toulbar2 API Reference

Release 1.0.0

INRAE

Feb 28, 2022

CONTENTS

toulbar2 is an open-source C++ solver for cost function networks.

See : `Class Diagram`.

CHAPTER

ONE

WEIGHTEDCSP CLASS

class **WeightedCSP**

Abstract class *WeightedCSP* representing a weighted constraint satisfaction problem

- problem lower and upper bounds
- list of variables with their finite domains (either represented by an enumerated list of values, or by a single interval)
- list of cost functions (created before and during search by variable elimination of variables with small degree)
- local consistency propagation (variable-based propagation) including cluster tree decomposition caching (separator-based cache)

Note: Variables are referenced by their lexicographic index number (as returned by *eg* *WeightedCSP::makeEnumeratedVariable*)

Note: Cost functions are referenced by their lexicographic index number (as returned by *eg* WeightedCSP::postBinaryConstraint)

Public Functions

virtual int **getIndex**() const = 0

instantiation occurrence number of current WCSP object

virtual string **getName**() const = 0

get WCSP problem name (defaults to filename with no extension)

virtual void **setName**(const string &problem) = 0

set WCSP problem name

virtual void ***getSolver**() const = 0

special hook to access solver information

virtual Cost **getLb**() const = 0

gets internal dual lower bound

virtual Cost **getUb**() const = 0

gets internal primal upper bound

virtual Double **getDPrimalBound**() const = 0
gets problem primal bound as a Double representing a decimal cost (upper resp. lower bound for minimization resp. maximization)

virtual Double **getDDualBound**() const = 0
gets problem dual bound as a Double representing a decimal cost (lower resp. upper bound for minimization resp. maximization)

virtual Double **getDLb**() const = 0
gets problem lower bound as a Double representing a decimal cost

virtual Double **getDUb**() const = 0
gets problem upper bound as a Double representing a decimal cost

virtual void **updateUb**(Cost newUb) = 0
sets initial problem upper bound and each time a new solution is found

virtual void **enforceUb**() = 0
enforces problem upper bound when exploring an alternative search node

virtual void **increaseLb**(Cost addLb) = 0
increases problem lower bound thanks to *eg* soft local consistencies

Parameters **addLb** – increment value to be **added** to the problem lower bound

virtual void **decreaseLb**(Cost shift) = 0
shift problem optimum toward negative costs

Parameters **shift** – positive shifting value to be subtracted to the problem optimum when printing the solutions

virtual Cost **getNegativeLb**() const = 0
gets constant term used to subtract to the problem optimum when printing the solutions

virtual Cost **finiteUb**() const = 0
computes the worst-case assignment finite cost (sum of maximum finite cost over all cost functions plus one)

> **Warning:** current problem should be completely loaded and propagated before calling this function

Returns the worst-case assignment finite cost

virtual void **setInfiniteCost**() = 0
updates infinite costs in all cost functions accordingly to the problem global lower and upper bounds

> **Warning:** to be used in preprocessing only

virtual bool **enumerated**(int varIndex) const = 0
true if the variable has an enumerated domain

virtual string **getName**(int varIndex) const = 0

Note: by default, variables names are integers, starting at zero

---

virtual unsigned int **getVarIndex**(const string &s) const = 0
return variable index from its name, or *numberOfVariables()* if not found

virtual Value **getInf**(int varIndex) const = 0
minimum current domain value

virtual Value **getSup**(int varIndex) const = 0
maximum current domain value

virtual Value **getValue**(int varIndex) const = 0
current assigned value

> **Warning:** undefined if not assigned yet

virtual unsigned int **getDomainSize**(int varIndex) const = 0
current domain size

virtual vector<Value> **getEnumDomain**(int varIndex) = 0
gets current domain values in an array

virtual vector<pair<Value, Cost>> **getEnumDomainAndCost**(int varIndex) = 0
gets current domain values and unary costs in an array

virtual unsigned int **getDomainInitSize**(int varIndex) const = 0
gets initial domain size (warning! assumes EnumeratedVariable)

virtual Value **toValue**(int varIndex, unsigned int idx) = 0
gets value from index (warning! assumes EnumeratedVariable)

