C++ Library of toulbar2

Release 1.0.0

INRAE

Dec 05, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

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

See : `Class Diagram`.

CHAPTER

TWO

MAIN TYPES AND CONSTANTS

The main types are:

- ::Value : domain value
- ::Cost : cost value (exact type depends on compilation flag)
- ::Long : large integer (long long int)
- ::TProb : probability value (exact type depends on compilation flag)
- ::TLogProb : log probability value (exact type depends on compilation flag)
- ::Double : large float (long double)
- ::tValue : a short Value type for tuples
- ::Tuple : vector of tValues to encode tuples

Note: Compilation flag for Cost is: `INT_COST` (int), `LONGLONG_COST` (long long), or `PARETOPAIR_COST` (see ::ParetoPair)

Note: Compilation flag for TProb is: `DOUBLE_PROB` or `LONGDOUBLE_PROB`

Note: Compilation flag for T(Log)Prob is: `DOUBLE_PROB` or `LONGDOUBLE_PROB`

Warning: `PARETOPAIR_COST` is fragile.

Variables

const string **IMPLICIT_VAR_TAG** = "#"

Special character value at the beginning of a variable's name to identify implicit variables (i.e., variables which are not decision variables)

const string **HIDDEN_VAR_TAG** = "^"

Special character value at the beginning of a variable's name to identify hidden variables like diverse extra variables corresponding to the current sequence of diverse solutions found so far.

const string **HIDDEN_VAR_TAG_HVE** = "^c"

const string **HIDDEN_VAR_TAG_HVE_PRE** = "^!"

const Value **MAX_VAL** = (std::numeric_limits<Value>::max() / 2)
Maximum domain value.

const Value **WRONG_VAL** = std::numeric_limits<Value>::max()
Forbidden domain value.

const Value **MIN_VAL** = -(std::numeric_limits<Value>::max() / 2)
Minimum domain value.

const Value **MAX_DOMAIN_SIZE** = 10000
Maximum domain size

const int **NARYPROJECTIONSIZE** = 3

const unsigned int **NARYPROJECTION3MAXDOMSIZE** = 30

const Long **NARYDECONNECTSIZE** = 4

const int **MAX_BRANCH_SIZE** = 1000000

const ptrdiff_t **CHOICE_POINT_LIMIT** = SIZE_MAX

const ptrdiff_t **OPEN_NODE_LIMIT** = SIZE_MAX

const int **STORE_SIZE** = 16

const int **MAX_ELIM_BIN** = 1000000000

const int **MAX_ARITY** = 1000

const Long **MAX_NB_TUPLES** = 100000LL
Maximum number of tuples in n-ary cost functions.

const int **LARGE_NB_VARS** = 10000

const int **DECIMAL_POINT** = 3

const bool **NARY2CLAUSE** = true

const int **MAX_EAC_ITER** = 10000

CHAPTER

THREE

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

inline virtual **~WeightedCSP**()

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 **updateDUb**(Double newDUb) = 0

sets 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 **isfinite**() const = 0

returns true if any complete assignment using current domains is a valid tuple with finite cost (i.e., cost strictly less than the problem upper bound minus the lower bound)

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 bool **getEnumDomain**(int varIndex, Value *array) = 0

virtual vector<pair<Value, Cost>> **getEnumDomainAndCost**(int varIndex) = 0

gets current domain values and unary costs in an array

virtual bool **getEnumDomainAndCost**(int varIndex, ValueCost *array) = 0

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 bool **assigned**(int varIndex) const = 0

virtual bool **unassigned**(int varIndex) const = 0

virtual bool **canbe**(int varIndex, Value v) const = 0

virtual bool **cannotbe**(int varIndex, Value v) const = 0

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
- **force** – boolean if true then apply assignLS even if the variable is already assigned Note this function is equivalent but faster than a sequence of assign.

virtual void **assignLS**(int *varIndexes, Value *newValues, unsigned int size, bool dopropagate, bool force = false) = 0

virtual void **deconnect**(vector<int> &varIndexes) = 0

deconnects a set of variables from the rest of the problem and assigns them to their support value (used by Incremental Search)

Parameters

varIndexes – vector of variable indexes as returned by makeXXXVariable

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 **deactivatePropagate**() = 0
forbids propagate calls

virtual bool **isactivatePropagate**() = 0
are propagate calls authorized?

virtual void **reactivatePropagate**() = 0
re-authorizes propagate calls

virtual void **propagate**(bool fromscratch = false) = 0
(if authorized) propagates until a fix point is reached (or throws a contradiction). If fromscratch is true then propagates every cost function at least once.

