# toulbar2

Generated by Doxygen 1.8.13

# Contents

# 1 Main Page

| | |
|---|---|
| **Cost Function Network Solver** | toulbar2 |
| **Copyright** | toulbar2 team |
| **Source** | `https://github.com/toulbar2/toulbar2` |

See the README for more details.

toulbar2 can be used as a stand-alone solver reading various problem file formats (wcsp, uai, wcnf, qpbo) or as a C++ library.
This document describes the wcsp native file format and the toulbar2 C++ library API.

**Note**

Use cmake flags LIBTB2=ON and TOULBAR2_ONLY=OFF to get the toulbar2 C++ library libtb2.so and toulbar2test executable example.

**See also**

./src/toulbar2test.cpp

# 2 toulbar2

### Exact optimization for cost function networks and additive graphical models

master: cpd:

### What is toulbar2?

toulbar2 is an open-source black-box C++ optimizer for cost function networks and discrete additive graphical models. It can read a variety of formats. The optimized criteria and feasibility should be provided factorized in local cost functions on discrete variables. Constraints are represented as functions that produce costs that exceed a user-provided primal bound. toulbar2 looks for a non-forbidden assignment of all variables that optimizes the sum of all functions (a decision NP-complete problem).

toulbar2 won several competitions on deterministic and probabilistic graphical models:

- Max-CSP 2008 Competition `CPAI08` (winner on 2-ARY-EXT and N-ARY-EXT)
- Probabilistic Inference Evaluation `UAI 2008` (winner on several MPE tasks, inra entries)
- 2010 UAI APPROXIMATE INFERENCE CHALLENGE `UAI 2010` (winner on 1200-second MPE task)
- The Probabilistic Inference Challenge `PIC 2011` (second place by ficolofo on 1-hour MAP task)
- UAI 2014 Inference Competition `UAI 2014` (winner on all MAP task categories, see Proteus, Robin, and IncTb entries)

### Installation from binaries

You can install toulbar2 directly using the package manager in Debian and Debian derived Linux distributions (Ubuntu, Mint,...). For the most recent version, compile from source.

### Download

Download the latest release from GitHub (`https://github.com/toulbar2/toulbar2`) or similarly use tag versions, e.g.:

```
git clone --branch 1.0.0 https://github.com/toulbar2/toulbar2.git
```

### Installation from sources

Compilation requires git, cmake and a C++-11 capable compiler.

Required library:

- libgmp-dev

Recommended libraries (default use):

- libboost-graph-dev
- libboot-iostream-dev
- zlib

Optional libraries:

- libxml2-dev
- libopenmpi-dev

GNU C++ Symbols to be defined if using Linux Eclipse/CDT IDE (no value needed):

- BOOST
- LINUX
- LONGDOUBLE_PROB
- LONGLONG_COST
- NARYCHAR
- OPENMPI
- WCSPFORMATONLY
- WIDE_STRING Also C++11 should be set as the language standard.

Commands for compiling toulbar2 on Linux in directory toulbar2/src without cmake:

```
bash
cd src
echo '#define Toulbar_VERSION "1.0"' > ToulbarVersion.hpp
g++ -o toulbar2 -I. tb2*.cpp applis/*.cpp core/*.cpp globals/*.cpp incop/*.cpp search/*.cpp utils/*.cpp vns/*.
 -DBOOST -DLINUX -DLONGDOUBLE_PROB -DLONGLONG_COST -DNARYCHAR -DWCSPFORMATONLY -DWIDE_STRING -lgmp -static
```

Replace LONGLONG_COST by INT_COST to reduce memory usage by two and reduced cost range (costs must be smaller than 10^8).

Use OPENMPI flag and MPI compiler for a parallel version of toulbar2:

```
bash
cd src
echo '#define Toulbar_VERSION "1.0"' > ToulbarVersion.hpp
mpicxx -o toulbar2 -I. tb2*.cpp applis/*.cpp core/*.cpp globals/*.cpp incop/*.cpp search/*.cpp utils/*.cpp vns
 -DBOOST -DLINUX -DLONGDOUBLE_PROB -DLONGLONG_COST -DNARYCHAR -DOPENMPI -DWCSPFORMATONLY -DWIDE_STRING -lgmp
```

### Authors

toulbar2 was originally developped by Toulouse (INRA MIAT) and Barcelona (UPC, IIIA-CSIC) teams, hence the name of the solver.

Additional contributions by:

- Caen University, France (GREYC) and University of Oran, Algeria for (parallel) variable neighborhood search methods
- The Chinese University of Hong Kong and Caen University, France (GREYC) for global cost functions
- Marseille University, France (LSIS) for tree decomposition heuristics
- Ecole des Ponts ParisTech, France (CERMICS/LIGM) for `INCOP` local search solver
- University College Cork, Ireland (Insight) for a Python interface in `NumberJack` and a portfolio dedicated to UAI graphical models `Proteus`
- Artois University, France (CRIL) for an XCSP 2.1 format reader of CSP and WCSP instances

### Citing

Please use one of the following references for citing toulbar2:

- Multi-Language Evaluation of Exact Solvers in Graphical Model Discrete Optimization Barry Hurley, Barry O'Sullivan, David Allouche, George Katsirelos, Thomas Schiex, Matthias Zytnicki, Simon de Givry Constraints, 21(3):413-434, 2016
- Tractability-preserving Transformations of Global Cost Functions David Allouche, Christian Bessiere, Patrice Boizumault, Simon de Givry, Patricia Gutierrez, Jimmy HM. Lee, Ka Lun Leung, Samir Loudni, Jean-Philippe Métivier, Thomas Schiex, Yi Wu Artificial Intelligence, 238:166-189, 2016
- Soft arc consistency revisited Martin Cooper, Simon de Givry, Marti Sanchez, Thomas Schiex, Matthias Zytnicki, and Thomas Werner Artificial Intelligence, 174(7-8):449-478, 2010

### What are the algorithms inside toulbar2?

- Soft arc consistency (AC): Arc consistency for Soft Constraints T. Schiex Proc. of CP'2000. Singapour, September 2000.
- More soft arc consistencies (NC, DAC, FDAC): In the quest of the best form of local consistency for Weighted CSP J. Larrosa & T. Schiex In Proc. of IJCAI-03. Acapulco, Mexico, 2003
- Soft existential arc consistency (EDAC) Existential arc consistency: Getting closer to full arc consistency in weighted csps S. de Givry, M. Zytnicki, F. Heras, and J. Larrosa In Proc. of IJCAI-05, Edinburgh, Scotland, 2005
- Depth-first Branch and Bound exploiting a tree decomposition (BTD) Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP S. de Givry, T. Schiex, and G. Verfaillie In Proc. of AAAI-06, Boston, MA, 2006
- Virtual arc consistency (VAC) Virtual arc consistency for weighted csp M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki In Proc. of AAAI-08, Chicago, IL, 2008
- Soft generalized arc consistencies (GAC, FDGAC) Towards Efficient Consistency Enforcement for Global Constraints in Weighted Constraint Satisfaction J. H. M. Lee and K. L. Leung In Proc. of IJCAI-09, Los Angeles, USA, 2010

- Russian doll search exploiting a tree decomposition (RDS-BTD) Russian doll search with tree decomposition M Sanchez, D Allouche, S de Givry, and T Schiex In Proc. of IJCAI'09, Pasadena (CA), USA, 2009
- Soft bounds arc consistency (BAC) Bounds Arc Consistency for Weighted CSPs M. Zytnicki, C. Gaspin, S. de Givry, and T. Schiex Journal of Artificial Intelligence Research, 35:593-621, 2009
- Counting solutions in satisfaction (#BTD, Approx_::BTD) Exploiting problem structure for solution counting A. Favier, S. de Givry, and P. Jégou In Proc. of CP-09, Lisbon, Portugal, 2009
- Soft existential generalized arc consistency (EDGAC) A Stronger Consistency for Soft Global Constraints in Weighted Constraint Satisfaction J. H. M. Lee and K. L. Leung In Proc. of AAAI-10, Boston, MA, 2010
- Preprocessing techniques (combines variable elimination and cost function decomposition) Pairwise decomposition for combinatorial optimization in graphical models A Favier, S de Givry, A Legarra, and T Schiex In Proc. of IJCAI-11, Barcelona, Spain, 2011
- Decomposable global cost functions (wregular, wamong, wsum) Decomposing global cost functions D Allouche, C Bessiere, P Boizumault, S de Givry, P Gutierrez, S Loudni, JP Métivier, and T Schiex In Proc. of AAAI-12, Toronto, Canada, 2012
- Pruning by dominance (DEE) Dead-End Elimination for Weighted CSP S de Givry, S Prestwich, and B O'Sullivan In Proc. of CP-13, pages 263-272, Uppsala, Sweden, 2013
- Hybrid best-first search exploiting a tree decomposition (HBFS) Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP D Allouche, S de Givry, G Katsirelos, T Schiex, and M Zytnicki In Proc. of CP-15, Cork, Ireland, 2015
- Unified parallel decomposition guided variable neighborhood search (UDGVNS/UPDGVNS) Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization A Ouali, D Allouche, S de Givry, S Loudni, Y Lebbah, F Eckhardt, and L Loukil In Proc. of UAI-17, pages 550-559, Sydney, Australia, 2017
- Clique cut global cost function (clique) Clique Cuts in Weighted Constraint Satisfaction S de Givry and G Katsirelos In Proc. of CP-17, pages 97-113, Melbourne, Australia, 2017

# 3 Module Documentation

## 3.1 Virtual Arc Consistency enforcing

The three phases of VAC are enforced in three different "Pass". Bool(P) is never built. Instead specific functions (getVACCost) booleanize the WCSP on the fly. The domain variables of Bool(P) are the original variable domains (saved and restored using trailing at each iteration) All the counter data-structures (k) are timestamped to avoid clearing them at each iteration.

**Note**

Simultaneously AC (and potentially DAC, EAC) are maintained by proper queuing.

**See also**

*Soft Arc Consistency Revisited.* Cooper et al. Artificial Intelligence. 2010.

## 3.2 Preprocessing techniques

Depending on toulbar2 options, the sequence of preprocessing techniques applied before the search is:

1. *i-bounded* variable elimination with user-defined *i* bound
2. pairwise decomposition of cost functions (binary cost functions are implicitly decomposed by soft AC and empty cost function removals)
3. MinSumDiffusion propagation (see VAC)
4. projects&substracts n-ary cost functions in extension on all the binary cost functions inside their scope ($3 <$ n $<$ max, see toulbar2 options)
5. functional variable elimination (see Variable elimination)
6. projects&substracts ternary cost functions in extension on their three binary cost functions inside their scope (before that, extends the existing binary cost functions to the ternary cost function and applies pairwise decomposition)
7. creates new ternary cost functions for all triangles (*ie* occurences of three binary cost functions *xy*, *yz*, *zx*)
8. removes empty cost functions while repeating #1 and #2 until no new cost functions can be removed

**Note**

the propagation loop is called after each preprocessing technique (see WCSP::propagate)

## 3.3 Output messages, verbosity options and debugging

Depending on verbosity level given as option "-v=level", `toulbar2` will output:

- (level=0, no verbosity) default output mode: shows version number, number of variables and cost functions read in the problem file, number of unassigned variables and cost functions after preprocessing, problem upper and lower bounds after preprocessing. Outputs current best solution cost found, ends by giving the optimum or "No solution". Last output line should always be: "end."
- (level=-1, no verbosity) restricted output mode: do not print current best solution cost found

1. (level=1) shows also search choices ("["*search_depth problem_lower_bound problem_upper_bound sum_↩ of_current_domain_sizes"] Try" variable_index operator value*) with *operator* being assignment ("=="), value removal ("!="), domain splitting ("<=" or ">=", also showing EAC value in parenthesis)
2. (level=2) shows also current domains (*variable_index list_of_current_domain_values* "/" *number_of_cost↩ _functions* (see approximate degree in Variable elimination) "/" *weighted_degree list_of_unary_costs* "s↩ :" *support_value*) before each search choice and reports problem lower bound increases, NC bucket sort data (see NC bucket sort), and basic operations on domains of variables
3. (level=3) reports also basic arc EPT operations on cost functions (see Soft arc consistency and problem reformulation)
4. (level=4) shows also current list of cost functions for each variable and reports more details on arc EPT operations (showing all changes in cost functions)
5. (level=5) reports more details on cost functions defined in extension giving their content (cost table by first increasing values in the current domain of the last variable in the scope)

For debugging purposes, another option "-Z=level" allows one to monitor the search:

1. (level 1) shows current search depth (number of search choices from the root of the search tree) and reports statistics on nogoods for BTD-like methods
2. (level 2) idem
3. (level 3) also saves current problem into a file before each search choice

**Note**

`toulbar2`, compiled in debug mode, can be more verbose and it checks a lot of assertions (pre/post conditions in the code)
`toulbar2` will output an help message giving available options if run without any parameters

## 3.4 NC bucket sort

maintains a sorted list of variables having non-zero unary costs in order to make NC propagation incremental.