virtual unsigned int **toIndex**(int varIndex, Value value) = 0
gets index from value (warning! assumes EnumeratedVariable)

virtual unsigned int **toIndex**(int varIndex, const string &valueName) = 0
gets index from value name (warning! assumes EnumeratedVariable with value names)

virtual int **getDACOrder**(int varIndex) const = 0
index of the variable in the DAC variable ordering

virtual Value **nextValue**(int varIndex, Value v) const = 0
first value after v in the current domain or v if there is no value

virtual void **increase**(int varIndex, Value newInf) = 0
changes domain lower bound

virtual void **decrease**(int varIndex, Value newSup) = 0
changes domain upper bound

virtual void **assign**(int varIndex, Value newValue) = 0
assigns a variable and immediately propagates this assignment

virtual void **remove**(int varIndex, Value remValue) = 0
removes a domain value (valid if done for an enumerated variable or on its domain bounds)

virtual void **assignLS**(vector<int> &varIndexes, vector<Value> &newValues, bool force = false) = 0
assigns a set of variables at once and propagates (used by Local Search methods such as Large Neighborhood Search)

Parameters

- **varIndexes** – vector of variable indexes as returned by makeXXXVariable
- **newValues** – vector of values to be assigned to the corresponding variables

virtual Cost **getUnaryCost**(int varIndex, Value v) const = 0
unary cost associated to a domain value

virtual Cost **getMaxUnaryCost**(int varIndex) const = 0
maximum unary cost in the domain

virtual Value **getMaxUnaryCostValue**(int varIndex) const = 0
a value having the maximum unary cost in the domain

virtual Value **getSupport**(int varIndex) const = 0
NC/EAC unary support value.

virtual Value **getBestValue**(int varIndex) const = 0
hint for some value ordering heuristics (only used by RDS)

virtual void **setBestValue**(int varIndex, Value v) = 0
hint for some value ordering heuristics (only used by RDS)

virtual bool **getIsPartOfOptimalSolution**() = 0
special flag used for debugging purposes only

virtual void **setIsPartOfOptimalSolution**(bool v) = 0
special flag used for debugging purposes only

virtual int **getDegree**(int varIndex) const = 0
approximate degree of a variable (*ie* number of active cost functions, see Variable elimination)

virtual int **getTrueDegree**(int varIndex) const = 0
degree of a variable

virtual Long **getWeightedDegree**(int varIndex) const = 0
weighted degree heuristic

virtual void **resetWeightedDegree**() = 0
initialize weighted degree heuristic

virtual void **resetTightness**() = 0
initialize constraint tightness used by some heuristics (including weighted degree)

virtual void **resetTightnessAndWeightedDegree**() = 0
initialize tightness and weighted degree heuristics

virtual void **preprocessing**() = 0
applies various preprocessing techniques to simplify the current problem

virtual void **sortConstraints**() = 0
sorts the list of cost functions associated to each variable based on smallest problem variable indexes

---

Note: must be called after creating all the cost functions and before solving the problem

---

Warning: side-effect: updates DAC order according to an existing variable elimination order

virtual void **whenContradiction**() = 0
after a contradiction, resets propagation queues

virtual void **propagate**() = 0
propagates until a fix point is reached (or throws a contradiction)

virtual bool **verify**() = 0
checks the propagation fix point is reached

virtual unsigned int **numberOfVariables**() const = 0
number of created variables

virtual unsigned int **numberOfUnassignedVariables**() const = 0
current number of unassigned variables

virtual unsigned int **numberOfConstraints**() const = 0
initial number of cost functions (before variable elimination)

virtual unsigned int **numberOfConnectedConstraints**() const = 0
current number of cost functions

virtual unsigned int **numberOfConnectedBinaryConstraints**() const = 0
current number of binary cost functions

virtual unsigned int **medianDomainSize**() const = 0
median current domain size of variables

virtual unsigned int **medianDegree**() const = 0
median current degree of variables

virtual unsigned int **medianArity**() const = 0
median arity of current cost functions

virtual unsigned int **getMaxDomainSize**() const = 0
maximum initial domain size found in all variables

virtual unsigned int **getMaxCurrentDomainSize**() const = 0
maximum current domain size found in all variables

virtual unsigned int **getDomainSizeSum**() const = 0
total sum of current domain sizes

virtual void **cartProd**(BigInteger &cartesianProduct) = 0
Cartesian product of current domain sizes.

