Numberjack User Guide

May 27, 2013

1 Variables

Constructor for the class `Variable`:

Constructor	Object
`Variable()`	Binary variable
`Variable(N)`	Variable in the domain of {`0, N-1`}
`Variable('x')`	Binary variable called `'x'`
`Variable(N, 'x')`	Variable in the domain of {`0, N-1`} called `'x'`
`Variable(l,u)`	Variable in the domain of {`l, u`}
`Variable(l,u, 'x')`	Variable in the domain of {`l, u`} called `'x'`
`Variable(list)`	Variable with domain specified as a list
`Variable(list, 'x')`	Variable with domain specified as a list called `'x'`

The class `VarArray` represents a list of `Variables`.

Constructor	Object
`VarArray(l)`	creates an array from a list `l`
`VarArray(n)`	creates an array of `n` Boolean variables
`VarArray(n, 'x')`	creates an array of `n` Boolean variables with names `'x0..xn-1'`
`VarArray(n, m, 'x')`	creates an array of `n` variables with domains `[0..m-1]` and names `'x0..xn-1'`
`VarArray(n, m)`	creates an array of `n` variables with domains `[0..m-1]`
`VarArray(n, d, 'x')`	creates an array of `n` variables with domains `d` and names `'x0..xn-1'`
`VarArray(n, d)`	creates an array of `n` variables with domains `d`
`VarArray(n, l, u, 'x')`	creates an array of `n` variables with domains `[l..u]` and names `'x0..xn-1'`
`VarArray(n, l, u)`	creates an array of `n` variables with domains `[l..u]`

The class `Matrix` represents a 2-dimensional array of `Variables`.

Constructor	Object
`Matrix(l)`	creates a Matrix from a list `l`
`Matrix(n, m)`	creates a `n x m` Matrix of Boolean variables
`Matrix(n, m, 'x')`	creates a `n x m` Matrix of Boolean variables with names `'x0.0..xn-1.m-1'`
`Matrix(n, m, u)`	creates a `n x m` Matrix of variables with domains `[0..u-1]`
`Matrix(n, m, u, 'x')`	creates a `n x m` Matrix of variables with domains `[0..u-1]` and names `'x0.0..xn-1.m-1'`
`Matrix(n, m, l, u)`	creates a `n x m` Matrix of variables with domains `[l..u]`
`Matrix(n, m, l, u, 'x')`	creates a `n x m` Matrix of variables with domains `[l..u]` and names `'x0.0..xn-1.m-1'`

Operators

These use the infix notation ($x \oplus y$ where $\oplus$ is an operator). They return an `Expression` object that can be constrained. Operators in the first table must be used as expressions in another constraint.

Symbol	Arguments	Value
$+$	`Expression`/`Integer` x, `Expression`/`Integer` y	an `Expression` constrained to be equal to $x + y$
$-$	`Expression`/`Integer` x, `Expression`/`Integer` y	an `Expression` constrained to be equal to $x - y$
$*$	`Expression`/`Integer` x, `Expression`/`Integer` y	an `Expression` constrained to be equal to $x \times y$
$/$	`Expression`/`Integer` x, `Expression`/`Integer` y	an `Expression` constrained to be equal to x/y
$\%$	`Expression`/`Integer` x, `Expression`/`Integer` y	an `Expression` constrained to be equal to $x \bmod y$

Operators of the second table may be posted as constraints. In this case, if x is the returned `Expression`, the posted constraint will have the semantic $x \neq 0$ (i.e., x is `True`).

Symbol	Arguments	Value
$==$	`Expression`/`Integer` x, `Expression`/`Integer` y	a (Boolean) `Expression` constrained to be 1 iff $x = y$
$!=$	`Expression`/`Integer` x, `Expression`/`Integer` y	a (Boolean) `Expression` constrained to be 1 iff $x \neq y$
$<=$	`Expression`/`Integer` x, `Expression`/`Integer` y	a (Boolean) `Expression` constrained to be 1 iff $x \leq y$
$<$	`Expression`/`Integer` x, `Expression`/`Integer` y	a (Boolean) `Expression` constrained to be 1 iff $x < y$
$>=$	`Expression`/`Integer` x, `Expression`/`Integer` y	a (Boolean) `Expression` constrained to be 1 iff $x \geq y$
$>$	`Expression`/`Integer` x, `Expression`/`Integer` y	a (Boolean) `Expression` constrained to be 1 iff $x > y$
$\vert$	`Expression`/`Integer` x, `Expression`/`Integer` y	a (Boolean) `Expression` constrained to be 1 iff $x \neq 0$ or $y \neq 0$
$\&$	`Expression`/`Integer` x, `Expression`/`Integer` y	a (Boolean) `Expression` constrained to be 1 iff $x \neq 0$ and $y \neq 0$

Functions

These are used as function (foo(args), where foo is the function and args the argunents). They return an `Expression` object that must be constrained.