- variables are sorted into buckets
- each bucket is associated to a single interval of non-zero costs (using a power-of-two scaling, first bucket interval is [1,2[, second interval is [2,4[, etc.)
- each variable is inserted into the bucket corresponding to its largest unary cost in its domain
- variables having all unary costs equal to zero do not belong to any bucket

NC propagation will revise only variables in the buckets associated to costs sufficiently large wrt current objective bounds.

## 3.5 Variable elimination

- *i-bounded* variable elimination eliminates all variables with a degree less than or equal to *i*. It can be done with arbitrary i-bound in preprocessing only and iff all their cost functions are in extension.
- *i-bounded* variable elimination with i-bound less than or equal to two can be done during the search.
- functional variable elimination eliminates all variables which have a bijective or functional binary hard constraint (*ie* ensuring a one-to-one or several-to-one value mapping) and iff all their cost functions are in extension. It can be done without limit on their degree, in preprocessing only.

**Note**

Variable elimination order used in preprocessing is either lexicographic or given by an external file ∗.order (see toulbar2 options)
2-bounded variable elimination during search is optimal in the sense that any elimination order should result in the same final graph

**Warning**

It is not possible to display/save solutions when bounded variable elimination is applied in preprocessing
toulbar2 maintains a list of current cost functions for each variable. It uses the size of these lists as an approximation of variable degrees. During the search, if variable *x* has three cost functions *xy*, *xz*, *xyz*, its true degree is two but its approximate degree is three. In toulbar2 options, it is the approximate degree which is given by the user for variable elimination during the search (thus, a value at most three). But it is the true degree which is given by the user for variable elimination in preprocessing.

### 3.6 Soft arc consistency and problem reformulation

Soft arc consistency is an incremental lower bound technique for optimization problems. Its goal is to move costs from high-order (typically arity two or three) cost functions towards the problem lower bound and unary cost functions. This is achieved by applying iteratively local equivalence-preserving problem transformations (EPTs) until some terminating conditions are met.

**Note**

*eg* an EPT can move costs between a binary cost function and a unary cost function such that the sum of the two functions remains the same for any complete assignment.

**See also**

*Arc consistency for Soft Constraints.* T. Schiex. Proc. of CP'2000. Singapour, 2000.

**Note**

Soft Arc Consistency in toulbar2 is limited to binary and ternary and some global cost functions (*eg* alldifferent, gcc, regular, same). Other n-ary cost functions are delayed for propagation until their number of unassigned variables is three or less.

**See also**

*Towards Efficient Consistency Enforcement for Global Constraints in Weighted Constraint Satisfaction.* Jimmy Ho-Man Lee, Ka Lun Leung. Proc. of IJCAI 2009, pages 559-565. Pasadena, USA, 2009.

## 3.7 Propagation loop

Propagates soft local consistencies and bounded variable elimination until all the propagation queues are empty or a contradiction occurs.
While (queues are not empty or current objective bounds have changed):

1. queue for bounded variable elimination of degree at most two (except at preprocessing)
2. BAC queue
3. EAC queue
4. DAC queue
5. AC queue
6. monolithic (flow-based and DAG-based) global cost function propagation (partly incremental)
7. NC queue
8. returns to #1 until all the previous queues are empty
9. DEE queue
10. returns to #1 until all the previous queues are empty
11. VAC propagation (not incremental)
12. returns to #1 until all the previous queues are empty (and problem is VAC if enable)
13. exploits goods in pending separators for BTD-like methods

Queues are first-in / first-out lists of variables (avoiding multiple insertions). In case of a contradiction, queues are explicitly emptied by WCSP::whenContradiction

## 3.8 Variable and value search ordering heuristics

**See also**

*Boosting Systematic Search by Weighting Constraints* . Frederic Boussemart, Fred Hemery, Christophe Lecoutre, Lakhdar Sais. Proc. of ECAI 2004, pages 146-150. Valencia, Spain, 2004.
*Last Conflict Based Reasoning* . Christophe Lecoutre, Lakhdar Sais, Sebastien Tabary, Vincent Vidal. Proc. of ECAI 2006, pages 133-137. Trentino, Italy, 2006.

## 3.9 Weighted Constraint Satisfaction Problem file format (wcsp)

It is a text format composed of a list of numerical and string terms separated by spaces. Instead of using names for making reference to variables, variable indexes are employed. The same for domain values. All indexes start at zero.

Cost functions can be defined in intention (see below) or in extension, by their list of tuples. A default cost value is defined per function in order to reduce the size of the list. Only tuples with a different cost value should be given (not mandatory). All the cost values must be positive. The arity of a cost function in extension may be equal to zero. In this case, there is no tuples and the default cost value is added to the cost of any solution. This can be used to represent a global lower bound constant of the problem.

The wcsp file format is composed of three parts: a problem header, the list of variable domain sizes, and the list of cost functions.

- Header definition for a given problem:

  ```
  <Problem name>
  <Number of variables (N)>
  <Maximum domain size>
  <Number of cost functions>
  <Initial global upper bound of the problem (UB)>
  ```

  The goal is to find an assignment of all the variables with minimum total cost, strictly lower than UB. Tuples with a cost greater than or equal to UB are forbidden (hard constraint).

- Definition of domain sizes

  ```
  <Domain size of variable with index 0>
  ...
  <Domain size of variable with index N - 1>
  ```

  **Note**

  domain values range from zero to *size-1*
  a negative domain size is interpreted as a variable with an interval domain in $[0, -size-1]$

  **Warning**

  variables with interval domains are restricted to arithmetic and disjunctive cost functions in intention (see below)

- General definition of cost functions

  - Definition of a cost function in extension

    ```
    <Arity of the cost function>
    <Index of the first variable in the scope of the cost function>
    ...
    <Index of the last variable in the scope of the cost function>
    <Default cost value>
    <Number of tuples with a cost different than the default cost>
    ```

    followed by for every tuple with a cost different than the default cost:

    ```
    <Index of the value assigned to the first variable in the scope>
    ...
    <Index of the value assigned to the last variable in the scope>
    <Cost of the tuple>
    ```

**Note**

Shared cost function: A cost function in extension can be shared by several cost functions with the same arity (and same domain sizes) but different scopes. In order to do that, the cost function to be shared must start by a negative scope size. Each shared cost function implicitly receives an occurrence number starting from 1 and incremented at each new shared definition. New cost functions in extension can reuse some previously defined shared cost functions in extension by using a negative number of tuples representing the occurrence number of the desired shared cost function. Note that default costs should be the same in the shared and new cost functions. Here is an example of 4 variables with domain size 4 and one AllDifferent hard constraint decomposed into 6 binary constraints.

- Shared CF used inside a small example in wcsp format:

```
AllDifferentDecomposedIntoBinaryConstraints 4 4 6 1
4 4 4 4
-2 0 1 0 4
0 0 1
1 1 1
2 2 1
3 3 1
2 0 2 0 -1
2 0 3 0 -1
2 1 2 0 -1
2 1 3 0 -1
2 2 3 0 -1
```

- Definition of a cost function in intension by replacing the default cost value by -1 and by giving its keyword name and its K parameters

```
<Arity of the cost function>
<Index of the first variable in the scope of the cost function>
...
<Index of the last variable in the scope of the cost function>
-1
<keyword>
<parameter1>
...
<parameterK>
```

Possible keywords of cost functions defined in intension followed by their specific parameters:

- $>=$ *cst delta* to express soft binary constraint $x \geq y + cst$ with associated cost function $max((y + cst - x \leq delta)?(y + cst - x) : UB, 0)$
- $>$ *cst delta* to express soft binary constraint $x > y + cst$ with associated cost function $max((y + cst + 1 - x \leq delta)?(y + cst + 1 - x) : UB, 0)$
- $<=$ *cst delta* to express soft binary constraint $x \leq y + cst$ with associated cost function $max((x - cst - y \leq delta)?(x - cst - y) : UB, 0)$
- $<$ *cst delta* to express soft binary constraint $x < y + cst$ with associated cost function $max((x - cst + 1 - y \leq delta)?(x - cst + 1 - y) : UB, 0)$
- = *cst delta* to express soft binary constraint $x = y + cst$ with associated cost function $(|y + cst - x| \leq delta)?|y + cst - x| : UB$
- disj *cstx csty penalty* to express soft binary disjunctive constraint $x \geq y + csty \vee y \geq x + cstx$ with associated cost function $(x \geq y + csty \vee y \geq x + cstx)?0 : penalty$
- sdisj *cstx csty xinfty yinfty costx costy* to express a special disjunctive constraint with three implicit hard constraints $x \leq xinfty$ and $y \leq yinfty$ and $x < xinfty \wedge y < yinfty \Rightarrow (x \geq y + csty \vee y \geq x + cstx)$ and an additional cost function $((x = xinfty)?costx : 0) + ((y = yinfty)?costy : 0)$
- Global cost functions using a dedicated propagator:
  - clique *1* (*nb_values* (*value*)$*$)$*$ to express a hard clique cut to restrict the number of variables taking their value into a given set of values (per variable) to at most *1* occurrence for all the variables (warning! it assumes also a clique of binary constraints already exists to forbid any two variables using both the restricted values)

- Global cost functions using a flow-based propagator:
  - salldiff var|dec|decbi *cost* to express a soft alldifferent constraint with either variable-based (*var* keyword) or decomposition-based (*dec* and *decbi* keywords) cost semantic with a given *cost* per violation (*decbi* decomposes into a binary cost function complete network)
  - sgcc var|dec|wdec *cost nb_values* (*value lower_bound upper_bound* (*shortage_weight excess_*↩ *weight*)?)∗ to express a soft global cardinality constraint with either variable-based (*var* keyword) or decomposition-based (*dec* keyword) cost semantic with a given *cost* per violation and for each value its lower and upper bound (if *wdec* then violation cost depends on each value shortage or excess weights)
  - ssame *cost list_size1 list_size2* (*variable_index*)∗ (*variable_index*)∗ to express a permutation constraint on two lists of variables of equal size (implicit variable-based cost semantic)
  - sregular var|edit *cost nb_states nb_initial_states* (*state*)∗ *nb_final_states* (*state*)∗ *nb_transitions* (*start*↩ *_state symbol_value end_state*)∗ to express a soft regular constraint with either variable-based (*var* keyword) or edit distance-based (*edit* keyword) cost semantic with a given *cost* per violation followed by the definition of a deterministic finite automaton with number of states, list of initial and final states, and list of state transitions where symbols are domain values
- Global cost functions using a dynamic programming DAG-based propagator:
  - sregulardp var *cost nb_states nb_initial_states* (*state*)∗ *nb_final_states* (*state*)∗ *nb_transitions* (*start_*↩ *state symbol_value end_state*)∗ to express a soft regular constraint with a variable-based (*var* keyword) cost semantic with a given *cost* per violation followed by the definition of a deterministic finite automaton with number of states, list of initial and final states, and list of state transitions where symbols are domain values
  - sgrammar|sgrammardp var|weight *cost nb_symbols nb_values start_symbol nb_rules* ((0 *terminal*↩ *_symbol value*)|(1 *nonterminal_in nonterminal_out_left nonterminal_out_right*)|(2 *terminal_symbol value weight*)|(3 *nonterminal_in nonterminal_out_left nonterminal_out_right weight*))∗ to express a soft/weighted grammar in Chomsky normal form
  - samong|samongdp var *cost lower_bound upper_bound nb_values* (*value*)∗ to express a soft among constraint to restrict the number of variables taking their value into a given set of values
  - salldiffdp var *cost* to express a soft alldifferent constraint with variable-based (*var* keyword) cost semantic with a given *cost* per violation (decomposes into samongdp cost functions)
  - sgccdp var *cost nb_values* (*value lower_bound upper_bound*)∗ to express a soft global cardinality constraint with variable-based (*var* keyword) cost semantic with a given *cost* per violation and for each value its lower and upper bound (decomposes into samongdp cost functions)
  - max|smaxdp *defCost nbtuples* (*variable value cost*)∗ to express a weighted max cost function to find the maximum cost over a set of unary cost functions associated to a set of variables (by default, *defCost* if unspecified)
  - MST|smstdp to express a spanning tree hard constraint where each variable is assigned to its parent variable index in order to build a spanning tree (the root being assigned to itself)
- Global cost functions using a cost function network-based propagator:
  - wregular *nb_states nb_initial_states* (*state* and cost)∗ *nb_final_states* (*state* and cost)∗ *nb_transitions* (*start_state symbol_value end_state cost*)∗ to express a weighted regular constraint with weights on initial states, final states, and transitions, followed by the definition of a deterministic finite automaton with number of states, list of initial and final states with their costs, and list of weighted state transitions where symbols are domain values
  - walldiff hard|lin|quad *cost* to express a soft alldifferent constraint as a set of wamong hard constraint (*hard* keyword) or decomposition-based (*lin* and *quad* keywords) cost semantic with a given *cost* per violation
  - wgcc hard|lin|quad *cost nb_values* (*value lower_bound upper_bound*)∗ to express a soft global cardinality constraint as either a hard constraint (*hard* keyword) or with decomposition-based (*lin* and *quad* keyword) cost semantic with a given *cost* per violation and for each value its lower and upper bound
  - wsame hard|lin|quad *cost* to express a permutation constraint on two lists of variables of equal size (implicitly concatenated in the scope) using implicit decomposition-based cost semantic