Parameters **cartesianProduct** – result obtained by the GNU Multiple Precision Arithmetic Library GMP

virtual Long **getNbDEE**() const = 0
number of value removals due to dead-end elimination

virtual int **makeEnumeratedVariable**(string n, Value iinf, Value isup) = 0
create an enumerated variable with its domain bounds

virtual int **makeEnumeratedVariable**(string n, vector<Value> &dom) = 0
create an enumerated variable with its domain values

virtual void **addValueName**(int xIndex, const string &valuename) = 0
add next value name

> **Warning:** should be called on EnumeratedVariable object as many times as its number of initial domain values

virtual int **makeIntervalVariable**(string n, Value iinf, Value isup) = 0
create an interval variable with its domain bounds

virtual int **postNaryConstraintBegin**(int *scope, int arity, Cost defval, Long nbtuples = 0, bool forcenary = false) = 0

> **Warning:** must call WeightedCSP::postNaryConstraintEnd after giving cost tuples

virtual int **postUnary**(int xIndex, Value *d, int dsize, Cost penalty) = 0

> **Warning:** must call *WeightedCSP::sortConstraints* after all cost functions have been posted (see *WeightedCSP::sortConstraints*)

virtual int **postWAmong**(vector<int> &scope, const string &semantics, const string &propagator, Cost baseCost, const vector<Value> &values, int lb, int ub) = 0

post a soft among cost function

Parameters

- **scopeIndex** – an array of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **arity** – the size of the array
- **semantics** – the semantics of the global cost function: “var” or — “hard” or “lin” or “quad” (network-based propagator only)—
- **propagator** – the propagation method (only “DAG” or “network”)
- **baseCost** – the scaling factor of the violation
- **values** – a vector of values to be restricted
- **lb** – a fixed lower bound for the number variables to be assigned to the values in *values*
- **ub** – a fixed upper bound for the number variables to be assigned to the values in *values*

post a soft weighted among cost function

virtual void **postWVarAmong**(vector<int> &scope, const string &semantics, Cost baseCost, vector<Value> &values, int varIndex) = 0

post a weighted among cost function with the number of values encoded as a variable with index *varIndex* (*network-based* propagator only)

virtual int **postWRegular**(vector<int> &scope, const string &semantics, const string &propagator, Cost baseCost, int nbStates, const vector<WeightedObjInt> &initial_States, const vector<WeightedObjInt> &accepting_States, const vector<DFATransition> &Wtransitions) = 0

post a soft or weighted regular cost function

> **Warning:** Weights are ignored in the current implementation of DAG and flow-based propagators post a soft weighted regular cost function

Parameters

- **scopeIndex** – an array of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **arity** – the size of the array
- **semantics** – the semantics of the soft global cost function: “var” or “edit” (flow-based propagator) or — “var” (DAG-based propagator)— (unused parameter for network-based propagator)

- **propagator** – the propagation method (“flow”, “DAG”, “network”)
- **baseCost** – the scaling factor of the violation (“flow”, “DAG”)
- **nbStates** – the number of the states in the corresponding DFA. The states are indexed as 0, 1, …, nbStates-1
- **initial_States** – a vector of WeightedObjInt specifying the starting states with weight
- **accepting_States** – a vector of WeightedObjInt specifying the final states
- **Wtransitions** – a vector of (weighted) transitions

virtual int **postWAllDiff**(int *scopeIndex, int arity, const string &semantics, const string &propagator, Cost baseCost) = 0

post a soft alldifferent cost function

Parameters

- **scopeIndex** – an array of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **arity** – the size of the array
- **semantics** – the semantics of the global cost function: for flow-based propagator: “var” or “dec” or “decbi” (decomposed into a binary cost function complete network), for DAG-based propagator: “var”, for network-based propagator: “hard” or “lin” or “quad” (decomposed based on wamong)
- **propagator** – the propagation method (“flow”, “DAG”, “network”)
- **baseCost** – the scaling factor of the violation

