: .\v1.08\main.c]
---- [load CLAIRe file: .\v1.08\chocapi.c]
---- Loading the module mychocoapp.
---- [load CLAIRe file: .\apps\myappfile.c]
mychocoapp>

```

You can then, from there, evaluate Claire expressions using the code that you have defined in your `mychocoapp` module. For instance, you may call the `sillyCSP` function

```

mychocoapp> sillyCSP()
eval[0]> x:2 y:1 z:3
nil
mychocoapp>

```

The function call is evaluated (and, yes, you do see the domain of the three variables reduced to a single value after propagation). The Claire toplevel prints `nil` when a procedure (method with void result) is called. Once the call is evaluated, the toplevel prompts again the user for a new expression to evaluate. In order to end the session, type `q` (and press return).

```

mychocoapp> q
eval[1]> Bye.

```

After this first trivial program, we suggest you go through the programs provided in the `bench` directory (`chocobench` module). They provide a tutorial by the example.

3. Creating Problem's

The `Problem` class is used to group a set of variables and a set of constraints.

There is one constructor for the `Problem` class

- `makeProblem(s:string, n:integer) : Problem`
creates a problem with name `s`, and ready to handle at most `n` variables¹.
As the reader will notice, it is mandatory to call this function before creating variables since the primitives for creating variables take a `Problem` as argument. It is therefore always the first function call among all CHOCO API methods in a session.

A few general methods are offered to inspect the current status of a problem:

- `self_print(p:Problem) : void`
pretty-prints the problem (displaying its name, number of variables, etc...).
- `showVars(p:Problem) : void`
displays the domains of all variables linked with the problem `p`.

4. Creating variables

CHOCO supports two kinds of variables: integer valued variables (i.e. unknowns that must be assigned integers) and set valued variables (i.e. unknown that must be assigned a finite set of integers).

¹ When an $n+1^{\text{th}}$ variable is defined, a warning will be displayed, explaining that event queue overflows may occur (resulting in incomplete propagation).

4.1. Integer variables

Integer variables in CHOCO are instances of the `IntVar` class. The library provides the user with two domain representations. The first one keeps the whole enumeration of values. The second one is an interval approximation where only the lower and upper bounds of the domain are stored. The second implementation abstracts the set of values in the domain into an integer interval and ignores all holes in the sequence of feasible values. This second implementation is useful when the variables are involved in linear constraints, or have very large domains. For instance, this is the case with variables modeling dates.

All domains must be subset of the interval $(\text{MININT} .. \text{MAXINT})$, where `MININT` and `MAXINT` are two preset constants. In CHOCO v1.0, `MININT = - 99999999` and `MAXINT = 99999999`.

The next functions create variables whose domain of values is stored without any approximation:

- `makeIntVar(p:Problem, s:string, d:(set[integer] U list[integer])) : Var`
creates a finite domain variable with domain `d`, with name `s`, related to the problem `p`.
- `makeIntVar(p:Problem, s:string, i:integer, j:integer) : Var`
creates a finite domain variable with domain $(i .. j)$, with name `s`, and related to the problem `p`.
- `makeIntVar(p:Problem, d:(set[integer] U list[integer])) : Var`
creates an un-named finite domain variable with domain `d`, related to the problem `p`.
- `makeIntVar(p:Problem, i:integer, j:integer) : Var`
creates an un-named finite domain variable with domain $(i .. j)$, related to the problem `p`.

The next three functions create variables whose domain is approximated by an interval of integers:

- `makeBoundIntVar(p:Problem, i:integer, j:integer) : Var`
creates a finite domain variable with domain $(i .. j)$ and related to the problem `p`
- `makeBoundIntVar(p:Problem, s:string) : Var`
creates a finite domain variable with domain $(-100 .. 100)$, with name `s`, and related to the problem `p`
- `makeBoundIntVar(p:Problem, s:string, i:integer, j:integer) : Var`
creates a finite domain variable with domain $(i .. j)$, with name `s`, and related to the problem `p`

This last constructor function creates a variable whose domain is exactly $\{x\}$, eg, an instantiated variable:

- `makeConstantIntVar(p:Problem, x:integer) : Var`

The state of an `IntVar` can be accessed through the following public methods :

- `getInf(x:IntVar) : integer`
returns the lower bound on the current domain of the variable. This value of this lower bound increases throughout propagation, and is stored in such a manner that former values may be retrieved upon backtracking
- `getSup(x:IntVar) : integer`
returns the upper bound on the current domain of the variable (the upper bound has the same monotony property as the lower bound).
- `getValue(x:IntVar) : (integer U {unknown})`
returns the value of the variable when it is instantiated (otherwise, returns unknown).
- `isInstantiated(x:IntVar) : boolean`
returns true iff the variable is instantiated (ie: when its domain is a singleton).