- wsamegcc hard|lin|quad *cost nb_values* (*value lower_bound upper_bound*)$*$ to express the combination of a soft global cardinality constraint and a permutation constraint
- wamong hard|lin|quad *cost nb_values* (*value*)$*$ *lower_bound upper_bound* to express a soft among constraint to restrict the number of variables taking their value into a given set of values
- wvaramong hard *cost nb_values* (*value*)$*$ to express a hard among constraint to restrict the number of variables taking their value into a given set of values to be equal to the last variable in the scope
- woverlap hard|lin|quad *cost comparator righthandside* overlaps between two sequences of variables X, Y (i.e. set the fact that Xi and Yi take the same value (not equal to zero))
- wsum hard|lin|quad *cost comparator righthandside* to express a soft sum constraint with unit coefficients to test if the sum of a set of variables matches with a given comparator and right-hand-side value
- wvarsum hard *cost comparator* to express a hard sum constraint to restrict the sum to be *comparator* to the value of the last variable in the scope

  Let us note $<>$ the comparator, K the right-hand-side value associated to the comparator, and Sum the result of the sum over the variables. For each comparator, the gap is defined according to the distance as follows:
  - if $<>$ is == : gap = abs(K - Sum)
  - if $<>$ is <= : gap = max(0,Sum - K)
  - if $<>$ is $<$ : gap = max(0,Sum - K - 1)
  - if $<>$ is != : gap = 1 if Sum != K and gap = 0 otherwise
  - if $<>$ is $>$ : gap = max(0,K - Sum + 1);
  - if $<>$ is $>=$ : gap = max(0,K - Sum);

**Warning**

The decomposition of wsum and wvarsum may use an exponential size (sum of domain sizes).
*list_size1* and *list_size2* must be equal in *ssame*.
Cost functions defined in intention cannot be shared.

**Note**

More about network-based global cost functions can be found here https://metivier.users.greyc.fr/decomposable/

Examples:

- quadratic cost function $x0 * x1$ in extension with variable domains $\{0, 1\}$ (equivalent to a soft clause $\neg x0 \vee \neg x1$):

```
2 0 1 0 1 1 1 1
```

- simple arithmetic hard constraint $x1 < x2$:

```
2 1 2 -1 < 0 0
```

- hard temporal disjunction $x1 \geq x2 + 2 \vee x2 \geq x1 + 1$:

```
2 1 2 -1 disj 1 2 UB
```

- clique cut ({x0,x1,x2,x3}) on Boolean variables such that value 1 is used at most once:

```
4 0 1 2 3 -1 clique 1 1 1 1 1 1 1 1 1
```

- soft_alldifferent({x0,x1,x2,x3}):

```
4 0 1 2 3 -1 salldiff var 1
```

- soft_gcc({x1,x2,x3,x4}) with each value *v* from 1 to 4 only appearing at least v-1 and at most v+1 times:

```
4 1 2 3 4 -1 sgcc var 1 4 1 0 2 2 1 3 3 2 4 4 3 5
```

- soft_same({x0,x1,x2,x3},{x4,x5,x6,x7}):

```
8 0 1 2 3 4 5 6 7 -1 ssame 1 4 4 0 1 2 3 4 5 6 7
```

- soft_regular({x1,x2,x3,x4}) with DFA (3∗)+(4∗):

```
4 1 2 3 4 -1 sregular var 1 2 1 0 2 0 1 3 0 3 0 0 4 1 1 4 1
```

- soft_grammar({x0,x1,x2,x3}) with hard cost (1000) producing well-formed parenthesis expressions:

```
4 0 1 2 3 -1 sgrammardp var 1000 4 2 0 6 1 0 0 0 1 0 1 2 1 0 1 3 1 2 0 3 0 1 0 0 3 1
```

- soft_among({x1,x2,x3,x4}) with hard cost (1000) if $\sum_{i=1}^{4}(x_i \in \{1,2\}) < 1$ or $\sum_{i=1}^{4}(x_i \in \{1,2\}) > 3$:

```
4 1 2 3 4 -1 samongdp var 1000 1 3 2 1 2
```

- soft max({x0,x1,x2,x3}) with cost equal to $\max_{i=0}^{3}((x_i! = i)?1000 : (4 - i))$:

```
4 0 1 2 3 -1 smaxdp 1000 4 0 0 4 1 1 3 2 2 2 3 3 1
```

- wregular({x0,x1,x2,x3}) with DFA (0(10)∗2∗):

```
4 0 1 2 3 -1 wregular 3 1 0 0 1 2 0 9 0 0 1 0 0 1 1 1 0 2 1 1 1 1 0 0 1 0 0 1 1 2 0 1 1 2 2 0 1 0 2 1 1 1 2
       1
```

- wamong ({x1,x2,x3,x4}) with hard cost (1000) if $\sum_{i=1}^{4}(x_i \in \{1,2\}) < 1$ or $\sum_{i=1}^{4}(x_i \in \{1,2\}) > 3$:

```
4 1 2 3 4 -1 wamong hard 1000 2 1 2 1 3
```

- wvaramong ({x1,x2,x3,x4}) with hard cost (1000) if $\sum_{i=1}^{3}(x_i \in \{1,2\}) \neq x_4$:

```
4 1 2 3 4 -1 wvaramong hard 1000 2 1 2
```

- woverlap({x1,x2,x3,x4}) with hard cost (1000) if $\sum_{i=1}^{2}(x_i = x_{i+2}) \geq 1$:

```
4 1 2 3 4 -1 woverlap hard 1000 < 1
```

- wsum ({x1,x2,x3,x4}) with hard cost (1000) if $\sum_{i=1}^{4}(x_i) \neq 4$:

```
4 1 2 3 4 -1 wsum hard 1000 == 4
```

- wvarsum ({x1,x2,x3,x4}) with hard cost (1000) if $\sum_{i=1}^{3}(x_i) \neq x_4$:

```
4 1 2 3 4 -1 wvarsum hard 1000 ==
```

Latin Square 4 x 4 crisp CSP example in wcsp format:

```
latin4 16 4 8 1
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 0 1 2 3 -1 salldiff var 1
4 4 5 6 7 -1 salldiff var 1
4 8 9 10 11 -1 salldiff var 1
4 12 13 14 15 -1 salldiff var 1
4 0 4 8 12 -1 salldiff var 1
4 1 5 9 13 -1 salldiff var 1
4 2 6 10 14 -1 salldiff var 1
4 3 7 11 15 -1 salldiff var 1
```

4-queens binary weighted CSP example with random unary costs in wcsp format:

```
4-WQUEENS 4 4 10 5
4 4 4 4
2 0 1 0 10
0 0 5
0 1 5
1 0 5
1 1 5
1 2 5
2 1 5
2 2 5
2 3 5
3 2 5
3 3 5
2 0 2 0 8
0 0 5
0 2 5
1 1 5
1 3 5
2 0 5
2 2 5
3 1 5
3 3 5
2 0 3 0 6
0 0 5
0 3 5
1 1 5
2 2 5
3 0 5
3 3 5
2 1 2 0 10
0 0 5
0 1 5
1 0 5
1 1 5
1 2 5
2 1 5
2 2 5
2 3 5
3 2 5
3 3 5
2 1 3 0 8
0 0 5
0 2 5
1 1 5
1 3 5
2 0 5
2 2 5
3 1 5
3 3 5
2 2 3 0 10
0 0 5
0 1 5
1 0 5
1 1 5
1 2 5
2 1 5
2 2 5
2 3 5
3 2 5
3 3 5
1 0 0 2
1 1
3 1
1 1 0 2
1 1
2 1
1 2 0 2
1 1
2 1
1 3 0 2
0 1
2 1
```

## 3.10 Variable and cost function modeling

Modeling a Weighted CSP consists in creating variables and cost functions.
Domains of variables can be of two different types:

- enumerated domain allowing direct access to each value (array) and iteration on current domain in times proportional to the current number of values (double-linked list)
- interval domain represented by a lower value and an upper value only (useful for large domains)

**Warning**

Current implementation of toulbar2 has limited modeling and solving facilities for interval domains. There is no cost functions accepting both interval and enumerated variables for the moment, which means all the variables should have the same type.

Cost functions can be defined in extension (table or maps) or having a specific semantic.
Cost functions in extension depend on their arity:

- unary cost function (directly associated to an enumerated variable)
- binary and ternary cost functions (table of costs)
- n-ary cost functions (n $>=$ 4) defined by a list of tuples with associated costs and a default cost for missing tuples (allows for a compact representation)

Cost functions having a specific semantic (see Weighted Constraint Satisfaction Problem file format (wcsp)) are:

- simple arithmetic and scheduling (temporal disjunction) cost functions on interval variables
- global cost functions (*eg* soft alldifferent, soft global cardinality constraint, soft same, soft regular, etc) with three different propagator keywords:
  - *flow* propagator based on flow algorithms with "s" prefix in the keyword (*salldiff*, *sgcc*, *ssame*, *sregular*)
  - *DAG* propagator based on dynamic programming algorithms with "s" prefix and "dp" postfix (*samongdp*, salldiffdp, sgccdp, sregulardp, sgrammardp, smstdp, smaxdp)
  - *network* propagator based on cost function network decomposition with "w" prefix (*wsum*, *wvarsum*, *walldiff*, *wgcc*, *wsame*, *wsamegcc*, *wregular*, *wamong*, *wvaramong*, *woverlap*)

**Note**

The default semantics (using *var* keyword) of monolithic (flow and DAG-based propagators) global cost functions is to count the number of variables to change in order to restore consistency and to multiply it by the basecost. Other particular semantics may be used in conjunction with the flow-based propagator
The semantics of the network-based propagator approach is either a hard constraint ("hard" keyword) or a soft constraint by multiplying the number of changes by the basecost ("lin" or "var" keyword) or by multiplying the square value of the number of changes by the basecost ("quad" keyword)
A decomposable version exists for each monolithic global cost function, except grammar and MST. The decomposable ones may propagate less than their monolithic counterpart and they introduce extra variables but they can be much faster in practice

**Warning**

Each global cost function may have less than three propagators implemented
Current implementation of toulbar2 has limited solving facilities for monolithic global cost functions (no BTD-like methods nor variable elimination)
Current implementation of toulbar2 disallows global cost functions with less than or equal to three variables in their scope (use cost functions in extension instead)
Before modeling the problem using make and post, call ::tb2init method to initialize toulbar2 global variables
After modeling the problem using make and post, call WeightedCSP::sortConstraints method to initialize correctly the model before solving it

## 3.11 Solving cost function networks

After creating a Weighted CSP, it can be solved using a local search method INCOP (see WeightedCSPSolver↩
::narycsp) and/or an exact search method (see WeightedCSPSolver::solve).
Various options of the solving methods are controlled by ::Toulbar2 static class members (see files ./src/core/tb2types.hpp and ./src/tb2main.cpp).
A brief code example reading a wcsp problem given as a single command-line parameter and solving it:

```
#include "toulbar2lib.hpp"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv) {

    tb2init(); // must be call before setting specific ToulBar2 options and creating a model

    // Create a solver object
    WeightedCSPSolver *solver =
      WeightedCSPSolver::makeWeightedCSPSolver(MAX_COST);

    // Read a problem file in wcsp format
    solver->read_wcsp(argv[1]);

    ToulBar2::verbose = -1;  // change to 0 or higher values to see more trace information

    // Uncomment if solved using INCOP local search followed by a partial Limited Discrepancy Search with a
       maximum discrepancy of one
    //  ToulBar2::incop_cmd = "0 1 3 idwa 100000 cv v 0 200 1 0 0";
    //  ToulBar2::lds = -1;  // remove it or change to a positive value then the search continues by a
       complete B&B search method
    // Uncomment the following lines if solved using Decomposition Guided Variable Neighborhood Search with
       min-fill cluster decomposition and absorption
    // ToulBar2::lds = 4;
    // ToulBar2::restart = 10000;
    // ToulBar2::searchMethod = DGVNS;
    // ToulBar2::vnsNeighborVarHeur = CLUSTERRAND;
    // ToulBar2::boostingBTD = 0.7;
    // ToulBar2::varOrder = reinterpret_cast<char*>(-3);

    if (solver->solve()) {
        // show (sub-)optimal solution
        vector<Value> sol;
        Cost ub = solver->getSolution(sol);
        cout << "Best solution found cost: " << ub << endl;
        cout << "Best solution found:";
        for (unsigned int i=0; i<sol.size(); i++) cout << ((i>0)?",":"") << " x" << i << " = " << sol[i];
        cout << endl;
    } else {
        cout << "No solution found!" << endl;
    }
    delete solver;
}
```

#### See also

another code example in ./src/toulbar2test.cpp

#### Warning

variable domains must start at zero, otherwise recompile libtb2.so without flag WCSPFORMATONLY

## 3.12 Backtrack management

Used by backtrack search methods. Allows to copy / restore the current state using Store::store and Store::restore methods. All storable data modifications are trailed into specific stacks.