virtual int **postWGcc**(int *scopeIndex, int arity, const string &semantics, const string &propagator, Cost baseCost, const vector<BoundedObjValue> &values) = 0

post a soft global cardinality cost function

Parameters

- **scopeIndex** – an array of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **arity** – the size of the array
- **semantics** – the semantics of the global cost function: “var” (DAG-based propagator only) or — “var” or “dec” or “wdec” (flow-based propagator only) or — “hard” or “lin” or “quad” (network-based propagator only)—
- **propagator** – the propagation method (“flow”, “DAG”, “network”)
- **baseCost** – the scaling factor of the violation
- **values** – a vector of BoundedObjValue, specifying the lower and upper bounds of each value, restricting the number of variables can be assigned to them

virtual int **postWSame**(int *scopeIndexG1, int arityG1, int *scopeIndexG2, int arityG2, const string &semantics, const string &propagator, Cost baseCost) = 0

post a soft same cost function (a group of variables being a permutation of another group with the same size)

Parameters

- **scopeIndexG1** – an array of the first group of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*

- **arityG1** – the size of *scopeIndexG1*
- **scopeIndexG2** – an array of the second group of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **arityG2** – the size of *scopeIndexG2*
- **semantics** – the semantics of the global cost function: “var” or — “hard” or “lin” or “quad” (network-based propagator only)—
- **propagator** – the propagation method (“flow” or “network”)
- **baseCost** – the scaling factor of the violation.

virtual void **postWSameGcc**(int *scopeIndex, int arity, string semantics, Cost baseCost, Value *values, int nbValues, int *lb, int *ub) = 0

post a combination of a same and gcc cost function decomposed as a cost function network

virtual int **postWGrammarCNF**(int *scopeIndex, int arity, const string &semantics, const string &propagator, Cost baseCost, int nbSymbols, int startSymbol, const vector<CFGProductionRule> WRuleToTerminal) = 0

post a soft/weighted grammar cost function with the dynamic programming propagator and grammar in Chomsky normal form

Parameters

- **scopeIndex** – an array of the first group of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **arity** – the size of *scopeIndex*
- **semantics** – the semantics of the global cost function: “var” or “weight”
- **propagator** – the propagation method (“DAG” only)
- **baseCost** – the scaling factor of the violation
- **nbSymbols** – the number of symbols in the corresponding grammar. Symbols are indexed as 0, 1, . . . , nbSymbols-1
- **startSymbol** – the index of the starting symbol
- **WRuleToTerminal** – a vector of *::CFGProductionRule*. Note that:
  - if *order* in *CFGProductionRule* is set to 0, it is classified as A -> v, where A is the index of the terminal symbol and v is the value.
  - if *order* in *CFGProductionRule* is set to 1, it is classified as A -> BC, where A,B,C the index of the nonterminal symbols.
  - if *order* in *CFGProductionRule* is set to 2, it is classified as weighted A -> v, where A is the index of the terminal symbol and v is the value.
  - if *order* in *CFGProductionRule* is set to 3, it is classified as weighted A -> BC, where A,B,C the index of the nonterminal symbols.
  - if *order* in *CFGProductionRule* is set to values greater than 3, it is ignored.

virtual int **postMST**(int *scopeIndex, int arity, const string &semantics, const string &propagator, Cost baseCost) = 0

post a Spanning Tree hard constraint

Parameters

- **scopeIndex** – an array of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*

- **arity** – the size of *scopeIndex*
- **semantics** – the semantics of the global cost function: “hard”
- **propagator** – the propagation method (“DAG” only)
- **baseCost** – unused in the current implementation (MAX_COST)

virtual int **postMaxWeight**(int *scopeIndex, int arity, const string &semantics, const string &propagator, Cost baseCost, const vector<WeightedVarValPair> weightFunction) = 0

post a weighted max cost function (maximum cost of a set of unary cost functions associated to a set of variables)