Symbol	Arguments	Value
`Abs`	an `Expression` x	an `Expression` constrained to be equal to $\|x\|$
`Neg`	an `Expression` x	an `Expression` constrained to be equal to $-x$
`Sum`	a list of `Expressions` $[x_1, \ldots, x_n]$ a list of `Integers` $[a_1, \ldots, a_n]$ (default $[1, \ldots, 1]$)	an `Expression` constrained to be equal to $\sum_{i=1}^{n} a_i x_i$
`Min`	a list of `Expressions` $[x_1, \ldots, x_n]$	an `Expression` constrained to be equal to $\min_{1 \leq i \leq n} x_i$
`Max`	a list of `Expressions` $[x_1, \ldots, x_n]$	an `Expression` constrained to be equal to $\max_{1 \leq i \leq n} x_i$
`Element*`	a list of `Expressions` $[x_1, \ldots, x_n]$ and an `Expression` y	an `Expression` constrained to be equal to x_y
	(*) Can be used with the square brackets operator is $[x_1, \ldots, x_n]$ is a `VarArray` X as follows: $X[y]$	

Constraints

These are used as function (foo(args), where foo is the function and args the argunents), they are not expressions and cannot be constrained.

Symbol	Arguments	Value
`AllDiff`	a `List` of `Expressions` $[x_1, \ldots, x_n]$	Constrains the variables $[x_1, \ldots, x_n]$ to take pairwise distinct values
`Gcc`	a `List` of `Expressions` $[x_1, \ldots, x_n]$ and a `Dictionary` mapping each value v_j in $\bigcup_{1 \leq i \leq n} D(x_i)$ to a pair (l_j, u_j)	Constrains each value v_j in $\bigcup_{1 \leq i \leq n} D(x_i)$ to appear between l_j and u_j times in the sequence $[x_1, \ldots, x_n]$

Objectives

These are used as function (foo(args), where foo is the function and args the argunents), they are not expressions and cannot be constrained. Only one objective can be added to the model.

Symbol	Arguments	Value
`Maximise`	an `Expression` x	Indicates that the value of x should be maximised
`Minimise`	an `Expression` x	Indicates that the value of x should be minimised

Variable Heuristics

Set by the method `setHeuristic(var-order, val-order, randomization)` of `Solver`.
The possible arguments for "var-order" are:

Symbol	Effect
`'Random'`	Branch on variables according to the input order
`'Lex'`	Branch on variables according to the input order
`'AntiLex'`	Branch on variables according to the inverse of input order
`'MaxDegree'`	Branch on the variable of highest dynamic degree first
`'MinDomain'`	Branch on the variable with smallest domain first
`'MinDomainMinVal'`	Branch on the variable with smallest domain first, ties broken by minium min value
`'MinDomainMaxDegree'`	Branch on the variable with smallest domain first, ties broken by dynamic degree
`'DomainOverDegree'`	Branch on the variable with smallest ratio (domain size / degree)
`'DomainOverWDegree'`	Branch on the variable with smallest ratio (domain size / weighted degree)
`'Ngihbour'`	Branch on the variable average (domain size / degree) over neighbouring variables
`'Impact'`	Branch on the variable of highest impact
`'ImpactOverDegree'`	Branch on the variable with smallest ratio (1 / (impact * degree))
`'ImpactOverDegree'`	Branch on the variable with smallest ratio (1 / (impact * weighted degree))

The possible arguments for "val-order" are:

Symbol	Effect
`'Lex'`	Select the minimum value in the domain
`'AntiLex'`	Select the maximum value in the domain
`'Random'`	Select a value randomly with uniform probability
`'RandomMinMax'`	Select either the minimum of maximum value randomly with uniform probability
`'DomainSplit'`	Reduce the domain splitting around the average of the bounds
`'RandomSplit'`	Reduce the domain splitting around a random value
`'Impact'`	Select the value with minimum impact

The randomization arguments indicates how many variables should be selected. The final choice is made randomly between them

Solving Methods

The standard way of calling the solver is the method `solve(X)` where `X` is a list of variables (or a `VarArray` or a `Matrix`). If no value is given for `X`, all variables are branched on. It return `True` is a solution was found and `False` otherwise.

The method `solveAndRestart(X)` works similarly as `solve(X)`, except that the search will be restarted after a number of failures.

The methods `startNewSearch(X)` and `getNextSolution()` allow to find sevral solutions. `startNewSearch(X)` must be called once to initialise the procedure, then `getNextSolution()` can be called, finding a new solution at each call.