Trailing stacks are associated to each storable type:

- Store::storeValue for storable domain values ::StoreValue (value supports, etc)
- Store::storeCost for storable costs ::StoreCost (inside cost functions, etc)
- Store::storeDomain for enumerated domains (to manage holes inside domains)
- Store::storeConstraint for backtrackable lists of constraints
- Store::storeVariable for backtrackable lists of variables
- Store::storeSeparator for backtrackable lists of separators (see tree decomposition methods)
- Store::storeBigInteger for very large integers ::StoreBigInteger used in solution counting methods

Memory for each stack is dynamically allocated by part of $2^x$ with *x* initialized to ::STORE_SIZE and increased when needed.

#### Note

storable data are not trailed at depth 0.

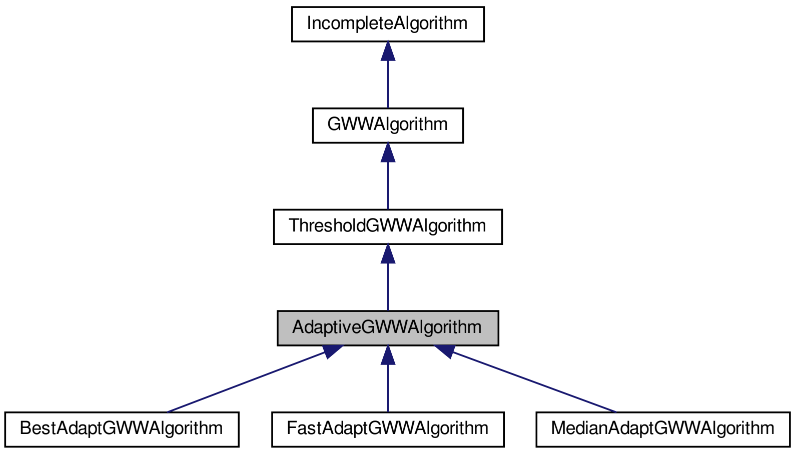
#### Warning

::StoreInt uses Store::storeValue stack (it assumes Value is encoded as int!).
Current storable data management is not multi-threading safe! (Store is a static virtual class relying on Store↩ Basic<T> static members)

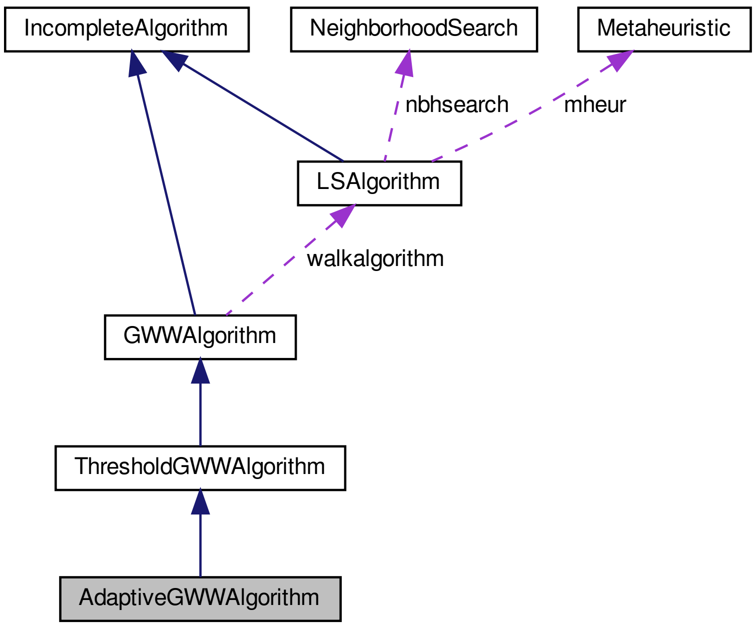
# 4 Class Documentation

## 4.1 AdaptiveGWWAlgorithm Class Reference

Inheritance diagram for AdaptiveGWWAlgorithm:



Collaboration diagram for AdaptiveGWWAlgorithm:



#### Public Attributes

- int nbkilled

#### Additional Inherited Members

### 4.1.1 Detailed Description

GWW with a threshold descent such as a given number of particles is regrouped

### 4.1.2 Member Data Documentation

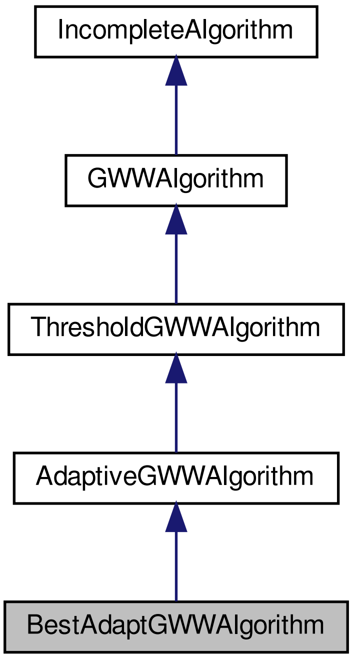
#### 4.1.2.1 nbkilled

```
int AdaptiveGWWAlgorithm::nbkilled
```

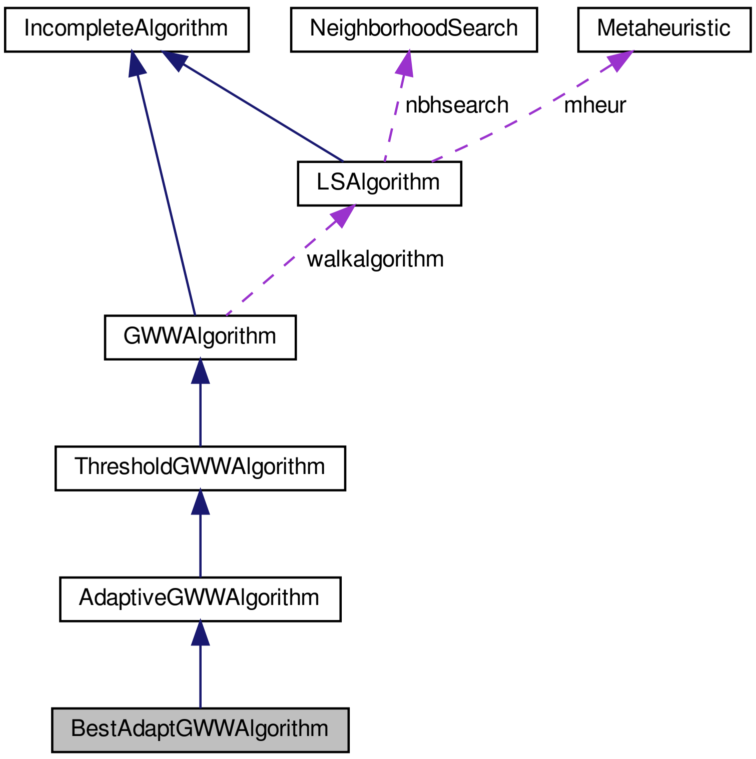
number of bad particles to be regrouped on good ones

## 4.2 BestAdaptGWWAlgorithm Class Reference

Inheritance diagram for BestAdaptGWWAlgorithm:



Collaboration diagram for BestAdaptGWWAlgorithm:



#### Public Attributes

- double bestdescent

#### Additional Inherited Members

### 4.2.1 Detailed Description

GWW with a descent depending on a distance between the worst and the best particle

### 4.2.2 Member Data Documentation

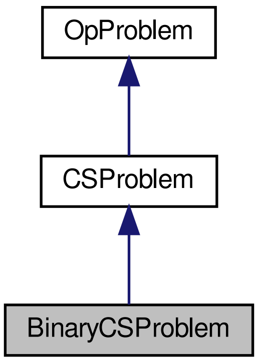
#### 4.2.2.1 bestdescent

```
double BestAdaptGWWAlgorithm::bestdescent
```

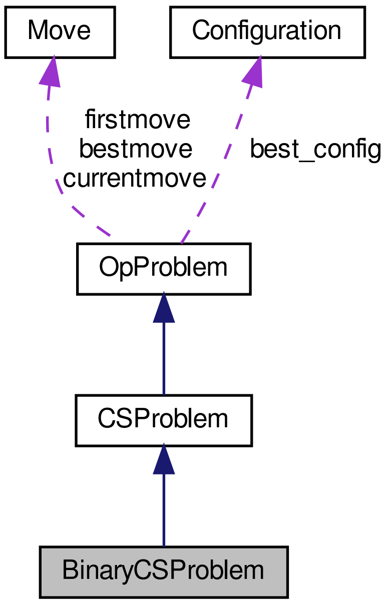
descent rate : porcentage of the distance between the worst and the best particles (between 0 and 1)

## 4.3 BinaryCSProblem Class Reference

Inheritance diagram for BinaryCSProblem:



Collaboration diagram for BinaryCSProblem:



### Public Attributes

- int $**$ constraints

#### Additional Inherited Members

### 4.3.1 Detailed Description

Binary CSPs : addition of the constraints array

### 4.3.2 Member Data Documentation

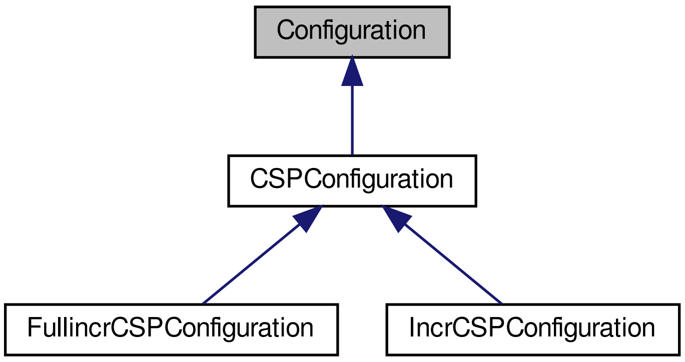
#### 4.3.2.1 constraints

```
int** BinaryCSProblem::constraints
```

for a couple (i,j) of variables, (i<j) , constraints[i][j] returns the constraint number + 1 if the variables are connected, 0 si the variables are not connected. It is assumed that at most one constraint exists between two variables (if not use WeightExtensionBinaryCSP class)

## 4.4 Configuration Class Reference

Inheritance diagram for Configuration:



#### Public Member Functions

- virtual void copy_element (Configuration ∗config2)
- virtual void init_conflicts ()
- virtual void incr_conflicts (int var, int val, int index, Long incr)
- virtual void set_conflicts (int var, int val, int index, Long nbconf)
- virtual Long get_conflicts (int var, int val, int index)
- virtual Long get_conflicts_problem (OpProblem ∗problem, int var, int val)
- virtual void update_conflicts (OpProblem ∗problem, Move ∗move)

#### Public Attributes

- int ∗ config
- Long valuation
- vector< int > var_conflict
- int regrouped

### 4.4.1 Detailed Description

the main class Configuration

### 4.4.2 Member Function Documentation

#### 4.4.2.1 copy_element()

```
void Configuration::copy_element (
            Configuration ∗ config2 )  [virtual]
```

copy a configuration config2 into this

References config, and valuation.

Referenced by LSAlgorithm::configurationmove(), GWWAlgorithm::populationkeepbest(), and GWWAlgorithm↩
::regrouping().

#### 4.4.2.2 get_conflicts()

```
Long Configuration::get_conflicts (
            int var,
            int val,
            int index )  [virtual]
```

get the number of conflicts (var,val) stored in the conflict datastructure

Reimplemented in FullincrCSPConfiguration.