- `canBeInstantiatedTo(x:IntVar, y:integer) : boolean`
returns true iff the value `y` is contained in the domain of variable `x`.
- `isInstantiatedTo(x:IntVar, y:integer) : boolean`
a method for testing whether the domain of `x` is restricted to the singleton $\{y\}$.
- `domainSize(v:IntVar) : integer`
returns the cardinal of the domain for a variable (the number of values that can still be assigned to the variable).
- `domain(x:IntVar) : (list[integer] U Interval[integer])`
returns the list of values in the domain of a variable, by increasing order. An optimized domain iterator is also available, and iterations for `v in domain(x)` are compiled efficiently, generating the values on the fly and avoiding to actually build the list of values from the domain.
- `self_print(v:IntVar) : integer`
`self_print` methods are the general CLAIRE pretty printing mechanism. They are called within `printf` statements, for all `IntVar` parameters corresponding to the `~S` escape pattern. For instance,
`printf("The last task date is ~S",timeVar)`
is interpreted as `(princ("The last task date is "), self_print(timeVar))`

When the `IntVar` (a domain variable, i.e. a modeling variable) bound to the local variable `timeVar` (a CLAIR variable, i.e. a variable from the programming language) is named “DATE”, this statement may produce one of the three results:

The last task date is DATE:2001

The last task date is DATE:[1999-2005]

The last task date is DATE:[4]{1999, 2001, 2003, 2005},

depending on whether the variable is instantiated to 2001, has an interval representation for its domain, with 1999 and 2005 as bounds or whether its domains is the set of odd values from 1999 to 2005.

The following methods can be used for changing the state of a variable (restricting its domain). These functions record events in queues; the next propagation phase will handle all pending events, trigger propagation computations and eventually transform the problem from an incomplete state either into a complete one (when all variables of the problem are instantiated after the call), or into an incomplete one (all domains have shrunk during the call, but there remain some variables with several values in their domain), or to a failure (a contradiction exception is raised). In all cases, the effect of the decision and its consequence may be backtracked. The four functions are:

- `setMin(v:IntVar, i:integer)`
removes all values strictly smaller than i from the domain of v .
- `setMax(v:IntVar, i:integer)`
removes all values strictly greater than i from the domain of v .
- `setVal(v:IntVar, i:integer)`
restrict the domain of v to value i .
- `remVal(v:IntVar, i:integer)`
removes value i from the domain of v .
- `remInterval(v:IntVar, a:integer, b:integer)`
removes all values between a and b from the domain of v .

⚠ These methods do NOT trigger propagation². If the modification of the domain is impossible (for instance if the value of a `setVal` is not in the domain) they throw a contradiction exception. Otherwise they modify the domain and post the corresponding event, that will be propagated on the next call to the `propagate` method.

4.2. Set variables

Set variables in CHOCO are instances of the `SetVar` class. The library provides the user with an implementation of domains using two bounds: the upper bound (the union of all possible set values for the variable) is called the domain envelope, and the lower bound (the intersection of all possible set values for the variable) is called the domain kernel. Both domain bounds are stored with bitvectors. Therefore, the propagation engine knows for each integer value, whether that value may not be contained in the set variable (when it is not present in the domain envelope) and whether that value must be contained in the set variable (when it is present in the domain kernel).

The next functions create variables whose domain of values is stored without any approximation:

- `makeSetVar(p:Problem, minv:integer, maxv:integer) : SetVar`
creates an unnamed set variable that may contain values between $minv$ and $maxv$, related to problem p .
- `makeSetVar(p:Problem, s:string, minv:integer, maxv:integer) : SetVar`
creates a set variable with name s that may contain values between $minv$ and $maxv$, related to problem p .
- `makeSetVar(p:Problem, minv:integer, maxv:integer, nb:IntVar) : SetVar`
creates an unnamed set variable that may contain values between $minv$ and $maxv$, related to problem p ; binds the cardinal of that set variable (ie: the number of values contained in the set) to the integer variables nb .
- `makeSetVar(p:Problem, s:string, minv:integer, maxv:integer, nb:IntVar) : SetVar`
creates a set variable with name s that may contain values between $minv$ and $maxv$, related to problem p ; binds the cardinal of that set variable (ie: the number of values contained in the set) to the integer variables nb .

The state of a `SetVar` can be accessed through the following public methods :

- `getDomainKernel(v:SetVar) : list[integer]`
returns the lower bound (kernel) on the current domain of the set variable (this value of this lower bound

² Even `setVal` and `remVal` since version 1.0.

increases throughout propagation, and is stored in such a manner that former values may be retrieved upon backtracking).

- `getDomainKernelSize(v:SetVar) : integer`
returns the number of values in the lower bound (kernel) on the current domain of the set variable.
- `getDomainKernelInf(v:SetVar) : integer`
returns least value in the lower bound (kernel) on the current domain of the set variable.