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 const string &**getValueName**(int xIndex, Value value) = 0

return the name associated to a value as defined by addValueName or an empty string if no name found

virtual int **makeIntervalVariable**(string n, Value iinf, Value isup) = 0

create an interval variable with its domain bounds

virtual void **postNullaryConstraint**(Double cost) = 0

add a zero-arity cost function with floating-point cost

virtual void **postUnaryConstraint**(int xIndex, vector<Double> &costs, bool incremental = false) = 0

add a unary cost function with floating-point costs (if incremental is true then it disappears upon backtrack)

virtual int **postBinaryConstraint**(int xIndex, int yIndex, vector<Double> &costs, bool incremental = false) = 0

add a binary cost function with floating-point costs (if incremental is true then it disappears upon backtrack)

virtual int **postTernaryConstraint**(int xIndex, int yIndex, int zIndex, vector<Double> &costs, bool incremental = false) = 0

add a ternary cost function with floating-point costs (if incremental is true then it disappears upon backtrack)

virtual void **postNullaryConstraint**(Cost cost) = 0

virtual void **postUnary**(int xIndex, vector<Cost> &costs) = 0

virtual void **postUnaryConstraint**(int xIndex, vector<Cost> &costs) = 0

virtual void **postIncrementalUnaryConstraint**(int xIndex, vector<Cost> &costs) = 0

virtual int **postBinaryConstraint**(int xIndex, int yIndex, vector<Cost> &costs) = 0

virtual int **postIncrementalBinaryConstraint**(int xIndex, int yIndex, vector<Cost> &costs) = 0

virtual int **postTernaryConstraint**(int xIndex, int yIndex, int zIndex, vector<Cost> &costs) = 0

virtual int **postIncrementalTernaryConstraint**(int xIndex, int yIndex, int zIndex, vector<Cost> &costs) = 0

virtual int **postNaryConstraintBegin**(vector<int> &scope, Cost defval, Long nbtuples = 0, bool forcenary = !*NARY2CLAUSE*) = 0

virtual int **postNaryConstraintBegin**(int *scope, int arity, Cost defval, Long nbtuples = 0, bool forcenary = !*NARY2CLAUSE*) = 0

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

virtual void **postNaryConstraintTuple**(int ctrindex, vector<Value> &tuple, Cost cost) = 0

virtual void **postNaryConstraintTuple**(int ctrindex, Value *tuple, int arity, Cost cost) = 0

virtual void **postNaryConstraintEnd**(int ctrindex) = 0

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

virtual int **postSupxyc**(int xIndex, int yIndex, Value cst, Value deltamax = *MAX_VAL* - *MIN_VAL*) = 0

virtual int **postDisjunction**(int xIndex, int yIndex, Value cstx, Value csty, Cost penalty) = 0

virtual int **postSpecialDisjunction**(int xIndex, int yIndex, Value cstx, Value csty, Value xinfty, Value yinfty, Cost costx, Cost costy) = 0

virtual int **postCliqueConstraint**(vector<int> scope, const string &arguments) = 0

virtual int **postCliqueConstraint**(int *scopeIndex, int arity, istream &file) = 0

virtual int **postKnapsackConstraint**(vector<int> scope, const string &arguments, bool isclique = false, int kp = 0, bool conflict = false) = 0

virtual int **postKnapsackConstraint**(int *scopeIndex, int arity, istream &file, bool isclique = false, int kp = 0, bool conflict = false) = 0

virtual int **postWeightedCSPConstraint**(vector<int> scope, *WeightedCSP* *problem, *WeightedCSP* *negproblem, Cost lb = MIN_COST, Cost ub = MAX_COST, bool duplicateHard = false, bool strongDuality = false) = 0

create a hard constraint such that the input cost function network (problem) must have its optimum cost in [lb,ub[interval.

> **Warning:** The input scope must contain all variables in problem in the same order.

> **Warning:** if duplicateHard is true it assumes any forbidden tuple in the original input problem is also forbidden by another constraint in the main model (you must duplicate any hard constraints in your input model into the main model).