#### 4.4.2.3 get_conflicts_problem()

```
Long Configuration::get_conflicts_problem (
            OpProblem ∗ problem,
            int var,
            int val )  [virtual]
```

get the number of conflicts of (var,val), computed if not stored

References OpProblem::compute_conflict().

Referenced by CSProblem::compute_var_conflict(), CSProblem::min_conflict_value(), and OpProblem::move_↩
execution().

#### 4.4.2.4 incr_conflicts()

```
void Configuration::incr_conflicts (
            int var,
            int val,
            int index,
            Long incr )  [virtual]
```

store the conflict of (var,val) incremented by incr

Referenced by INCOP::NaryCSProblem::config_evaluation().

#### 4.4.2.5 init_conflicts()

```
void Configuration::init_conflicts ( )  [virtual]
```

initialization to 0 of the conflict datastructure

Referenced by INCOP::NaryCSProblem::config_evaluation().

#### 4.4.2.6 set_conflicts()

```
void Configuration::set_conflicts (
            int var,
            int val,
            int index,
            Long nbconf )  [virtual]
```

store the number of conflicts nbconf of (var,val) in the conflict datastructure

#### 4.4.2.7 update_conflicts()

```
void Configuration::update_conflicts (
            OpProblem * problem,
            Move * move )  [virtual]
```

update the conflict datastructure after a move is done

References OpProblem::compute_conflict(), OpProblem::incr_update_conflicts(), and OpProblem::value2index().

Referenced by LSAlgorithm::configurationmove().

### 4.4.3 Member Data Documentation

#### 4.4.3.1 config

```
int* Configuration::config
```

the current values of the variables : implemented with an array of integers

Referenced by INCOP::NaryCSProblem::compute_conflict(), CSProblem::compute_var_conflict(), CSPMove↩ ::computetabumove(), INCOP::NaryConstraint::constraint_value(), copy_element(), INCOP::NaryCSProblem↩ ::create_configuration(), CSProblem::CSProblem(), CSProblem::init_domain_tabdomain(), CSProblem::min_↩ conflict_value(), and OpProblem::move_execution().

#### 4.4.3.2 regrouped

```
int Configuration::regrouped
```

indicates if the configuration has been regrouped before (for GWW)

Referenced by GWWAlgorithm::randomwalk(), GWWAlgorithm::regrouping(), and GWWAlgorithm::run().

#### 4.4.3.3 valuation

```
Long Configuration::valuation
```

the configuration value

Referenced by Metaheuristic::acceptance(), TabuSearch::acceptance(), Metropolis::acceptance(), Threshold↩ Accepting::acceptance(), SimulatedAnnealing::acceptance(), TabuAcceptingrate::acceptance(), copy_element(), INCOP::NaryCSProblem::create_configuration(), DynamicNeighborhoodSearch::dynamicmaxneighbors(), LS↩ Algorithm::isfeasible(), OpProblem::move_execution(), GWWAlgorithm::populationkeepbest(), GWWAlgorithm↩ ::randomwalk(), GWWAlgorithm::regrouping(), GWWAlgorithm::run(), and GWWAlgorithm::thresholdcomputedelta().

#### 4.4.3.4 var_conflict

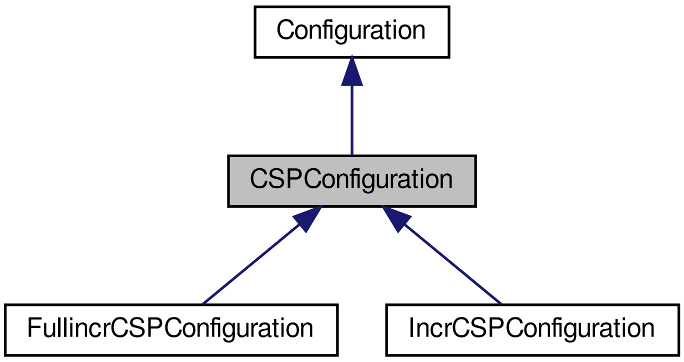
```
vector<int> Configuration::var_conflict
```

the variables taking part to a conflict : implemented with a vector

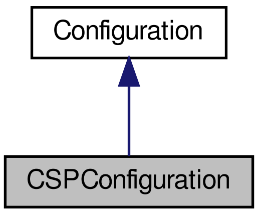
Referenced by CSProblem::compute_var_conflict(), DynamicNeighborhoodSearch::dynamicmaxneighbors(), CS↩ Problem::random_conflict_variable(), and CSProblem::random_variable().

## 4.5 CSPConfiguration Class Reference

Inheritance diagram for CSPConfiguration:



Collaboration diagram for CSPConfiguration:



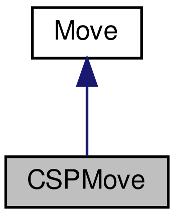
### Additional Inherited Members

### 4.5.1 Detailed Description

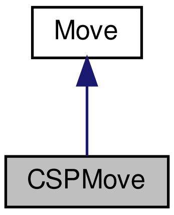
CSPConfiguration : for the CSPs

## 4.6 CSPMove Class Reference

Inheritance diagram for CSPMove:



Collaboration diagram for CSPMove:



#### Public Member Functions

- Move ∗ computetabumove (Configuration ∗config)

### 4.6.1 Detailed Description

class CSPMove : a classical move for a CSP : variable, value

### 4.6.2 Member Function Documentation

#### 4.6.2.1 computetabumove()

```
Move * CSPMove::computetabumove (
             Configuration * config )  [virtual]
```

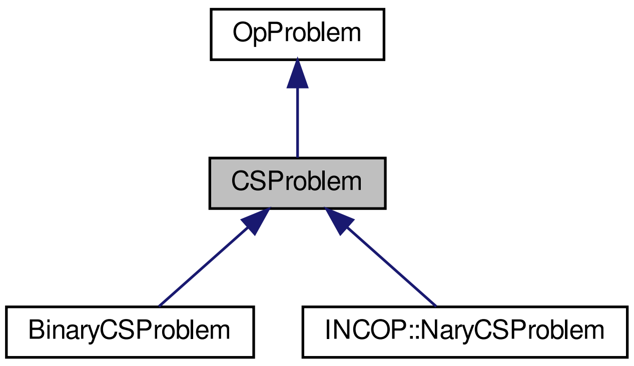
the move stored is the inverse of the move done

Reimplemented from Move.

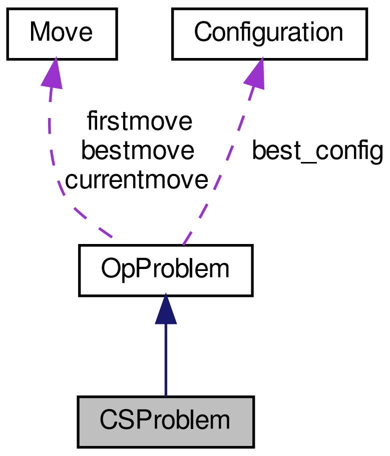
References Configuration::config.

## 4.7 CSProblem Class Reference

Inheritance diagram for CSProblem:



Collaboration diagram for CSProblem:



### Public Member Functions

- CSProblem (int nbvar, int nbconst)
- CSProblem (int nbvar, int nbconst, int lower)
- virtual int variable_domainsize (int var)
- virtual int random_variable (Configuration ∗configuration)
- virtual int random_conflict_variable (Configuration ∗configuration)
- virtual int random_value (int var, int val)
- virtual int min_conflict_value (int var, int val, Configuration ∗configuration)
- virtual void init_domains (int nbvar, int s)
- virtual void init_tabdomains (int s)
- void compute_var_conflict (Configuration ∗configuration)
- virtual void set_domains_connections (int ∗dom, vector< int > ∗tabledom, vector< int > ∗connect)
- virtual void init_domain_tabdomain ()

### Public Attributes

- int nbconst
- vector< int > ∗ tabdomains
- int ∗ domains
- vector< int > ∗ connections

### 4.7.1 Detailed Description

Finite domain CSP class

### 4.7.2 Constructor & Destructor Documentation

#### 4.7.2.1 CSProblem() [1/2]

```
CSProblem::CSProblem (
              int nbvar,
              int nbconst )
```

constructor

#### 4.7.2.2 CSProblem() [2/2]

```
CSProblem::CSProblem (
              int nbvar,
              int nbconst,
              int lower )
```

constructor with lower bound (stopping condition when it is reached)

References Configuration::config.

### 4.7.3 Member Function Documentation

#### 4.7.3.1 compute_var_conflict()

```
void CSProblem::compute_var_conflict (
             Configuration * configuration )  [virtual]
```

compute the variables in conflict : rebuilding the vector of conflict variables of the configuration

Reimplemented from OpProblem.

References Configuration::config, Configuration::get_conflicts_problem(), and Configuration::var_conflict.

#### 4.7.3.2 init_domain_tabdomain()

```
void CSProblem::init_domain_tabdomain ( )  [virtual]
```

initialization of the domains : call init_domains and init_tabdomains

References Configuration::config.

#### 4.7.3.3 init_domains()

```
void CSProblem::init_domains (
             int nbvar,
             int s )  [virtual]
```

standard domain initialization : a unique domain number 0 for all variables

#### 4.7.3.4 init_tabdomains()

```
void CSProblem::init_tabdomains (
             int s )  [virtual]
```

standard unique domain : integers from 0 to s-1

#### 4.7.3.5 min_conflict_value()

```
int CSProblem::min_conflict_value (
             int var,
             int val,
             Configuration * configuration )  [virtual]
```

a value in the domain minimizing the conflict with the configuration (implementation of Minton min-conflict heuristics) returns the index of the value in the domain

References Configuration::config, Configuration::get_conflicts_problem(), NeighborhoodSearch::val_conflict, and NeighborhoodSearch::var_conflict.

#### 4.7.3.6 random_conflict_variable()

```
int CSProblem::random_conflict_variable (
             Configuration * configuration )  [virtual]
```

a variable taking part to a conflict in the configuration

References Configuration::var_conflict.

#### 4.7.3.7 random_value()

```
int CSProblem::random_value (
             int var,
             int val )  [virtual]
```

a value for variable var, randomly chosen in its domain, if possible distinct with val : returns the index of the value in the domain

#### 4.7.3.8 random_variable()

```
int CSProblem::random_variable (
             Configuration * configuration )  [virtual]
```

a variable randomly chosen

References Configuration::var_conflict.

#### 4.7.3.9 set_domains_connections()

```
void CSProblem::set_domains_connections (
             int * dom,
             vector< int > * tabledom,
             vector< int > * connect )  [virtual]
```

set the domains and connections of a problem

Referenced by INCOP::NaryCSProblem::create_configuration().

#### 4.7.3.10 variable_domainsize()

```
int CSProblem::variable_domainsize (
             int var )  [virtual]
```

the domain size of variable var

### 4.7.4 Member Data Documentation

#### 4.7.4.1 connections

`vector<int>* CSProblem::connections`

connections table : for each variable, vector of connected variables

#### 4.7.4.2 domains

`int* CSProblem::domains`

for each variable, domain number : index in tabdomains array

#### 4.7.4.3 nbconst

`int CSProblem::nbconst`

constraint number

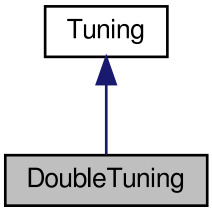
#### 4.7.4.4 tabdomains

`vector<int>* CSProblem::tabdomains`

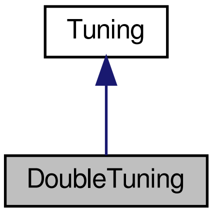
domain array : each domain is implemented by a vector of integers

## 4.8 DoubleTuning Class Reference

Inheritance diagram for DoubleTuning:



Collaboration diagram for DoubleTuning:

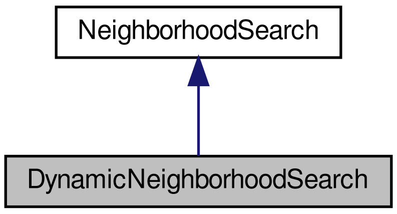


### 4.8.1 Detailed Description

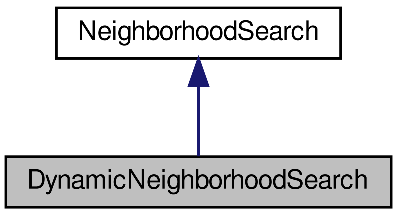
Automatic tuning of a local search algorithm with two parameters

## 4.9 DynamicNeighborhoodSearch Class Reference

Inheritance diagram for DynamicNeighborhoodSearch:



Collaboration diagram for DynamicNeighborhoodSearch:



### Public Member Functions

- void dynamicmaxneighbors (int &maxneigh, int &minneigh, int nbmoves)

### Public Attributes

- int initmaxneighbors
- int initminneighbors
- int adjustperiod

### 4.9.1 Detailed Description

Neighborhood with dynamic parameter tuning

### 4.9.2 Member Function Documentation

#### 4.9.2.1 dynamicmaxneighbors()

```
void DynamicNeighborhoodSearch::dynamicmaxneighbors (
             int & maxneigh,
             int & minneigh,
             int nbmoves )  [virtual]
```

adjust the parameters maxneighbors and minneighbors

Reimplemented from NeighborhoodSearch.