- `getDomainKernelSup(v:SetVar) : integer`
returns greatest value in the lower bound (kernel) on the current domain of the set variable.
- `isInDomainKernel(v:SetVar, x:integer) : boolean`
tests whether an integer is contained in the lower bound (kernel) on the current domain of the set variable.
- `getDomainEnvelope(v:SetVar) : list[integer]`
returns the upper bound (enveloppe) on the current domain of the set variable (the upper bound has the same monotony property as the lower bound).
- `getDomainEnvelopeSize(v:SetVar) : integer`
returns the number of values in the upper bound (enveloppe) on the current domain of the set variable.
- `getDomainEnvelopeInf(v:SetVar) : integer`
returns least value in the upper bound (enveloppe) on the current domain of the set variable.
- `getDomainEnvelopeSup(v:SetVar) : integer`
returns greatest value in the upper bound (enveloppe) on the current domain of the set variable.
- `isInDomainEnvelope(v:SetVar, x:integer) : boolean`
tests whether an integer is contained in the upper bound (enveloppe) on the current domain of the set variable.
- `getValue(x:SetVar) : list[integer]`
returns the value of the set variable when it is instantiated.
- `isInstantiated(x:SetVar) : boolean`
returns true iff the variable is instantiated (ie: when its two domain bounds are equal).
- `isInstantiatedTo(x:SetVar, y:list[integer]) : boolean`
a method for testing whether the two domain bounds are equal to the list of integer values *y*.
- `self_print(v:SetVar) : integer`
`self_print` methods are the general CLAIRE pretty printing mechanism. They are called within `printf` statements, for all `IntVar` parameters corresponding to the `~S` escape pattern. For instance,
`printf("The last task date is ~S",timeVar)`
is interpreted as `(princ("The last task date is "), self_print(timeVar))`

The following methods can be used for changing the state of a variable (restricting its domain). They behave just like the methods on integer variables, recording propagation events in queues. The two functions are:

- `setIn(v:SetVar,i:integer) : void`
enforces that the set *v* contains value *i* by adding *i* to the domain lower bound (kernel) of *v*.
- `setOut(v:IntVar,i:integer) : void`
enforces that the set *v* does not contains value *i* by removing *i* from the domain upper bound (enveloppe) of *v*.

5. Stating constraints

Like domain variables, constraints are stored in objects organized into a class hierarchy: each constraint subclass implements a filtering algorithm for a constraint type. This hierarchy is rooted under `AbstractConstraint`, an abstract class (a class without instances), from which all constraints inherit.

Stating a constraint is systematically done in two phases: first, the constraint is created, then, it is posted to a problem.

- `post(p:Problem,c:AbstractConstraint) : void`
adds the constraint *c* to the set of constraints of *p*, and connects it to the constraint graph. Constraints that are created but not posted will not have any effect: they will neither reduce their variable domains through propagation, nor cause contradictions.

⚠ Note that constraint posts are NOT backtrackable, thus posting a constraint is not a valid decision in a tree search. There are two ways to implement backtrackable constraint posts:

- either with Boolean connectors that propagate a constraint *d* as soon as a condition *c* holds (see constructors *c implies d* and *ifThen(c,d)* defined in section 5.5.

- or by posting a constraint c , and using the *setActive* and *setPassive* methods defined in section 5 in order to remove the constraint from the propagation network and to put it back in.

We now review the various constraints that may be posted to a problem.

5.1. Arithmetic constraints

The simplest constraints are comparisons. The comparators can be used with an infix syntax; both arguments can be either variables, integer constants or arithmetic terms. For instance $x \geq y$ creates a constraint between two variables x and y . This constraint is added to the constraint network (and becomes active) after the following call `post(p, x >= y)`. As explained below, a term is an expression obtained by (possibly recursive) calls to the binary $+$ and $-$ operators, on variables and/or integers.

Arithmetic terms are temporary objects used for storing arithmetic (sub-)expressions. They are stored in three classes (an abstract class, *Term*, and subclasses *UnTerm*, *BinTerm* and *LinTerm*). For instance, the expression $3*x + 5*y - 7*z \geq 4$ generates nested function calls:

$$\geq(-(+(*(3,x), *(5,y)), *(7,z)), 4)$$

The overall result (returned by the call to the operator \geq) will be a linear constraint object. The result of sub-expressions will be stored in Terms: a *UnTerm* for $*(3,x)$, a *BinTerm* for $+(*(3,x), *(5,y))$ and a *LinTerm* for $-(+(*(3,x), *(5,y)), *(7,z))$.

Linear terms represent linear combinations of variables (with integer coefficients). They can be formed with arithmetic operators: for instance $3 * X + 2 * Y - Z + 5$ is a linear term.