> **Warning:** if strongDuality is true then it assumes the propagation is complete when all channeling variables in the scope are assigned and the semantic of the constraint enforces that the optimum on the remaining variables is between lb and ub.

virtual int **postGlobalConstraint**(int *scopeIndex, int arity, const string &gcname, istream &file, int *constrcounter = NULL, bool mult = true) = 0

virtual void **postGlobalFunction**(vector<int> scope, const string &gcname, const string &arguments) = 0

generic function to post any global cost function

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

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

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 void **postWVarAmong**(int *scopeIndex, int arity, const string &semantics, Cost baseCost, Value *values, int nbValues, int varIndex) = 0

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 **postWRegular**(int *scopeIndex, int arity, 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

virtual void **postWRegular**(int *scopeIndex, int arity, int nbStates, vector<pair<int, Cost>> initial_States, vector<pair<int, Cost>> accepting_States, int **Wtransitions, vector<Cost> transitionsCosts) = 0

virtual int **postWAllDiff**(vector<int> &scope, 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 post a soft alldifferent cost function

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

virtual void **postWAllDiff**(int *scopeIndex, int arity, string semantics, Cost baseCost) = 0

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

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 **postWSame**(int *scopeIndex, int arity, string semantics, Cost baseCost) = 0

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 **isKnapsack**() = 0

true if there are knapsack constraints defined in the problem

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 vector<pair<Double, vector<Value>>> **getSolutions**() const = 0

returns all solutions found

virtual void **initSolutionCost**() = 0

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 Cost **decimalToCost**(const string &decimalToken, const unsigned int lineNumber) const = 0

virtual Cost **DoubletoCost**(const Double &c) const = 0

virtual Double **Cost2ADCost**(const Cost &c) const = 0

converts an integer cost from a lower or upper bound of the whole problem into a real value

virtual Double **Cost2RDCost**(const Cost &c) const = 0

converts an integer cost from a local cost function into a real value

virtual Cost **Prob2Cost**(TProb p) const = 0

virtual TProb **Cost2Prob**(Cost c) const = 0

virtual TLogProb **Cost2LogProb**(Cost c) const = 0

virtual Cost **LogProb2Cost**(TLogProb p) const = 0

virtual Cost **LogSumExp**(Cost c1, Cost c2) const = 0

virtual TLogProb **LogSumExp**(TLogProb logc1, Cost c2) const = 0

virtual TLogProb **LogSumExp**(TLogProb logc1, TLogProb logc2) const = 0

virtual void **setLb**(Cost newLb) = 0

virtual void **setUb**(Cost newUb) = 0

virtual void **restoreSolution**(Cluster *c = NULL) = 0

virtual void **buildTreeDecomposition**() = 0

virtual TreeDecomposition ***getTreeDec**() = 0

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

virtual void **initDivVariables**() = 0
initializes diversity variables with all decision variables in the problem

virtual void **iniSingleton**() = 0

virtual void **updateSingleton**() = 0

virtual void **removeSingleton**() = 0

virtual void **printVACStat**() = 0

Public Static Functions

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

CHAPTER

FOUR

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

inline virtual **~WeightedCSPSolver**()

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 void **beginSolve**(Cost ub) = 0

virtual Cost **preprocessing**(Cost ub) = 0

virtual void **recursiveSolve**(Cost lb = MIN_COST) = 0

virtual void **recursiveSolveLDS**(int discrepancy) = 0

virtual pair<Cost, Cost> **hybridSolve**() = 0

virtual void **endSolve**(bool isSolution, Cost cost, bool isComplete) = 0

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 Cost **pils**(string cmd, vector<Value> &solution) = 0

solves the current problem using PILS local search @ Francois Beuvin, David Simoncini, Sebastien Verel

> **Warning:** cannot solve problems with non-binary cost functions

Parameters

cmd – command line argument for PILS local search solver (cmd format: nbruns perturb_mode perturb_strength flatMaxIter nbEvalHC nbEvalMax strengthMin strengthMax incrFactor decrFactor)

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

virtual Double **getDDualBound**() const = 0

after (partially) solving the problem (possibly interrupted before the search is complete), return a global problem dual bound as a Double representing a decimal cost (lower resp. upper bound for minimization resp. maximization)

virtual set<int> **getUnassignedVars**() const = 0

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

Public Static Functions

static *WeightedCSPSolver* ***makeWeightedCSPSolver**(Cost initUpperBound, *WeightedCSP* *wcsp = NULL)

WeightedCSP Solver factory.

CHAPTER

FIVE

MULTICFN CLASS

class **MultiCFN**

store a combination of several cost function network

Public Types

typedef std::map<std::string, std::string> **Solution**

Public Functions

MultiCFN()

default constructor

MultiCFN(std::vector<WCSP*> &wcsps, std::vector<Double> &weights)

constructor: build a multicfn combining wcsps given as input

Parameters

- **wcsps** – a vector of wcsp objects to combine with weighted sums
- **weights** – a list of weights for each wcsp

void **push_back**(WCSP *wcsp, Double weight = 1.0)

add a wcsp to the network, create the variables if they do not exist, the wcsp is stored internally, the original wcsp will not be referenced

Parameters

- **wcsp** – the wcsp to add
- **weight** – the weight of the wcsp in the objective function (sum of the cost functions)

void **setWeight**(unsigned int wcsp_index, Double weight)

set the weight of the cost functions of one of the network

Parameters

- **wcsp_index** – the index of the network to modify
- **weight** – the new weight