Parameters

- **scopeIndex** – an array of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **arity** – the size of *scopeIndex*
- **semantics** – the semantics of the global cost function: “val”
- **propagator** – the propagation method (“DAG” only)
- **baseCost** – if a variable-value pair does not exist in *weightFunction*, its weight will be mapped to baseCost.
- **weightFunction** – a vector of WeightedVarValPair containing a mapping from variable-value pairs to their weights.

virtual void **postWSum**(int *scopeIndex, int arity, string semantics, Cost baseCost, string comparator, int rightRes) = 0

post a soft linear constraint with unit coefficients

Parameters

- **scopeIndex** – an array of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **arity** – the size of *scopeIndex*
- **semantics** – the semantics of the global cost function: “hard” or “lin” or “quad” (network-based propagator only)
- **propagator** – the propagation method (“network” only)
- **baseCost** – the scaling factor of the violation
- **comparator** – the comparison operator of the linear constraint (“==”, “!=”, “<”, “<=”, “>,”, “>=”)
- **rightRes** – right-hand side value of the linear constraint

virtual void **postWVarSum**(int *scopeIndex, int arity, string semantics, Cost baseCost, string comparator, int varIndex) = 0

post a soft linear constraint with unit coefficients and variable right-hand side

virtual void **postWOverlap**(int *scopeIndex, int arity, string semantics, Cost baseCost, string comparator, int rightRes) = 0

post a soft overlap cost function (a group of variables being point-wise equivalent — and not equal to zero — to another group with the same size)

Parameters

- **scopeIndex** – an array of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*

- **arity** – the size of *scopeIndex* (should be an even value)
- **semantics** – the semantics of the global cost function: "hard" or "lin" or "quad" (network-based propagator only)
- **propagator** – the propagation method ("network" only)
- **baseCost** – the scaling factor of the violation.
- **comparator** – the point-wise comparison operator applied to the number of equivalent variables ("==", "!=", "<", "<=", ">,", ">=")
- **rightRes** – right-hand side value of the comparison

virtual void **postWDivConstraint**(vector<int> &scope, unsigned int distance, vector<Value> &values, int method = 0) = 0

post a diversity Hamming distance constraint between a list of variables and a given fixed assignment

Note: depending on the decomposition method, it adds dual and/or hidden variables

Parameters

- **scope** – a vector of variable indexes as returned by *WeightedCSP::makeEnumeratedVariable*
- **distance** – the Hamming distance minimum bound
- **values** – a vector of values (same size as scope)
- **method** – the network decomposition method (0:Dual, 1:Hidden, 2:Ternary)

virtual vector<vector<int>> ***getListSuccessors**() = 0

generating additional variables vector created when berge decomposition are included in the WCSP

virtual vector<int> **getBergeDecElimOrder**() = 0

return an elimination order compatible with Berge acyclic decomposition of global decomposable cost functions (if possible keep reverse of previous DAC order)

virtual void **setDACOrder**(vector<int> &elimVarOrder) = 0

change DAC order and propagate from scratch

virtual bool **isGlobal**() = 0

true if there are soft global constraints defined in the problem

virtual Cost **read_wcsp**(const char *fileName) = 0

load problem in all format supported by toulbar2. Returns the UB known to the solver before solving (file and command line).

virtual void **read_legacy**(const char *fileName) = 0

load problem in wcsp legacy format

virtual void **read_uai2008**(const char *fileName) = 0

load problem in UAI 2008 format (see http://graphmod.ics.uci.edu/uai08/FileFormat and http://www.cs.huji.ac.il/project/UAI10/fileFormat.php)