- `*(v1:integer, v2:IntVar) : Term`
- `+(t1:(Term U IntVar U integer), t2:(Term U IntVar U integer)) : Term`
- `-(t1:(Term U IntVar U integer), t2:(Term U IntVar U integer)) : Term`

⚠ Note: watch out that such arithmetic operators, like all Claire operators, MUST be surrounded with blanks. The reader will understand $x + y$ as a proper call $+(x, y)$, while $x+y$ will be read as one single new symbol.

Two additional functions provide the user with the ability to form linear combinations, while stating separately coefficients and variables. The *sumVars* method returns a linear combination with all coefficients set to 1.

- `scalar(coeffs:list[integer], vars:list[IntVar]) : Term`
- `sumVars(vars:list[IntVar]) : Term`

The following comparators for terms are available:

- `>=(v1:Term, v2:Term) : AbstractConstraint`
- `>(v1:Term, v2:Term) : AbstractConstraint`
- `<=(v1:Term, v2:Term) : AbstractConstraint`
- `<(v1:Term, v2:Term) : AbstractConstraint`
- `==(v1:Term, v2:Term) : AbstractConstraint`
- `!=(v1:Term, v2:Term) : AbstractConstraint`

⚠ Note that the `==` operator is used for equality, and `!=` for disequality³. The usual Claire operators, `=` and `!=`, are reserved for comparing the objects (testing whether two variables refer to the same physical object in memory and not stating that they should be bound to the same value).

5.2. Set constraints

In addition to the use of integer variables for denoting set cardinality, Choco supports the statement of constraints among sets through the following primitives:

- `memberOf(sv:SetVar, a:integer) : AbstractConstraint`
states that the *sv* set variable contains value a ,
- `notMemberOf(sv:SetVar, a:integer) : AbstractConstraint`
states that the *sv* set variable does not contain value a ,
- `memberOf(sv:SetVar, v:IntVar) : AbstractConstraint`

³ note the change from `<>` to `!=` in choco v1.0. (due to the definition of valid operators in Claire v3.0).

- states that the *sv* set variable contains the value of the integer variable *v*,
- `notMemberOf(sv:SetVar, v:IntVar) : AbstractConstraint`
states that the *sv* set variable does not contains the value of the integer variable *v*,
- `setIntersection(s1:SetVar, s2:SetVar, s3:SetVar) : AbstractConstraint`
states that the *s3* set variable is the intersection of set variables *s1* and *s2* ; e.g. states that *s3* contains exactly those values contained in both sets *s1* and *s2*,
- `setUnion(s1:SetVar, s2:SetVar, s3:SetVar) : AbstractConstraint`
states that the *s3* set variable is the union of set variables *s1* and *s2* ; e.g. states that *s3* contains exactly those values contained in either sets *s1* or *s2*,
- `subset(s1:SetVar, s2:SetVar) : AbstractConstraint`
states that *s1* is a subset of *s2* ; e.g. that all values contained in *s1* are also contained in *s2*,
- `disjoint(s1:SetVar, s2:SetVar) : AbstractConstraint`
states that *s1* and *s2* are disjoint sets ; e.g. that *s1* and *s2* contain no common values
- `overlap(s1:SetVar, s2:SetVar) : AbstractConstraint`
states that *s1* and *s2* are overlapping sets ; e.g. that *s1* and *s2* contain at least one common value

5.3. User-defined binary constraints

Choco also supports the statement of user-defined constraints, defined by arbitrary relations⁴. The relation defines feasible pairs of values for the two variables involved in the constraint. Relations may be defined by two means:

- *Tables*: specifying those pairs of values for which the constraint is satisfied,
- *Predicates*: specifying the function to be called in order to check whether a pair of values is feasible or not.

The functions for creating a constraint for a relation are the following:

- `binConstraint(va:IntVar,vb:IntVar,feasRel:BinRelation,feas:boolean,ac:integer) : CSPBinConstraint`
- `binConstraint(va:IntVar,vb:IntVar,feasRel:BinRelation,feas:boolean) : CSPBinConstraint`
- `binConstraint(va:IntVar,vb:IntVar,feasRel:BinRelation,ac:integer) : CSPBinConstraint`
- `binConstraint(va:IntVar,vb:IntVar,feasRel:BinRelation) : CSPBinConstraint`

Parameters *va*, *vb* and *feasRel* should be self-explanatory. Parameter *feas* indicates whether the relation models feasible pairs of values (default *feas = true*); parameter *ac* selects the algorithm for enforcing arc-consistency (default *ac = 2001*). Supported values for this parameter are:

- 3 for the AC3 algorithm (searching from scratch for supports on all values)
- 4 for the AC4 algorithm (maintaining a count of supports for each value)
- 2001 for the AC2001 algorithm (keeping a support iterator for each value).

Binary relations may be created by two methods, defining the relation either by a predicate or a table.