Double **getWeight**(unsigned int wcsp_index)

get the wieght of a network

Parameters
wcsp_index – the index of the network (by adding order)

Returns
the weight assigned to the network

unsigned int **nbNetworks**()

number of networks loaded in the combiner

Returns
the number of network

unsigned int **nbVariables**()

number of variables in the problem

Returns
the number of variables

std::string **getNetworkName**(unsigned int index)

get the name of one of the network added to the multiwcsp

Parameters
index – the index of the network

Returns
the name associated to hte networks

unsigned int **getDecimalPoint**()

return the precision used in the combined wcsp (max of the decimalPoint of the wcsp given as input)

void **print**(std::ostream &os)

print the cfn

os the stream to print to

WeightedCSP ***makeWeightedCSP**()

make a wcsp from the convex combination of all the wcsps

void **makeWeightedCSP**(*WeightedCSP* *wcsp)

fill a wcsp with the convex combination of all the wcsps already added

Parameters
wcsp – the weighted csp to be filled

unsigned int **tupleToIndex**(std::vector<mcriteria::Var*> variables, std::vector<unsigned int> tuple)

convert a tuple to a cost index, rightmost value indexed first

Parameters

- **variables** – the list of variables from the tuple
- **tuple** – the tuple: value indexes for each variable

Returns
the index corresponding to the tuple

Solution **getSolution**()

get the solution of the created wcsp after being solved

Returns
the solution as a dictionary of variable names/value names

Pre
the wcsp must have been solved and not been deleted

std::vector<Double> **getSolutionValues**()

get the objective values of the different cost function networks from the created wcsp after being solved

Returns
the objective values as a dictionary of variable names/value names

Pre
the wcsp must have been solved and not been deleted

std::vector<Double> **computeSolutionValues**(*Solution* &solution)

compute the values of an existing solution

Parameters
solution – the solution given

Returns
the costs of the solution

Solution **convertToSolution**(std::vector<Value> &solution)

convert a solution returned by *ToulBar2* to a dictionary with variable names and values as labels

Parameters
solution – the solution given bu *ToulBar2*

Returns
the solution as a dictionary

Public Members

std::vector<mcriteria::Var> **var**

std::map<std::string, int> **var_index**

std::vector<mcriteria::CostFunction> **cost_function**

std::map<std::string, unsigned int> **cost_function_index**

Public Static Attributes

static constexpr Double **epsilon** = 1e-6

static constexpr Double **accuracy** = 1e-3

CHAPTER

SIX

TOULBAR2 CLASS

class **ToulBar2**

It contains all toulbar2 global variables encapsulated as static class members of this class.

Each variable may correspond to some command-line option of toulbar2 executable.

Public Static Attributes

static string **version** = Toulbar_VERSION

toulbar2 version number

static int **verbose**

verbosity level (-1:no output, 0: new solutions found, 1: choice points, 2: current domains, 3: basic EPTs, 4: active cost functions, 5: detailed cost functions, 6: more EPTs, 7: detailed EPTs) (command line option -v)

static bool **FullEAC**

VAC-integrality/Full-EAC variable ordering heuristic (command line option -vacint and optionally -A)

static bool **VACthreshold**

automatic threshold cost value selection for VAC during search (command line option -vacthr)

static int **nbTimesIsVAC**

static int **nbTimesIsVACitThresholdMoreThanOne**

static bool **RASPS**

static int **useRASPS**

VAC-based upper bound probing heuristic (0: no rasps, 1: rasps using DFS, >1: using LDS with bounded discrepancy + 1) (command line option -raspslds or -rasps)

static bool **RASPSreset**

reset weighted degree variable ordering heuristic after doing upper bound probing (command line option -raspsini)

static int **RASPSangle**

automatic threshold cost value selection for probing heuristic (command line option -raspsdeg)

static Long **RASPSnbBacktracks**

number of backtracks of VAC-based upper bound probing heuristic (command line option -rasps)

static int **RASPSnbStrictACVariables**

static Cost **RASPSlastitThreshold**

static bool **RASPSsaveitThresholds**

static vector<pair<Cost, Double>> **RASPSitThresholds**

static int **debug**

debug mode(0: no debug, 1: current search depth and statics on nogoods for BTD, 2: idem plus some information on heuristics, 3: idem plus save problem at each node if verbose >= 1) (command line option -Z)

static string **externalUB**

initial upper bound in CFN format

static int **showSolutions**

shows each solution found (0: nothing, 1: value indexes, 2: value names, 3: variable&value names) (command line option -s)