Warning: UAI10 evidence file format not recognized by toulbar2 as it does not allow multiple evidence (you should remove the first value in the file)

virtual void **read_random**(int n, int m, vector<int> &p, int seed, bool forceSubModular = false, string globalname = "") = 0
create a random WCSP with *n* variables, domain size *m*, array *p* where the first element is a percentage of tuples with a nonzero cost and next elements are the number of random cost functions for each different arity (starting with arity two), random seed, a flag to have a percentage (last element in the array *p*) of the binary cost functions being permutated submodular, and a string to use a specific global cost function instead of random cost functions in extension

virtual void **read_wcnf**(const char *fileName) = 0
load problem in (w)cnf format (see http://www.maxsat.udl.cat/08/index.php?disp=requirements)

virtual void **read_qpbo**(const char *fileName) = 0
load quadratic pseudo-Boolean optimization problem in unconstrained quadratic programming text format (first text line with n, number of variables and m, number of triplets, followed by the m triplets (x,y,cost) describing the sparse symmetric nXn cost matrix with variable indexes such that x <= y and any positive or negative real numbers for costs)

virtual void **read_opb**(const char *fileName) = 0
load pseudo-Boolean optimization problem

virtual const vector<Value> **getSolution**() = 0
after solving the problem, return the optimal solution (warning! do not use it if doing solution counting or if there is no solution, see *WeightedCSPSolver::solve* output for that)

virtual Double **getSolutionValue**() const = 0
returns current best solution cost or MAX_COST if no solution found

virtual Cost **getSolutionCost**() const = 0
returns current best solution cost or MAX_COST if no solution found

virtual const vector<Value> **getSolution**(Cost *cost_ptr) = 0
returns current best solution and its cost

virtual void **initSolutionCost**() = 0
returns all solutions found

invalidate best solution by changing its cost to MAX_COST

virtual void **setSolution**(Cost cost, TAssign *sol = NULL) = 0
set best solution from current assigned values or from a given assignment (for BTD-like methods)

virtual void **printSolution**() = 0
prints current best solution on standard output (using variable and value names if cfn format and ToulBar2::showSolution>1)

virtual void **printSolution**(ostream &os) = 0
prints current best solution (using variable and value names if cfn format and ToulBar2::writeSolution>1)

virtual void **printSolution**(FILE *f) = 0
prints current best solution (using variable and value names if cfn format and ToulBar2::writeSolution>1)

virtual void **print**(ostream &os) = 0
print current domains and active cost functions (see Output messages, verbosity options and debugging)

virtual void **dump**(ostream &os, bool original = true) = 0
output the current WCSP into a file in wcsp format

Parameters

- **os** – output file
- **original** – if true then keeps all variables with their original domain size else uses unassigned variables and current domains recoding variable indexes

virtual void **dump_CFN**(ostream &os, bool original = true) = 0
output the current WCSP into a file in wcsp format

Parameters

- **os** – output file
- **original** – if true then keeps all variables with their original domain size else uses unassigned variables and current domains recoding variable indexes

virtual vector<Variable*> &**getDivVariables**() = 0
returns all variables on which a diversity request exists

Public Static Functions

static *WeightedCSP* ***makeWeightedCSP**(Cost upperBound, void *solver = NULL)
Weighted CSP factory.

CHAPTER

TWO

WEIGHTEDCSPSOLVER CLASS

class **WeightedCSPSolver**

Abstract class *WeightedCSPSolver* representing a WCSP solver

- link to a *WeightedCSP*
- generic complete solving method configurable through global variables (see ::ToulBar2 class and command line options)
- optimal solution available after problem solving
- elementary decision operations on domains of variables
- statistics information (number of nodes and backtracks)
- problem file format reader (multiple formats, see Weighted Constraint Satisfaction Problem file format (wcsp))
- solution checker (output the cost of a given solution)

Public Functions

virtual *WeightedCSP* ***getWCSP**() = 0

access to its associated Weighted CSP

virtual Long **getNbNodes**() const = 0

number of search nodes (see *WeightedCSPSolver::increase*, *WeightedCSPSolver::decrease*, *WeightedCSPSolver::assign*, *WeightedCSPSolver::remove*)

virtual Long **getNbBacktracks**() const = 0

number of backtracks

virtual void **increase**(int varIndex, Value value, bool reverse = false) = 0

changes domain lower bound and propagates

virtual void **decrease**(int varIndex, Value value, bool reverse = false) = 0

changes domain upper bound and propagates

virtual void **assign**(int varIndex, Value value, bool reverse = false) = 0

assigns a variable and propagates

virtual void **remove**(int varIndex, Value value, bool reverse = false) = 0

removes a domain value and propagates (valid if done for an enumerated variable or on its domain bounds)

virtual Cost **read_wcsp**(const char *fileName) = 0

reads a Cost function network from a file (format as indicated by ToulBar2:: global variables)

virtual void **read_random**(int n, int m, vector<int> &p, int seed, bool forceSubModular = false, string globalname = "") = 0
create a random WCSP, see *WeightedCSP::read_random*

virtual bool **solve**(bool first = true) = 0
simplifies and solves to optimality the problem