- `makeBinTruthTest(m:method[range = boolean]) : BinRelation`
This call passes the test method *m* as argument (*m* must take two integer parameters and return a Boolean, indicating whether the pair of values is feasible or not).

⚠ Note that the feasibility test for a relation must be deterministic. It should always return the same result for a pair of values. For instance, one should not make random draws in such a test.

- `makeBinRelation(min1:integer, max1:integer, min2:integer, max2:integer) : TruthTable2D`
The second method creates an empty relation table across value range (*min1* .. *max1*) for the first variable and (*min2* .. *max2*) for the second one. (No pairs of values are allowed). The range *TruthTable2D* is a subclass of *BinRelation*
- `makeBinRelation(min1:integer, max1:integer, min2:integer, max2:integer, mytuples:list[tuple(integer,integer)]) : TruthTable2D`
Creates a relation table across ranges (*min1* .. *max1*) and (*min2* .. *max2*), directly containing all pairs in *mytuples*.
- `setTruePair(dtt:TruthTable2D, x:integer, y:integer) : void`

⁴ A third (advanced) way to create a constraint is to define a new descendent of the `AbstractConstraint` class, as explained in document “Inside Choco”.

This method updates a relation table by adding the pair (x,y) to the relation

- `getTruthValue(dtt:BinRelation, x:integer, y:integer) : boolean`
This method accesses the value of a pair to check whether it is in the relation.

In addition to the modeling point of view, there is a run-time difference in efficiency between constraints specified by a truth table and constraints defined by a satisfaction test:

On the one hand, table constraints may become rather memory consuming in case of large domains, although relation tables may be shared by different constraints. On the other hand, predicate constraints require little memory as they do not cache truth values, but imply some run-time overhead for calling the feasibility test. Table constraints are thus well suited for constraints over small domains; while predicate constraints are well suited for situations will large domains.

5.4. User-defined larger constraints

The situations for binary constraints is extended to the case of relations involving more than two variables, upto a significant difference from the propagation point of view: the propagation engine maintains arc consistency for binary constraints throughout the solving process, while for n-ary constraints, it uses a weaker propagation mechanism with a forward checking algorithm.

The functions for creating such constraints are the following ones:

- `feasTupleConstraint(vars:list[IntVar], goodTuples:list[list[integer]]) : AbstractConstraint`
- `inFeasTupleConstraint(vars:list[IntVar], badTuples:list[list[integer]]) : AbstractConstraint`
- `feasTestConstraint(vars:list[IntVar], m:method[domain = list[integer], range = boolean]) : AbstractConstraint`
- `infeasTestConstraint(vars:list[IntVar], m:method[domain = list[integer], range = boolean]) : AbstractConstraint`

5.5. Boolean operators

Arithmetic constraints can be combined through Boolean connectors. Such Boolean operators can be used with an infix syntax; there are five of them :

- `or(c1:AbstractConstraint, c2:AbstractConstraint) : AbstractConstraint`
- `and(c1:AbstractConstraint, c2:AbstractConstraint) : AbstractConstraint`
- `implies(c1:AbstractConstraint, c2:AbstractConstraint) : AbstractConstraint`
- `ifThen(c1:AbstractConstraint, c2:AbstractConstraint) : AbstractConstraint`
- `ifOnlyIf(c1:AbstractConstraint, c2:AbstractConstraint) : AbstractConstraint`

For instance, here is a way to state that the absolute value of the difference between two variables x and y should be equal to 2: `post(p, (x - y == 2) or (y - x == 2))`.

⚠ Note that only the Boolean composition is posted to the problem p , and not the branches. Indeed we may not propagate the constraint $x - y == 2$ before we definitely know which of the two branches is actually valid in the solution.

The *implies* and *ifThen* operators have the same semantics: when either the constraint *implies*($c1,c2$) or the constraint *ifThen*($c1,c2$) is posted to a problem, the valid solutions of that problem must either satisfy both the constraints $c1$ and $c2$, or not satisfy $c1$.

The difference between both constraints is the propagation algorithm that they use. *ifThen*($c1,c2$) starts propagating $c2$ as soon as constraint $c1$ becomes entailed (when the engine is sure that $c1$ will be satisfied given the current domain of the variables). In a sense, *ifThen*($c1,c2$) performs a kind of lazy propagation for $c2$. *implies*($c1,c2$) performs more propagation, by propagating the opposite of $c1$ as soon as the opposite of $c2$ is entailed (when the engine is sure that $c2$ will not be satisfied, given the current state of domains). Thus, the constraint *implies*($c1,c2$) is exactly equivalent (in terms of semantics and propagation behavior) to *or*(*opposite*($c1$), $c2$).