static bool **showHidden**

shows hidden variables for each solution found (command line option -s with a negative value)

static int **writeSolution**

writes each solution found (0: nothing, 1: value indexes, 2: value names, 3: variable&value names) (command line option -w)

static FILE ***solutionFile**

static long **solutionFileRewindPos**

static Long **allSolutions**

finds at most a given number of solutions with a cost strictly lower than the initial upper bound and stops (or counts the number of zero-cost satisfiable solutions in conjunction with BTD) (command line option -a)

static int **dumpWCSP**

saves the problem in wcsp (0: do not save, 1: original or 2: after preprocessing) or cfn (3: original or 4: after preprocessing) format (command line option -z)

static bool **dumpOriginalAfterPreprocessing**

saves the problem with initial domains after preprocessing (used in conjunction with dumpWCSP)

static bool **approximateCountingBTD**

approximate zero-cost satisfiable solution counting using BTD (command line options -D and -a and -B=1)

static bool **binaryBranching**

tree search using binary branching instead of n-ary branching for enumerated domains (command line option -b)

static int **dichotomicBranching**

tree search using dichotomic branching if current domain size is strictly greater than *ToulBar2::dichotomicBranchingSize* (0: no dichotomic branching, 1: splitting in the middle of domain range, 2: splitting in the middle of sorted unary costs) (command line option -d)

static unsigned int **dichotomicBranchingSize**

dichotomic branching threshold (related to command line option -d)

static bool **sortDomains**

sorts domains in preprocessing based on increasing unary costs (command line option -sortd)

> **Warning:** Works only for binary WCSPs.

static map<int, ValueCost*> **sortedDomains**

static bool **solutionBasedPhaseSaving**

solution-based phase saving value heuristic (command line option -solr)

static Double **bisupport**

value heuristic in bi-objective optimization when the second objective is encapsulated by a bounding constraint (command line option -bisupport)

static int **elimDegree**

boosting search with variable elimination of small degree (0: no variable elimination, 1: linked to at most one binary cost function, 2: linked to at most two binary cost functions, 3: linked to at most one ternary cost function and two scope-included cost functions) (command line option -e)

static int **elimDegree_preprocessing**

in preprocessing, generic variable elimination of degree less than or equal to a given value (0: no variable elimination) (command line option -p)

static int **elimDegree_**

static int **elimDegree_preprocessing_**

static int **elimSpaceMaxMB**

maximum space size for generic variable elimination (in MegaByte) (related to command line option -p)

static int **minsumDiffusion**

in preprocessing, applies Min Sum Diffusion algorithm a given number of iterations (command line option -M)

static int **preprocessTernaryRPC**

in preprocessing, simulates restricted path consistency by adding ternary cost functions on most-promising triangles of binary cost functions (maximum space size in MegaByte) (command line option -t)

static int **pwc**

pairwise consistency by dual encoding into a binary WCSP

static int **hve**

hidden variable encoding into a binary WCSP

static bool **pwcMinimalDualGraph**

minimizes dual intersection graph by removing redundant edges

static int **preprocessFunctional**

in preprocessing, applies variable elimination of 0: no variable, 1: functional, or 2: bijective variables (command line option -f)

static bool **costfuncSeparate**

in preprocessing, applies pairwise decomposition of non-binary cost functions (command line option -dec)

static int **preprocessNary**

in preprocessing, projects n-ary cost functions on all their scope-included binary cost functions if n is lower than a given value (0: no projection) (command line option -n)

static bool **QueueComplexity**

ensures optimal worst-case time complexity of DAC and EAC (command line option -o)

static bool **Static_variable_ordering**

tree search using a static variable ordering heuristic (same order as DAC) (command line option -svo)

static bool **lastConflict**

tree search using binary branching with last conflict backjumping variable ordering heuristic (command line options -c and -b)

static int **weightedDegree**

weighted degree variable ordering heuristic if the number of cost functions is less than a given value (command line option -q)

static int **weightedTightness**

in preprocessing, initializes weighted degrees associated to cost functions by their 1: average or 2: median costs (command line options -m and -q)

static int **constrOrdering**

in preprocessing, sorts constraints based on 0: do not sort, 1: lexicographic ordering, 2: decreasing DAC ordering, 3: decreasing constraint tightness, 4: DAC then tightness, 5: tightness then DAC, 6: random order, or the opposite order if using a negative value (command line option -sortc)

static bool **MSTDAC**

maximum spanning tree DAC ordering (command line option -mst)

static int **DEE**

soft neighborhood substitutability, a.k.a., dead-end elimination (0: no elimination, 1: restricted form during search, 2: full in preprocessing and restricted during search, 3: full always, 4: full in preprocessing) (command line option -dee)