References OpProblem::adjust_parameters(), OpProblem::domainsize, Configuration::valuation, and Configuration↩
::var_conflict.

### 4.9.3 Member Data Documentation

#### 4.9.3.1 adjustperiod

```
int DynamicNeighborhoodSearch::adjustperiod
```

parameter readjustment period

#### 4.9.3.2 initmaxneighbors

```
int DynamicNeighborhoodSearch::initmaxneighbors
```

initial value of maxneighbors parameter

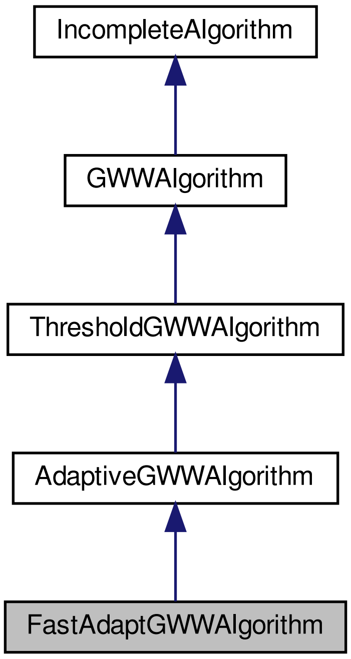
#### 4.9.3.3 initminneighbors

```
int DynamicNeighborhoodSearch::initminneighbors
```

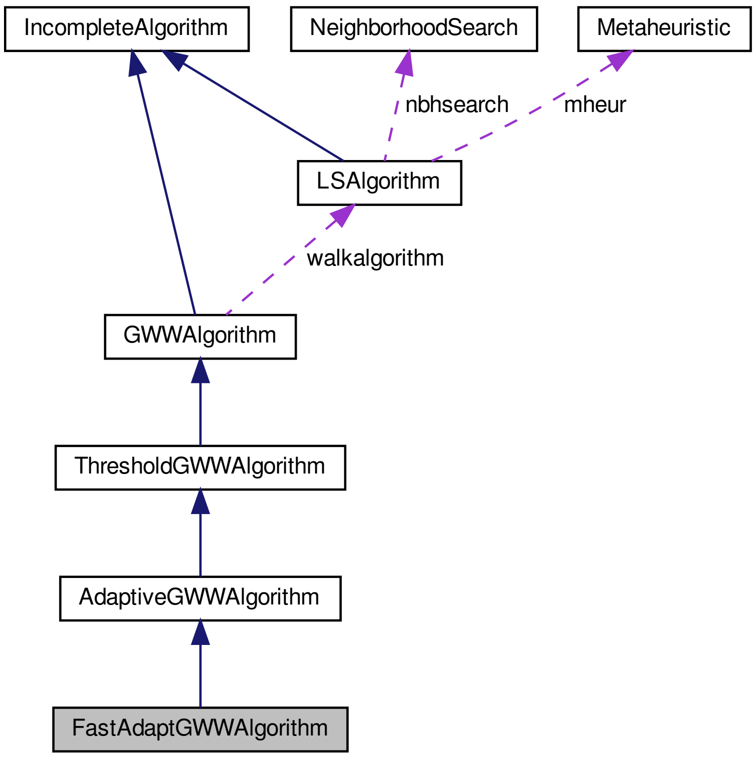
initial value of minneighbors parameter

## 4.10 FastAdaptGWWAlgorithm Class Reference

Inheritance diagram for FastAdaptGWWAlgorithm:



Collaboration diagram for FastAdaptGWWAlgorithm:



#### Public Attributes

- double thresholddescent

#### Additional Inherited Members

### 4.10.1 Detailed Description

GWW with a threshold descent at the lowest value obtained by AdaptiveGWWAlgorithm et FastStandardGWW↩ Algorithm using a number of particles to be redistributed and a rate

### 4.10.2 Member Data Documentation

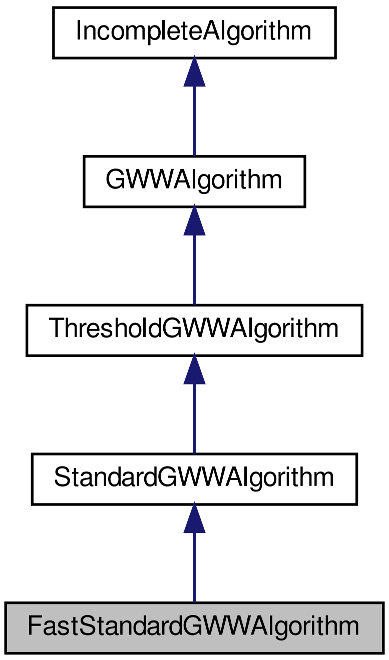
#### 4.10.2.1 thresholddescent

```
double FastAdaptGWWAlgorithm::thresholddescent
```

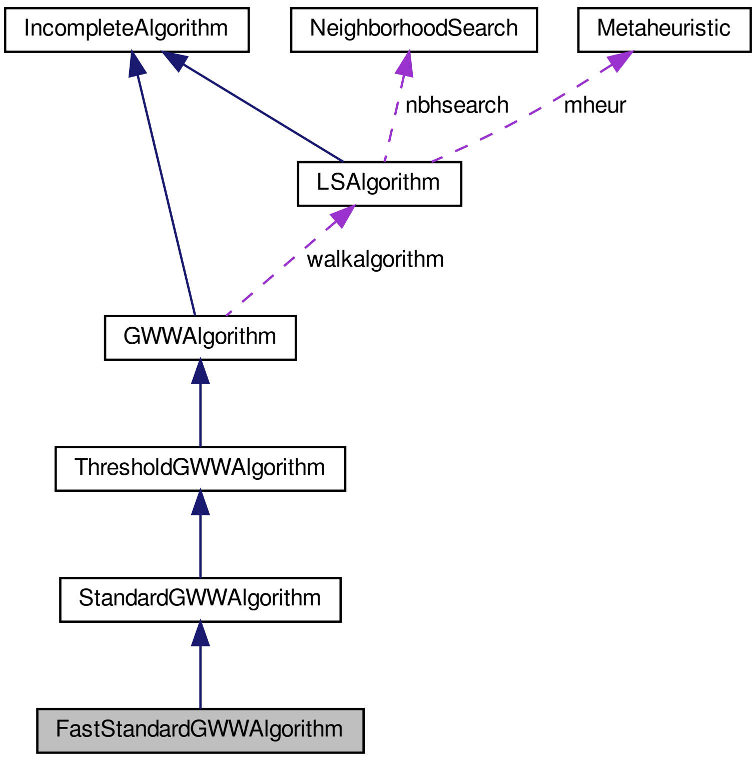
threshold descent rate

## 4.11 FastStandardGWWAlgorithm Class Reference

Inheritance diagram for FastStandardGWWAlgorithm:



Collaboration diagram for FastStandardGWWAlgorithm:



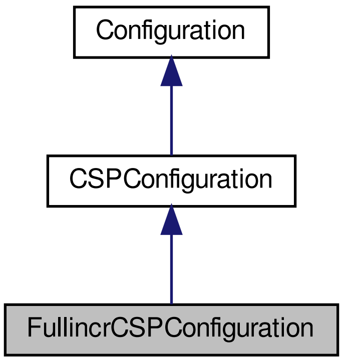
**Additional Inherited Members**

### 4.11.1 Detailed Description

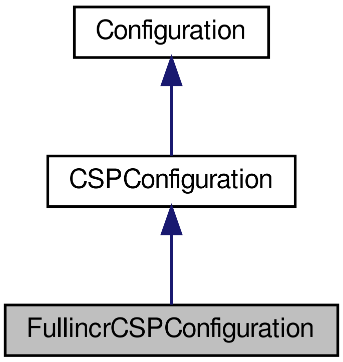
StandardGWW with a threshold descent at least until the worst particle

## 4.12 FullincrCSPConfiguration Class Reference

Inheritance diagram for FullincrCSPConfiguration:



Collaboration diagram for FullincrCSPConfiguration:



#### Public Member Functions

- Long get_conflicts (int var, int val, int index)

#### Additional Inherited Members

### 4.12.1 Detailed Description

Full incremental evaluation : the participation of every value of every variable is stored in the 2 dimension array tabconflicts (variable, valueindex)

### 4.12.2 Member Function Documentation

#### 4.12.2.1 get_conflicts()

```
Long FullincrCSPConfiguration::get_conflicts (
             int var,
             int val,
             int index )  [virtual]
```

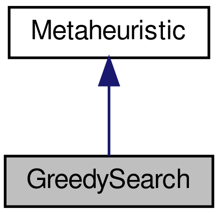
get the number of conflicts (var,val) stored in the conflict datastructure using the value index in the domain

Reimplemented from Configuration.

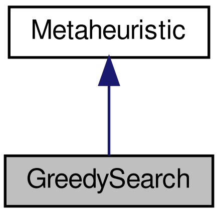
References OpProblem::fullincr_update_conflicts().

## 4.13 GreedySearch Class Reference

Inheritance diagram for GreedySearch:



Collaboration diagram for GreedySearch:



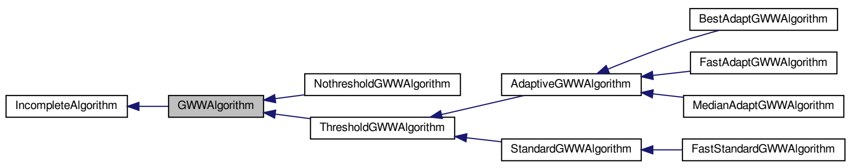
### Additional Inherited Members

### 4.13.1 Detailed Description

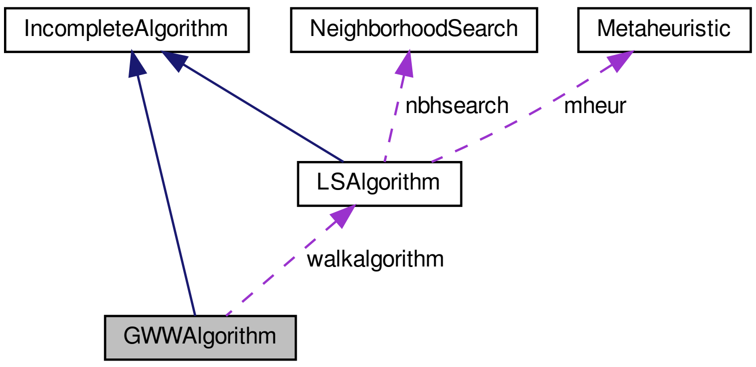
Greedy walk : a neighbor with better or same cost as the current configuration is accepted

## 4.14 GWWAlgorithm Class Reference

Inheritance diagram for GWWAlgorithm:



Collaboration diagram for GWWAlgorithm:



### Public Member Functions

- virtual void populationrandomwalk (OpProblem ∗problem, Configuration ∗∗population)
- virtual int nb_threshold_population (Configuration ∗∗population)
- void randomwalk (OpProblem ∗problem, Configuration ∗configuration)
- void initthreshold (Configuration ∗∗population, int popsize)
- virtual void thresholdupdate ()
- virtual void thresholdcomputedelta (Configuration ∗∗population)
- void run (OpProblem ∗problem, Configuration ∗∗population)
- virtual void regrouping (Configuration ∗∗population)
- void populationkeepbest (OpProblem ∗problem, Configuration ∗∗population)
- virtual void thresholdchangesupdate ()

#### Public Attributes

- int populationsize
- int regrouptest
- int lastmovedescent
- int elitism
- int nomovestop
- Long thresholddelta
- int nbiteration
- int thresholdchanges
- int total_nhtries
- int total_nbmoves
- LSAlgorithm ∗ walkalgorithm

### 4.14.1 Detailed Description

the GWW (Go with the winners) algorithms : the different subclasses differ by the way a threshold is managed and the particles are regrouped

### 4.14.2 Member Function Documentation

#### 4.14.2.1 initthreshold()

```
void GWWAlgorithm::initthreshold (
            Configuration ** population,
            int popsize )  [virtual]
```

intialization of the threshold

Reimplemented from IncompleteAlgorithm.

References IncompleteAlgorithm::threshold, thresholdchanges, and walkalgorithm.