⚠ Note that the lightweight guard mechanism: *ifThen*($c1,c2$) can be useful for constraint that arise from hierarchical problems. For instance, in a mixed resource-allocation / production-scheduling problem, one may not want to consider any low-level constraint (from production planning) until the high-level problem (resource

allocation) has been fully solved. Such a decomposition is achieved through the *ifThen* operator, while the *implies* operator would trigger many checks on the feasibility of *c2* before *c1* is instantiated.

One last operator offers the possibility to specify the number of constraints among a collection that are actually satisfied.

- `card(l:list[Constraint], nb:IntVar) : AbstractConstraint`

5.6. Array Constraints

The following constraints are used to access an array through a variable index (for expressions using variable subscripts). The user may post a constraint $getNth(l,i) == x$ for representing by variable x , the i -th entry of the list of integer values l (i is an *IntVar*). Such use is generalized to the bi-dimensional case, with expressions $getNth(t,i,j) == x$ that represent by variable x the (i,j) entry of the table integer table t .

- `getNth(lvars:list[integer], x:IntVar) : Term`
- `getNth(lvars:table[range = integer], x:IntVar, y:IntVar) : Term`

5.7. Global constraints

Two global constraints are offered in CHOCO, a difference constraint and an occurrence constraint.

- `allDifferent(vars:list[IntVar]) : Constraint`
this primitive states that a list of variables should be assigned pair-wise different values. The same propagation is performed as if all $n(n-1)/2$ binary difference constraints among the n variables had been posted. For an implementation of the generalized arc consistency propagation algorithm [Régin 94], the user could use the Iceberg library of global constraints.
- `occur(target:integer, lvars:list[IntVar]) : Term`
this primitive accounts the number of occurrences of value `target` within the list of variables `lvar`. It is to be used within comparisons with a constant or an *IntVar*. For instance, `occur(2, l) >= 4` states that at least 4 among the variables in `l` should take value 2. Its propagation is equivalent to
 $card(l[1] == 2, l[2] == 2, l[4] == 2, \dots, l[n] == 2) == 4$

6. Solving

This section describes the available tools for solving problems: propagation, global and local search.

6.1. Controlling propagation

Two functions are responsible for controlling propagation:

- `propagate(pb:Problem) : void`
propagates all constraints until the fix-point is reached. The first call triggers the initial propagation of all constraints that have been posted to the problem. Further calls react to all propagation events (domain reductions) that have occurred since the last call to `propagate`. Note that since `propagate` reaches a fixpoint, it is always useless to call the method several times in a row if no propagation events have occurred between the calls.
- `setPassive(c:AbstractConstraint) : void`
removes a constraint from the propagation network. If this method is called before the first call to `propagate`, the problem behaves as if the constraint had never been posted to the problem.
- `setActive(c:AbstractConstraint) : void`
puts back into the propagation network a constraint that had been previously removed (through a call to `setPassive`). This method generates a propagation event, so a call to `propagate` is necessary thereafter. Note that the effect of both `setPassive` and `setActive` are undone upon backtracking.
- `raiseContradiction(pb:Problem) : void`
- `raiseContradiction(v:IntVar) : void`
- `raiseContradiction(c:AbstractConstraint) : void`
calling one of these functions states that an infeasible state has been reached. This call raises an exception from the `contradiction` class. This function is used by all functions performing constraint checks. The parameter indicates the origin of the contradiction (the overall problem, a variable with an empty domain or a constraint that becomes infeasible)

- `getContradictionCause(pb:Problem) : any`
This function can be called whenever a contradiction has been raised. It returns the cause of the contradiction (the parameter that was passed to the `raiseContradiction` function).

6.2. Global search

In order to perform tree search algorithms for finding a solution, one needs to create a `GlobalSearchSolver` object, attach it to the problem and run it.

The global search object can be created as follows

- `makeGlobalSearchSolver(allSolutions:boolean) : GlobalSearchSolver`
creates an object representing an algorithm searching either for one or all solutions, depending on the Boolean value of the second parameter. In case `allSolutions=true`, the solver goes through all solutions and returns to the root node of the search tree, once the exploration is over. In case `allSolutions=false`, the solver stops at the first solution and remains in the state of that first feasible solution.
- `makeGlobalSearchMaximizer(objective:IntVar, restartSearch:boolean) : GlobalSearchSolver`
- `makeGlobalSearchMinimizer(objective:IntVar, restartSearch:boolean) : GlobalSearchSolver`
creates an object representing an algorithm searching for a solution that maximizes (resp. minimizes) the value of objective. The solver goes through a tree search exploration generating a sequence of solutions with increasing (resp. decreasing) values of the objective variable, until it comes to a solution with the optimal value and proves its optimality. When the second parameter (`restartSearch`) is set to true, a new tree search is started after each solution, posting at the root of such a new tree that the objective should strictly improve over the previous value. When that parameter (`restartSearch`) is set to false, only one tree is searched; after each solution the improvement constraint is posted.