static int **DEE_**

static int **nbDecisionVars**

tree search by branching only on the first variables having a lexicographic order position below a given value, assuming the remaining variables are completely assigned by this first group of variables (0: branch on all variables) (command line option -var)

static int **lds**

iterative limited discrepancy search (0: no LDS), use a negative value to stop the search after the given absolute number of discrepancies has been explored (command line option -l)

static bool **limited**

static Long **restart**

randomly breaks ties in variable ordering heuristics and Luby restarts until a given number of search nodes (command line option -L)

static Long **backtrackLimit**

limit on the number of backtracks (command line option -bt)

static externalevent **setvalue**

static externalevent **setmin**

static externalevent **setmax**

static externalevent **removevalue**

static externalcostevent **setminobj**

static externalsolution **newsolution**

static Pedigree ***pedigree**

static Haplotype ***haplotype**

static string **map_file**

static bool **cfn**

static bool **gz**

static bool **bz2**

static bool **xz**

static bool **bayesian**

static int **uai**

static int **resolution**

defines the number of digits that should be representable in UAI/OPB/QPBO formats (command line option -precision)

static bool **resolution_Update**

static TProb **errorg**

static TLogProb **NormFactor**

static int **foundersprob_class**

Allele frequencies of founders

- 0: equal frequencies
- 1: probs depending on the frequencies found in the problem
- otherwise: read probability distribution from command line

static vector<TProb> **allelefreqdistrib**

static bool **consecutiveAllele**

static bool **generation**

static int **pedigreeCorrectionMode**

static int **pedigreePenalty**

static int **vac**

enforces VAC at each search node having a search depth less than the absolute value of a given value (0: no VAC, 1: VAC in preprocessing, >1: VAC during search up to a given search depth), if given a negative value then VAC is not performed inside depth-first search of hybrid best-first search method (command line option -A and possibly -hbfs)

static string **costThresholdS**

threshold cost value for VAC in CFN format (command line option -T)

static string **costThresholdPreS**

in preprocessing, threshold cost value for VAC in CFN format (command line option -P)

static Cost **costThreshold**

threshold cost value for VAC (command line option -T)

static Cost **costThresholdPre**

in preprocessing, threshold cost value for VAC (command line option -P)

static Double **trwsAccuracy**

in preprocessing , enforces TRW-S until a given accuracy is reached (command line option -trws)

static bool **trwsOrder**

replaces DAC order by Kolmogorov's TRW-S order (command line option —trws-order)

static unsigned int **trwsNIter**

enforces at most n iterations of TRW-S (command line option —trws-n-iters)

static unsigned int **trwsNIterNoChange**

stops TRW-S when n iterations did not change the lower bound (command line option —trws-n-iters-no-change)

static unsigned int **trwsNIterComputeUb**

computes an upper bound every n steps in TRW-S (command line option —trws-n-iters-compute-ub)

static Double **costMultiplier**

multiplies all costs internally by this number when loading a problem in WCSP format (command line option -C)

static unsigned int **decimalPoint**

static string **deltaUbS**

stops search if the absolute optimality gap reduces below a given value in CFN format (command line option -agap)

static Cost **deltaUb**

static Cost **deltaUbAbsolute**

stops search if the absolute optimality gap reduces below a given value (command line option -agap)

static Double **deltaUbRelativeGap**

stops search if the relative optimality gap reduces below a given value (command line option -rgap)

static bool **singletonConsistency**

in preprocessing, performs singleton soft local consistency (command line option -S)

static bool **vacValueHeuristic**

VAC-based value ordering heuristic (command line options -V and -A)

static BEP ***bep**

static LcLevelType **LcLevel**

soft local consistency level (0: NC, 1: AC, 2: DAC, 3: FDAC, 4: EDAC) (command line option -k)

static int **maxEACIter**

maximum number of iterations in EDAC before switching to FDAC

static bool **wcnf**

static bool **qpbo**

static Double **qpboQuadraticCoefMultiplier**

defines coefficient multiplier for quadratic terms in QPBO format (command line option -qpmult)

static bool **opb**

static bool **addAMOConstraints**

automatically detects and adds at-most-one constraints to existing knapsack constraints

static bool **addAMOConstraints_**

automatically detects and adds at-most-one constraints to existing knapsack constraints

static int **knapsackDP**

solves exactly knapsack constraints using dynamic programming (at every search node or less often)

static unsigned int **divNbSol**

upper bound on the number of diverse solutions (0: no diverse solution) (keep it small as it controls model size)

static unsigned int **divBound**

minimum Hamming distance between diverse solutions (command line options -div and -a)