#### 4.14.2.2 nb_threshold_population()

```
int GWWAlgorithm::nb_threshold_population (
            Configuration ** population )  [virtual]
```

the number of particles at the threshold (for statistics) , the population being yet sorted at the function call

References populationsize, IncompleteAlgorithm::threshold, and walkalgorithm.

Referenced by run().

#### 4.14.2.3 populationkeepbest()

```
void GWWAlgorithm::populationkeepbest (
            OpProblem * problem,
            Configuration ** population )
```

in case of elitism, the best particle is put into the population

References OpProblem::best_config, Configuration::copy_element(), populationsize, and Configuration::valuation.

Referenced by run().

#### 4.14.2.4 populationrandomwalk()

```
void GWWAlgorithm::populationrandomwalk (
            OpProblem * problem,
            Configuration ** population )  [virtual]
```

local search on the whole population

References OpProblem::lower_bound, LSAlgorithm::nbmoves, LSAlgorithm::nhtries, populationsize, randomwalk(), total_nbmoves, total_nhtries, and walkalgorithm.

Referenced by run().

#### 4.14.2.5 randomwalk()

```
void GWWAlgorithm::randomwalk (
            OpProblem * problem,
            Configuration * configuration )  [virtual]
```

a local search for a particle

Reimplemented from IncompleteAlgorithm.

References LSAlgorithm::configurationmove(), lastmovedescent, OpProblem::lower_bound, nomovestop, Configuration::regrouped, IncompleteAlgorithm::threshold, thresholdupdate(), Configuration::valuation, walkalgorithm, and LSAlgorithm::walklength.

Referenced by LSAlgorithm::isfeasible(), and populationrandomwalk().

#### 4.14.2.6 regrouping()

```
void GWWAlgorithm::regrouping (
            Configuration ** population )  [virtual]
```

regrouping of the best particles on the good ones

References Configuration::copy_element(), NothresholdGWWAlgorithm::nbkilled, populationsize, Configuration↩
::regrouped, IncompleteAlgorithm::threshold, and Configuration::valuation.

#### 4.14.2.7 run()

```
void GWWAlgorithm::run (
            OpProblem * problem,
            Configuration ** population )  [virtual]
```

main function for running the algorithm

Reimplemented from IncompleteAlgorithm.

References elitism, OpProblem::lower_bound, nb_threshold_population(), nbiteration, populationkeepbest(), populationrandomwalk(), populationsize, Configuration::regrouped, regrouptest, IncompleteAlgorithm::threshold, thresholdchanges, thresholdchangesupdate(), thresholdcomputedelta(), thresholddelta, thresholdupdate(), total_↩ nbmoves, total_nhtries, Configuration::valuation, and walkalgorithm.

#### 4.14.2.8 thresholdchangesupdate()

```
void GWWAlgorithm::thresholdchangesupdate ( )  [virtual]
```

incrementing the threshold updates counter (for the statistics)

References thresholdchanges.

Referenced by run().

#### 4.14.2.9 thresholdcomputedelta()

```
void GWWAlgorithm::thresholdcomputedelta (
            Configuration ** population )  [virtual]
```

method for computing the threshold decrement

References NothresholdGWWAlgorithm::nbkilled, populationsize, IncompleteAlgorithm::threshold, thresholddelta, Configuration::valuation, and walkalgorithm.

Referenced by run().

#### 4.14.2.10 thresholdupdate()

```
void GWWAlgorithm::thresholdupdate ( )  [virtual]
```

method for lowering the threshold( the delta has already been computed)

References IncompleteAlgorithm::threshold, thresholddelta, and walkalgorithm.

Referenced by randomwalk(), and run().

### 4.14.3 Member Data Documentation

#### 4.14.3.1 elitism

```
int GWWAlgorithm::elitism
```

elitism parameter : is the best particle put again in the population at each regroupment ( 1 yes, 0 no)

Referenced by run(), and SimulatedAnnealing::SimulatedAnnealing().

#### 4.14.3.2 lastmovedescent

```
int GWWAlgorithm::lastmovedescent
```

parameter if the threshold is lowered at the last move of the walk (for trying to avoid the particle to be redistributed (1 yes, 0 no)

Referenced by randomwalk(), and SimulatedAnnealing::SimulatedAnnealing().

#### 4.14.3.3 nbiteration

```
int GWWAlgorithm::nbiteration
```

the maximum number of iterations : useful when no threshold is managed (NothresholdGWWAlgorithm)

Referenced by run(), and SimulatedAnnealing::SimulatedAnnealing().

#### 4.14.3.4 nomovestop

```
int GWWAlgorithm::nomovestop
```

parameter for stopping the walk in case of stagnation (1 yes, 0 no)

Referenced by randomwalk(), and SimulatedAnnealing::SimulatedAnnealing().

#### 4.14.3.5 populationsize

```
int GWWAlgorithm::populationsize
```

number of particles

Referenced by nb_threshold_population(), populationkeepbest(), populationrandomwalk(), regrouping(), run(), SimulatedAnnealing::SimulatedAnnealing(), and thresholdcomputedelta().

#### 4.14.3.6 regrouptest

```
int GWWAlgorithm::regrouptest
```

walk indicator : a walk is performed only is the particle has been regrouped : (1 yes, 0 no) (useful for a standard GWW with random walk (and no local search))

Referenced by run(), and SimulatedAnnealing::SimulatedAnnealing().

#### 4.14.3.7 thresholdchanges

```
int GWWAlgorithm::thresholdchanges
```

number of threshold changes (for the statistics)

Referenced by initthreshold(), run(), and thresholdchangesupdate().

#### 4.14.3.8 thresholddelta

```
Long GWWAlgorithm::thresholddelta
```

the threshold decrement (compted by thresholdcomputedelta)

Referenced by run(), thresholdcomputedelta(), and thresholdupdate().

#### 4.14.3.9 total_nbmoves

```
int GWWAlgorithm::total_nbmoves
```

total number of moves between 2 regroupments (for the statistics)

Referenced by populationrandomwalk(), and run().

#### 4.14.3.10 total_nhtries

```
int GWWAlgorithm::total_nhtries
```

total number of move tries between 2 regroupments (for the statistics)

Referenced by populationrandomwalk(), and run().

#### 4.14.3.11 walkalgorithm

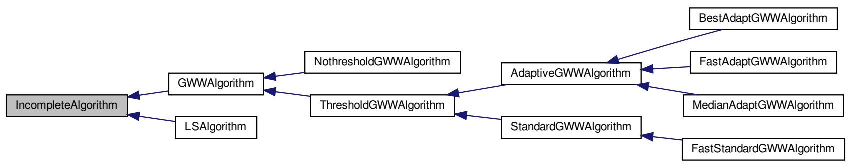
```
LSAlgorithm* GWWAlgorithm::walkalgorithm
```

the local search algorithm used

Referenced by initthreshold(), nb_threshold_population(), populationrandomwalk(), randomwalk(), run(), SimulatedAnnealing::SimulatedAnnealing(), thresholdcomputedelta(), and thresholdupdate().

## 4.15 IncompleteAlgorithm Class Reference

Inheritance diagram for IncompleteAlgorithm:



### Public Member Functions

- virtual void randomwalk (OpProblem ∗problem, Configuration ∗configuration)
- virtual void run (OpProblem ∗problem, Configuration ∗∗population)

### Public Attributes

- Long threshold

### 4.15.1 Detailed Description

Root class of algorithms

### 4.15.2 Member Function Documentation

#### 4.15.2.1 randomwalk()

```
void IncompleteAlgorithm::randomwalk (
            OpProblem ∗ problem,
            Configuration ∗ configuration )  [virtual]
```

walk for a particule

Reimplemented in GWWAlgorithm.

#### 4.15.2.2 run()

```
void IncompleteAlgorithm::run (
            OpProblem * problem,
            Configuration ** population )  [virtual]
```

Run the algorithm on a population (array of configurations)

Reimplemented in GWWAlgorithm.

### 4.15.3 Member Data Documentation

#### 4.15.3.1 threshold

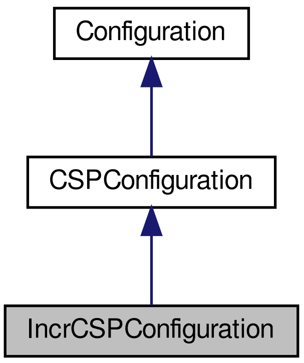
```
Long IncompleteAlgorithm::threshold
```

a threshold can be used to forbid moves above this threshold (used in LSAlgorithms implementing walks inside GWW)

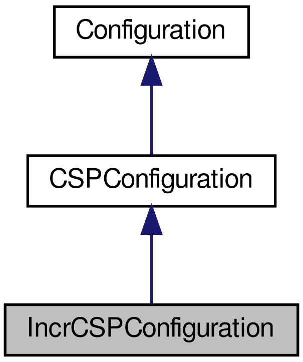
Referenced by GWWAlgorithm::initthreshold(), LSAlgorithm::isfeasible(), GWWAlgorithm::nb_threshold_↩ population(), GWWAlgorithm::randomwalk(), GWWAlgorithm::regrouping(), GWWAlgorithm::run(), LSAlgorithm↩ ::test_bestfound(), GWWAlgorithm::thresholdcomputedelta(), and GWWAlgorithm::thresholdupdate().

## 4.16 IncrCSPConfiguration Class Reference

Inheritance diagram for IncrCSPConfiguration:



Collaboration diagram for IncrCSPConfiguration:



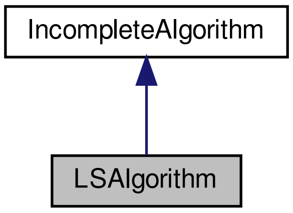
#### Additional Inherited Members

### 4.16.1 Detailed Description

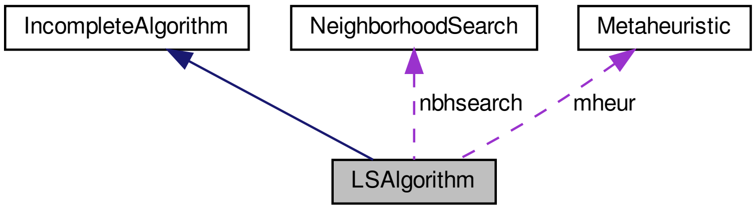
Incremental evaluation with storage in the conflict datastructure tabconflicts the participation of the current values of the configuration

## 4.17 LSAlgorithm Class Reference

Inheritance diagram for LSAlgorithm:



Collaboration diagram for LSAlgorithm:



#### Public Member Functions

- virtual int isfeasible (Move ∗move)
- virtual int configurationmove (OpProblem ∗problem, Configuration ∗configuration)
- int test_bestfound (Move ∗move)

#### Public Attributes

- int walklength
- NeighborhoodSearch ∗ nbhsearch
- Metaheuristic ∗ mheur
- int nhtries
- int nbmoves

### 4.17.1 Detailed Description

The class of local search algorithm on one particle : the random walk is parameterized with the walk lengh,a neighborhood and a metaheuristics

### 4.17.2 Member Function Documentation

#### 4.17.2.1 configurationmove()

```
int LSAlgorithm::configurationmove (
            OpProblem ∗ problem,
            Configuration ∗ configuration ) [virtual]
```

Neighborhood exploration algorithm for selecting and do a move from the current configuration : returns 1 if a move has been done and 0 if no move has been done

References OpProblem::best_config, OpProblem::bestmove, OpProblem::compute_var_conflict(), Configuration::copy_element(), Move::copymove(), OpProblem::currentmove, OpProblem::firstmove, OpProblem::move_execution(), OpProblem::next_move(), and Configuration::update_conflicts().

Referenced by GWWAlgorithm::randomwalk().

#### 4.17.2.2 isfeasible()

```
int LSAlgorithm::isfeasible (
            Move * move )  [virtual]
```

feasability of a move (under or at threshold level pour GWW walks)

References OpProblem::lower_bound, GWWAlgorithm::randomwalk(), IncompleteAlgorithm::threshold, and Configuration::valuation.

#### 4.17.2.3 test_bestfound()

```
int LSAlgorithm::test_bestfound (
            Move * move )
```

test if a global best configuration has been found (returns 1 in that case)

References IncompleteAlgorithm::threshold.

### 4.17.3 Member Data Documentation

#### 4.17.3.1 mheur

```
Metaheuristic* LSAlgorithm::mheur
```

the metaheuristics used

#### 4.17.3.2 nbhsearch

```
NeighborhoodSearch* LSAlgorithm::nbhsearch
```

the way the neighborhood is explored

#### 4.17.3.3 nbmoves

```
int LSAlgorithm::nbmoves
```

number of moves done

Referenced by GWWAlgorithm::populationrandomwalk().