Once the solver has been created, it needs to be attached to the problem and run:

- `attach(gs:GlobalSearchSolver, pb:Problem) : void`
- `run(gs:GlobalSearchSolver) : void`
returns a Boolean indicating whether a solution has been found or not, or in the case of optimization solvers, the objective value of the best solution found.

Once the search has been performed, the state of the solver can be accessed through the following functions:

- `getFeasibility(gs:GlobalSearchSolver) : boolean`
returns a Boolean indicating whether the problem was found feasible or not.
- `getNbSol(gs:GlobalSearchSolver) : integer`
returns the number of feasible solutions found in the last search
- `getNbBk(gs:GlobalSearchSolver) : integer`
returns the number of backtracks in the last tree search.

The solutions found by these primitives are not directly accessible, but they can be restored using the following function:

- `restoreSolution(gs:GlobalSearchSolver, i:integer) : void`
restores the i^{th} solution (the total number of solutions being stored in the `nbSol` slot of `Problem`)

6.3. Controlling the exploration of the search tree

The behavior of the algorithm can be controlled through the following methods (that should be called before the algorithm is run).

- `setMaxSolutionStorage(algo:GlobalSearchSolver, nb:integer) : void`
a method for controlling the behavior of the algorithm whenever it runs into a solution. The solution may be stored in a historical record of best solutions encountered, or not. `nb` is the maximal size of the historical buffer (the `nb` last solutions will be kept). Therefore, with `nb=0` (the default setting), no solutions are stored and the only thing done by the engine when it finds a solution is printing out a message.
- `setMaxPrintDepth(gs:GlobalSearchSolver, n:integer) : void`
generates tracing statements up to depth `n` (the actual trace printing is controlled through the global `SVIEW` verbosity parameter).
- `setVarsToShow(gs:GlobalSearchSolver, l:list[IntVar]) : void`

records that, upon each solution, the domain of all variables from the list l should be displayed. Note that if the global search algorithm instantiates all variables in l , these domains will be singletons and the variable assignments will be shown.

6.4. Limiting the search space

Limits may be imposed on the search algorithm to avoid spending too much time in the exploration:

- `setNodeLimit(algo:GlobalSearchSolver, n:integer) : void`
stops the search algorithm after n nodes have been expanded
- `setTimeLimit(algo:GlobalSearchSolver, n:integer) : void`
stops the search algorithm after n milliseconds have been spent searching
- `setBacktrackLimit(algo:GlobalSearchSolver, n:integer) : void`
stops the search algorithm after n backtracks to a father node.

6.5. Defining your own tree search

The construction of the search tree can also be modified. By default, the search algorithm follows a three steps process that goes as follows:

1. First, settle all disjunctions
2. Then, as long as there exist no domain with few values, select a domain and try to restrict it to its lower (resp. upper) half
3. Last, instantiate variables.

This process is described by a sequence of *Branching* objects (that play the role of rules to achieve intermediate goals in logic programming). The user may specify that another sequence of *Branching* objects should be used to build the search tree. This can be done with the following primitives:

- `makeGlobalSearchSolver(allSolutions:boolean, l:list[AbstractBranching]) : GlobalSearchSolver`
- `makeGlobalSearchMaximizer(objective:IntVar, restartSearch:boolean, b:list[AbstractBranching]) : GlobalSearchSolver`
- `makeGlobalSearchMinimizer(objective:IntVar, restartSearch:boolean, b:list[AbstractBranching]) : GlobalSearchSolver`

These primitives create global search solvers that explore trees based on the sequence of branching objects b . The first levels of the search tree are structured using the first branching object of the list; then the second object is used, and so on. The list of branchings therefore describes a kind of hierarchical search procedure.

- `makeDefaultBranchingList(pb:Problem) : list[AbstractBranching]`
returns the list of branching objects used in Choco's default search strategy can be retrieved by the following method (ie: it returns a list of three objects: the first one for settling disjunctions, the second one for splitting large domains, the third one for assigning variables).

Choco provides the following *Branching* objects:

- `makeDisjunctionBranching(pb:Problem) : AbstractBranching`
creates a *Branching* object implementing a search scheme based on selecting suspended binary disjunctions (disjunctions for which the solver has not decided yet which of the branches is posted as a constraint) and enforcing the constraint from either branch.
- `makeAssignVarBranching() : AssignVar`
creates a *Branching* object implementing a search scheme based on assigning variables: at each choice point, an un-instantiated variable x (e.g. a variable with more than one value in its domain) is selected and a branch is created for each value v of the domain of x : in the branch, the assignment constraint $x==v$ is posted and propagated.

By default, the variable is selected as to minimize the number of values in the domain (the so called "first fail" variable ordering heuristic), and the values are tried in order of increasing values.