static unsigned int **divWidth**

adds a global MDD constraint with a given maximum relaxed width for finding diverse solutions (command line option -mdd)

static unsigned int **divMethod**

diversity encoding method (0: Dual, 1: Hidden, 2: Ternary, 3: Knapsack) (command line option -divm)

static unsigned int **divRelax**

MDD relaxation heuristic (0: random, 1: high diversity, 2: small diversity, 3: high unary costs) (command line option -mddh)

static char ***varOrder**

variable elimination order for DAC, BTD, and VNS methods (0: lexicographic ordering, -1: maximum cardinality search ordering, -2: minimum degree ordering, -3: minimum fill-in ordering, -4: maximum spanning tree ordering, -5: reverse Cuthill-Mckee ordering, -6: approximate minimum degree ordering, -7: same as 0, 8: lexicographic ordering using variable names, string: variable ordering filename) (command line option -O)

static int **btdMode**

tree search exploiting tree/path decomposition (0: no tree decomposition, 1: BTD with tree decomposition, 2: RDS-BTD with tree decomposition, 3: RDS-BTD with path decomposition) (command line option -B)

static int **btdSubTree**

in RDS-BTD, cluster index for solving only this particular rooted cluster subtree (command line option -I)

static int **btdRootCluster**

chooses the root cluster index (command line option -R)

static int **rootHeuristic**

root cluster heuristic (0: maximum size, 1: maximum ratio of size by height-size, 2: minimum ratio of size by height-size, 3: minimum height) (command line option -root)

static bool **reduceHeight**

minimize cluster tree height when searching for the root cluster (command line option -minheight)

static bool **maxsateval**

static bool **xmlflag**

static bool **xmlcop**

static TLogProb **markov_log**

static string **evidence_file**

static FILE ***solution_uai_file**

static string **solution_uai_filename**

static string **problemsaved_filename**

static bool **isZ**

computes logarithm of probability of evidence (a.k.a. log-partition function) in UAI format (command line option -logz)

static TLogProb **logZ**

static TLogProb **logU**

static TLogProb **logepsilon**

approximation factor for computing the log-partition function (command line option -epsilon)

static bool **uaieval**

static string **stdin_format**

file format used when reading a problem from a Unix pipe ("cfn", "wcsp", "uai", "LG", "cnf", "wcnf", "qpbo", "opb") (command line option —stdin)

static double **startCpuTime**

static double **startRealTime**

static double **startRealTimeAfterPreProcessing**

static int **splitClusterMaxSize**

splits large clusters into a chain of smaller embedded clusters with a number of proper variables less than a given value (command line option -j)

static double **boostingBTD**

in BTD, merges recursively leaf clusters with their fathers if separator size smaller than *ToulBar2::elimDegree*, else in VNS, merges clusters if the ratio of number of separator variables by number of cluster variables is above a given threshold (command line option -E and possibly -e)

static int **maxSeparatorSize**

merges recursively clusters with their fathers if separator size greater than a given threshold (command line option -r)

static int **minProperVarSize**

merges recursively clusters with their fathers if the number of proper variables is less than a given threshold (command line option -X)

static bool **heuristicFreedom**

merges clusters automatically to give more freedom to variable ordering heuristics in BTD methods (command line option -F)

static int **heuristicFreedomLimit**

stops merging a cluster subtree during BTD search if we tried repeatedly to solve this cluster for the same separator assignment more than a given number of times (-1: no merging) (command line option -F)

static bool **Berge_Dec**

static bool **learning**

static externalfunc **timeOut**

static std::atomic<bool> **interrupted**

static int **seed**

initial random seed value, or use current time if a negative value is given (command line option -seed)

static Double **sigma**

initial random noise standard deviation to be added to energy values when reading UAI format files (command line option -sigma)

static string **incop_cmd**

in preprocessing, executes INCOP local search method to produce a better initial upper bound (default parameter string value "0 1 3 idwa 100000 cv v 0 200 1 0 0", see INCOP user manual http://imagine.enpc.fr/~neveub/incop/incop1.1/usermanual.ps) (command line option -i)

static string **pils_cmd**

in preprocessing, executes PILS local search method to produce a better initial upper bound (default parameter string value "3 0 0.333 150 150 1500 0.1 0.5 0.1 0.1", see PILS article https://doi.org/10.1002/prot.26174) (command line option -pils)

static SearchMethod **searchMethod**

chooses between tree search and variable neighborhood search methods (0: tree search, 1: sequential unified VNS, 2: sequential unified decomposition guided VNS, 3: synchronous parallel UDGVNS, 4: asynchronous parallel UDGVNS, 5: tree decomposition heuristic) (command line option -vns)