Warning: after solving, the current problem has been modified by various preprocessing techniques

Warning: DO NOT READ VALUES OF ASSIGNED VARIABLES USING *WeightedCSP::getValue* (temporally wrong assignments due to variable elimination in preprocessing) BUT USE *WeightedCSPSolver::getSolution* INSTEAD

Returns false if there is no solution found

virtual Cost **narycsp**(string cmd, vector<Value> &solution) = 0
solves the current problem using INCOP local search solver by Bertrand Neveu

Note: side-effects: updates current problem upper bound and propagates, best solution saved (using WCSP::setBestValue)

Warning: cannot solve problems with global cost functions

Parameters

- **cmd** – command line argument for narycsp INCOP local search solver (cmd format: lowerbound randomseed nbiterations method nbmoves neighborhoodchoice neighborhoodchoice2 minnbneighbors maxnbneighbors neighborhoodchoice3 autotuning tracemode)
- **solution** – best solution assignment found (MUST BE INITIALIZED WITH A DEFAULT COMPLETE ASSIGNMENT)

Returns best solution cost found

virtual bool **solve_symmax2sat**(int n, int m, int *posx, int *posy, double *cost, int *sol) = 0
quadratic unconstrained pseudo-Boolean optimization Maximize $h' \times W \times h$ where W is expressed by all its non-zero half squared matrix costs (can be positive or negative, with $\forall i, posx[i] \leq posy[i]$)

See also:

::solvesymmax2sat_ for Fortran call

Note: costs for $posx \neq posy$ are multiplied by 2 by this method

Note: by convention: $h = 1 \equiv x = 0$ and $h = -1 \equiv x = 1$

Warning: does not allow infinite costs (no forbidden assignments, unconstrained optimization)

Returns true if at least one solution has been found (array *sol* being filled with the best solution)

virtual void **dump_wcsp**(const char *fileName, bool original = true, ProblemFormat format = WCSP_FORMAT) = 0
output current problem in a file

See also:

WeightedCSP::dump

virtual void **read_solution**(const char *fileName, bool updateValueHeuristic = true) = 0
read a solution from a file

virtual void **parse_solution**(const char *certificate, bool updateValueHeuristic = true) = 0
read a solution from a string (see ToulBar2 option *-x*)

virtual const vector<Value> **getSolution**() = 0
after solving the problem, return the optimal solution (warning! do not use it if doing solution counting or if there is no solution, see *WeightedCSPSolver::solve* output for that)

virtual Double **getSolutionValue**() const = 0
after solving the problem, return the optimal solution value (can be an arbitrary real cost in minimization or preference in maximization, see CFN format) (warning! do not use it if doing solution counting or if there is no solution, see *WeightedCSPSolver::solve* output for that)

virtual Cost **getSolutionCost**() const = 0
after solving the problem, return the optimal solution nonnegative integer cost (warning! do not use it if doing solution counting or if there is no solution, see *WeightedCSPSolver::solve* output for that)

virtual Cost **getSolution**(vector<Value> &solution) const = 0
after solving the problem, add the optimal solution in the input/output vector and returns its optimum cost (warning! do not use it if doing solution counting or if there is no solution, see *WeightedCSPSolver::solve* output for that)

virtual vector<pair<Double, vector<Value>>> **getSolutions**() const = 0
after solving the problem, return all solutions found with their corresponding value

Public Static Functions

static *WeightedCSPSolver* ***makeWeightedCSPSolver**(Cost initUpperBound)
WeightedCSP Solver factory.

INDEX

W