- `makeSplitDomBranching() : SplitDomain`
creates a *Branching* object implementing a binary search scheme based on splitting the domains of the variables: at each choice point, an un-instantiated variable x is selected, a value v is selected from its domain, and two branches are created: one with the constraint $x \leq v$ and one with the constraint $x > v$.
By default, the variable is selected as to minimize the number of values in the domain, and value is selected as the mean of the domain.

Choco also provides means of composing search trees by specifying the heuristics used for selecting variables and values. The previous primitives can use as parameters *VarSelector*, *ValIterator* and *ValSelector* objects in order to specify the heuristics used:

- `makeAssignVarBranching(varh:VarSelector, valh:ValIterator) : AssignVar`
creates a *Branching* object implementing a search scheme based on assigning variables. Both the variable heuristic (for choosing the variable to instantiate) and the value heuristic (specifying the order in which values from the domain are iterated) are passed as parameters.
- `makeAssignVarBranching(varh:VarSelector) : AssignVar`
similar primitive; but values from the domain are considered in increasing order.
- `makeSplitDomBranching(varh:VarSelector, valh:ValSelector, threshold:integer) : SplitDomain`
creates a *Branching* object implementing a binary search scheme based on splitting the domains of the variables. Both the variable heuristic (for choosing the variable to instantiate) and the value heuristic (for choosing the pivot value for splitting the domain) are passed as parameters. Splitting occurs only when the domain of the variable features more than *threshold* values.
- `makeSplitDomBranching(varh:VarSelector, threshold:integer) : SplitDomain`
similar primitive; but values from the pivot value is selected as the mean of the domain.
- `makeSplitDomBranching(varh:VarSelector) : SplitDomain`
similar primitive; but the threshold is set to 5.
- `makeAssignOrForbidBranching(varh:VarSelector, valh:ValSelector) : AssignOrForbid`
creates a *Branching* object implementing a binary search scheme based on variable x and a value v and considering two branches, one with the constraint $x==v$ and the other with constraint $x!=v$. The heuristics for selecting x and v are passed as parameters.

Choco comes with a small library of well-known heuristics for selecting variables and values.

Four heuristics for selecting a branching variables are available (*VarSelector*):

- `makeStaticVarHeuristic(l:list[IntVar]) : VarSelector`
static heuristic, selects the uninstantiated variables according the l sequence
- `makeMinDomVarHeuristic() : VarSelector`
default heuristic, selects the uninstantiated variable with smallest domain
- `makeMaxDegVarHeuristic() : VarSelector`
selects the uninstantiated variable with highest degree (greatest number of constraints involving the variable)
- `makeDomDegVarHeuristic() : VarSelector`
selects the uninstantiated variable minimizing the ratio of the number of values remaining in the domain divided by the number of constraints involving the variable (a combination of the two previous heuristics).

Two heuristics for iterating values in a domain are available (*ValIterator*):

- `makeIncValIterator() : ValIterator`
iterates values in the domain in ascending order
- `makeDecValIterator() : ValIterator`
iterates values in the domain in descending order

Three heuristics for picking a variable in a domain are available (*ValSelector*):

- `makeMinValSelector() : ValSelector`
selects the minimal value in the domain
- `makeMaxValSelector() : ValSelector`
selects the maximal value in the domain
- `makeMidValSelector() : ValSelector`
selects the mean value in the domain

You may also extend this small library of branching schemes and heuristics by defining your own subclasses of *AbstractBranching*, *VarSelector*, *ValIterator* and *ValSelector*. For more explanations on how to derive new branching objects, we refer the reader to the tutorial and the reference manual.

6.6. Local search

Some things are available but not yet publicly documented. To come in a future version...

7. Tracing and debugging

7.1. Verbosity levels

In order to display more information on its behavior, the CHOCO library provides the user with several verbosity level that can be set to a value below that of `system.verbose` in order to produce tracing information:

- `claire/PVIEW:integer` (default value 4)
general propagation info (layered fix-points)
- `claire/PTALK:integer` (default value 5)
tracing propagation (event queues)
- `claire/SVIEW:integer` (default value 1)
general search info (solutions found)
- `claire/STALK:integer` (default value 2)
tracing decisions in the search tree
- `claire/SDEBUG:integer` (default value 3)
debugging search tree construction (variable selection heuristics)
- `claire/IVIEW:integer` (default value 1)
tracing information on invariants (used for local moves). Traces general info about conflict counts
- `claire/ITALK:integer` (default value 2)
tracing improvement of conflict counts
- `claire/LVIEW:integer` (default value 1)
local search: Tracing general info about iterations (good solutions found)
- `claire/LTALK:integer` (default value 2)
local search: Tracing assignments and value flips.

7.2. Saving and retrieving information to/from files

It is a known shortcoming of Choco, not to provide facilities for saving problems, states, computations onto files. Tools for supporting this should come in a future version