static string **clusterFile**

cluster tree decomposition filename in COV or DEC format (with or without running intersection property)

static ofstream **vnsOutput**

static VNSSolutionInitMethod **vnsInitSol**

initial solution for VNS-like methods (-1: random, -2: minimum domain values, -3: maximum domain values, -4: first solution found by DFS, >=0: or by LDS with at most n discrepancies (command line option -vnsini)

static int **vnsLDSmin**

minimum discrepancy value for VNS-like methods (command line option -ldsmin)

static int **vnsLDSmax**

maximum discrepancy value for VNS-like methods (command line option -ldsmax)

static VNSInc **vnsLDSinc**

discrepancy increment strategy for VNS-like methods (1: Increment by 1, 2: Multiply by 2, 3: Luby operator) (command line option -ldsinc)

static int **vnsKmin**

minimum neighborhood size for VNS-like methods (command line option -kmin)

static int **vnsKmax**

maximum neighborhood size for VNS-like methods (command line option -kmax)

static VNSInc **vnsKinc**

neighborhood size increment strategy for VNS-like methods (1: Increment by 1, 2: Multiply by 2, 3: Luby operator, 4: Increment by 1 until maximum cluster size then considers all variables) (command line option -kinc)

static int **vnsLDScur**

static int **vnsKcur**

static VNSVariableHeuristic **vnsNeighborVarHeur**

neighborhood heuristic method (0: random variables, 1: variables in conflict, 2: connected variables in conflict, 3: random cluster, 4: variables in conflict with maximum degree, 5: sorted cluster, 6: sorted cluster separator, 7: similar to 6, 8: randomized root cluster, 9: variables in partial conflict)

static bool **vnsNeighborChange**

static bool **vnsNeighborSizeSync**

static bool **vnsParallelLimit**

static bool **vnsParallelSync**

static string **vnsOptimumS**

stops VNS if a solution is found with a given cost (or better) in CFN format (command line option -best)

static Cost **vnsOptimum**

stops VNS if a solution is found with a given cost (or better) (command line option -best)

static bool **parallel**

parallel mode for tree search and VNS (see mpirun toulbar2)

static Long **hbfs**

performs hybrid best-first search with a given limit in the number of backtracks for depth-first search before visiting another open node (0: always DFS, 1: HBFS) (related to command line option -hbfs)

static Long **hbfsGlobalLimit**

restarts BTD-HBFS from the root cluster after a given number of backtracks (command line option -hbfs)

static Long **hbfsAlpha**

minimum recomputation node redundancy percentage threshold value (command line option -hbfsmin)

static Long **hbfsBeta**

maximum recomputation node redundancy percentage threshold value (command line option -hbfsmax)

static ptrdiff_t **hbfsCPLimit**

maximum number of stored choice points before switching to normal DFS (warning! should always be cast to std::size_t when used, so that -1 is equivalent to +infinity)

static ptrdiff_t **hbfsOpenNodeLimit**

maximum number of stored open nodes before switching to normal DFS (command line option -open) (warning! should always be cast to std::size_t when used, so that -1 is equivalent to +infinity)

static Long **eps**

performs HBFS until a given number of open nodes are collected and exits (command line option -eps)

static string **epsFilename**

a given filename to print remaining valid (lower bound less than current upper bound) open nodes as partial assignments before exits (command line option -eps)

static bool **verifyOpt**

compiled in debug, checks if a given (optimal) solution is never pruned by propagation when the current upper bound is greater than the cost of this solution (see Solver::read_solution, related to command line option -opt)

static Cost **verifiedOptimum**

compiled in debug, a given (optimal) solution cost (see Solver::read_solution, related to command line option -opt)

static int **bilevel**

bilevel optimization using modified BTD with (at least) four clusters (P0 -> P1, P0 -> P2, P0 -> NegP2) corresponding to the restricted leader problem (P0 and P1), the follower problem (P2), and the negative follower problem (NegP2) (command line option -bilevel)

static vector<unsigned int> **decimalPointBLP**

static vector<Double> **costMultiplierBLP**

static vector<Cost> **negCostBLP**

static vector<Cost> **initialLbBLP**

static vector<Cost> **initialUbBLP**

CHAPTER

SEVEN

MISCELLANEOUS FUNCTIONS

void **tb2init**()

initialization of *ToulBar2* global variables (needed by numberjack/toulbar2)

initialization of *ToulBar2* global variables (needed by numberjack/toulbar2)

void **tb2checkOptions**()

checks compatibility between selected options of *ToulBar2* (needed by numberjack/toulbar2)

checks compatibility between selected options of *ToulBar2* (needed by numberjack/toulbar2)

INDEX

W