#### 4.17.3.4 nhtries

```
int LSAlgorithm::nhtries
```

number of move tries (for statistics)

Referenced by GWWAlgorithm::populationrandomwalk().

#### 4.17.3.5 walklength

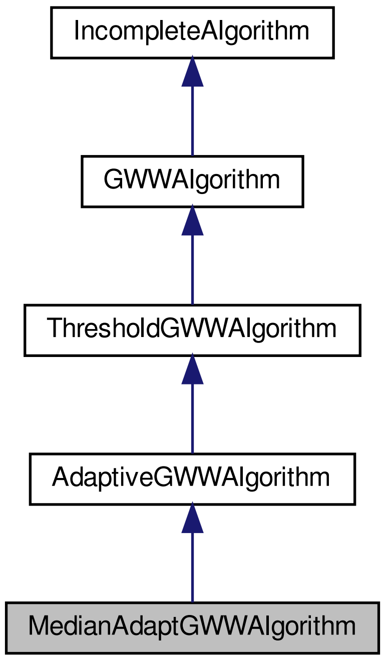
```
int LSAlgorithm::walklength
```

walk length

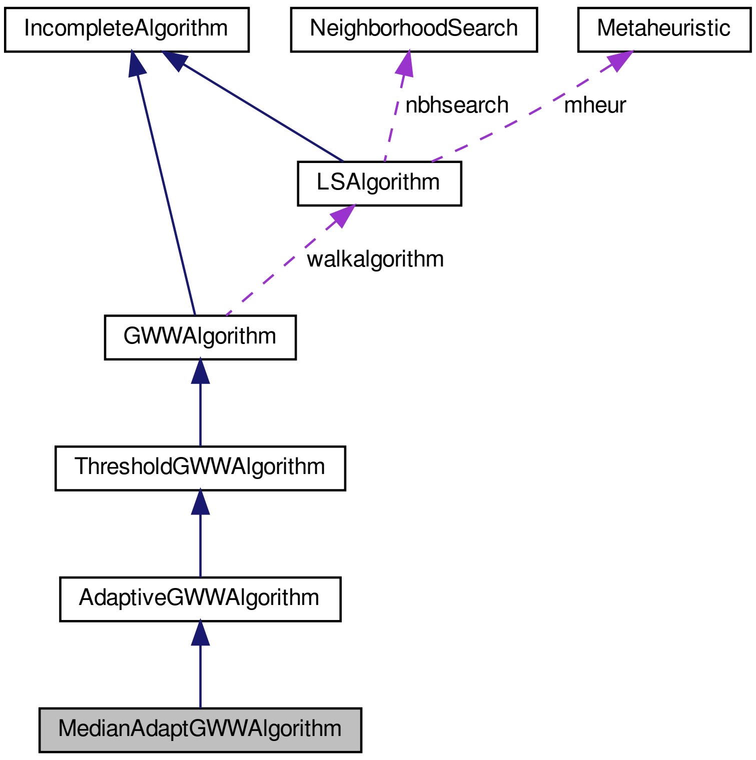
Referenced by GWWAlgorithm::randomwalk().

## 4.18 MedianAdaptGWWAlgorithm Class Reference

Inheritance diagram for MedianAdaptGWWAlgorithm:



Collaboration diagram for MedianAdaptGWWAlgorithm:



### Public Attributes

- double mediandescent

### Additional Inherited Members

### 4.18.1 Detailed Description

GWW with a descent depending on a distance between the worst and the median particle

### 4.18.2 Member Data Documentation

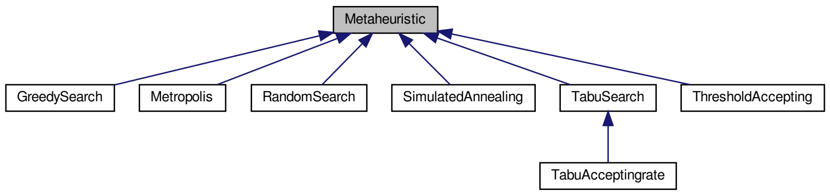
#### 4.18.2.1 mediandescent

`double MedianAdaptGWWAlgorithm::mediandescent`

descent rate : porcentage of the distance between the worst and the median particles (between 0 and 1)

## 4.19 Metaheuristic Class Reference

Inheritance diagram for Metaheuristic:



### Public Member Functions

- virtual void executebeforemove (Move ∗move, Configuration ∗configuration, OpProblem ∗problem)
- virtual void reinit (OpProblem ∗problem)
- virtual int acceptance (Move ∗move, Configuration ∗config)

### 4.19.1 Detailed Description

Root class for Metaheuritics

### 4.19.2 Member Function Documentation

#### 4.19.2.1 acceptance()

```
int Metaheuristic::acceptance (
            Move * move,
            Configuration * config )  [virtual]
```

acceptance condition of a move : returns 1 if the move is accepted

Reimplemented in TabuAcceptingrate, SimulatedAnnealing, ThresholdAccepting, Metropolis, and TabuSearch.

References Configuration::valuation.

#### 4.19.2.2 executebeforemove()

```
void Metaheuristic::executebeforemove (
            Move * move,
            Configuration * configuration,
            OpProblem * problem )  [virtual]
```

update of the metaheuristic data just before a move is performed

Reimplemented in SimulatedAnnealing, ThresholdAccepting, and TabuSearch.

#### 4.19.2.3 reinit()

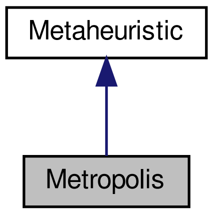
```
void Metaheuristic::reinit (
             OpProblem * problem )  [virtual]
```

initialization of the meteheuristic data at the beginning of a local search

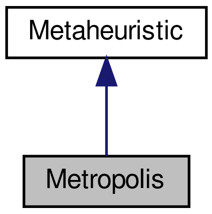
Reimplemented in ThresholdAccepting, and TabuSearch.

## 4.20 Metropolis Class Reference

Inheritance diagram for Metropolis:



Collaboration diagram for Metropolis:



### Public Member Functions

- int acceptance (Move ∗move, Configuration ∗config)

### 4.20.1 Detailed Description

Metropolis algorithm : a unique parameter - a constant temperature

### 4.20.2 Member Function Documentation

#### 4.20.2.1 acceptance()

```
int Metropolis::acceptance (
            Move * move,
            Configuration * config )  [virtual]
```

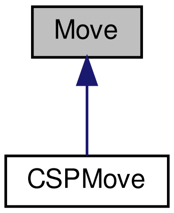
the classical Metropolis formula for accepting a bad move : probability = exp (-evaluationdelta/temperature)

Reimplemented from Metaheuristic.

References Configuration::valuation.

## 4.21 Move Class Reference

Inheritance diagram for Move:



### Public Member Functions

- virtual int eqmove (Move ∗move1)
- virtual void copymove (Move ∗move)
- virtual Move ∗ computetabumove (Configuration ∗config)

### 4.21.1 Detailed Description

root class Move

### 4.21.2 Member Function Documentation

#### 4.21.2.1 computetabumove()

```
virtual Move* Move::computetabumove (
             Configuration * config )  [virtual]
```

the move to be put in the tabu list (to be implemented in the subclasses)

Reimplemented in CSPMove.

Referenced by TabuSearch::executebeforemove().

#### 4.21.2.2 copymove()

```
void Move::copymove (
             Move * move )  [virtual]
```

copy of move move1 into this

Referenced by LSAlgorithm::configurationmove().

#### 4.21.2.3 eqmove()

```
int Move::eqmove (
             Move * move1 )  [virtual]
```

the test of equality of a move (used for searching a move in the tabu list)

## 4.22 INCOP::NaryConstraint Class Reference

### Public Member Functions

- Long constraint_value (Configuration ∗configuration)
- int nbtuples (vector< int > ∗tabdomaines)

### Public Attributes

- vector< int > constrainedvariables
- vector< Long > tuplevalues

### 4.22.1 Detailed Description

Nary constraint in extension with weigths defined on the tuples

### 4.22.2 Member Function Documentation

#### 4.22.2.1 constraint_value()

```
Long INCOP::NaryConstraint::constraint_value (
            Configuration * configuration )
```

Constraint Evalution : searching in the tuple table

References Configuration::config.

#### 4.22.2.2 nbtuples()

```
int INCOP::NaryConstraint::nbtuples (
            vector< int > * tabdomaines )
```

nombre de n-uplets d'une contrainte

### 4.22.3 Member Data Documentation

#### 4.22.3.1 constrainedvariables

```
vector<int> INCOP::NaryConstraint::constrainedvariables
```

variables linked by the constraint

Referenced by INCOP::NaryCSProblem::create_configuration().

#### 4.22.3.2 tuplevalues

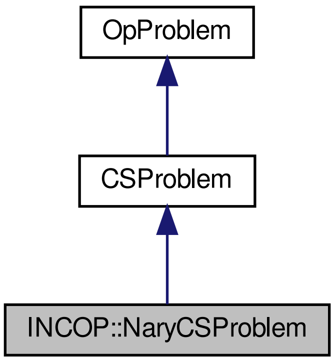
```
vector<Long> INCOP::NaryConstraint::tuplevalues
```

table of valued tuples

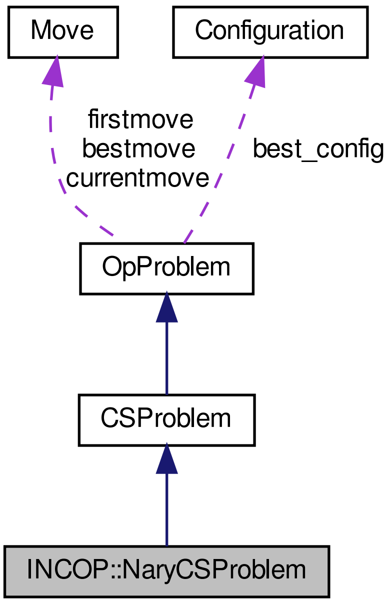
Referenced by INCOP::NaryCSProblem::create_configuration().

## 4.23 INCOP::NaryCSProblem Class Reference

Inheritance diagram for INCOP::NaryCSProblem:



Collaboration diagram for INCOP::NaryCSProblem:



### Public Member Functions

- Long config_evaluation (Configuration ∗configuration)
- Long compute_conflict (Configuration ∗configuration, int var, int val)
- Configuration ∗ create_configuration ()

#### Additional Inherited Members

### 4.23.1 Detailed Description

NaryCSPs solved as weighted Max-CSPs with weights on the tuples

### 4.23.2 Member Function Documentation

#### 4.23.2.1 compute_conflict()

```
Long INCOP::NaryCSProblem::compute_conflict (
            Configuration * configuration,
            int var,
            int val )  [virtual]
```

number of conflicts of a simple assignment in a complete configuration

calcul du nombre de conflits d'une affectation - appele par l'évaluation d'un mouvement (cas incr)

Reimplemented from OpProblem.

References Configuration::config.

#### 4.23.2.2 config_evaluation()

```
Long INCOP::NaryCSProblem::config_evaluation (
            Configuration * configuration )  [virtual]
```

evaluation and filling the conflict datastructure

code optimisé pour configuration semi-incrementale IncrCSPConfiguration

Reimplemented from OpProblem.

References Configuration::incr_conflicts(), and Configuration::init_conflicts().

#### 4.23.2.3 create_configuration()

```
Configuration * INCOP::NaryCSProblem::create_configuration ( )  [virtual]
```

choice of incrementality mode : IncrCSPConfiguration ou FullincrCSPConfiguration

utilisation des configurations "semi-incrementales"IncrCSPConfiguration - les conflits des valeurs courantes des variables sont stockés dans le tableau tabconflicts ou tout-incrémentales FullincrCSPConfiguration : les conflits de toutes les valeurs avec la configuration courante sont maintenus dans tabconflicts

Reimplemented from CSProblem.

References OpProblem::allocate_moves(), OpProblem::best_config, Configuration::config, INCOP::Nary↵
Constraint::constrainedvariables, OpProblem::domainsize, OpProblem::lower_bound, CSProblem::set_domains↵
_connections(), INCOP::NaryConstraint::tuplevalues, and Configuration::valuation.

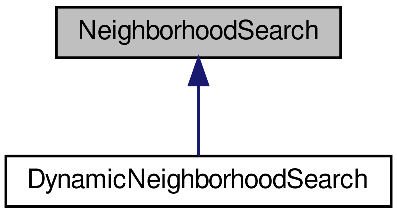
## 4.24 INCOP::NaryVariable Class Reference

### 4.24.1 Detailed Description

Variable constrained by a n-ary constraint

## 4.25 NeighborhoodSearch Class Reference

Inheritance diagram for NeighborhoodSearch:



**Public Attributes**

- int minneighbors
- int maxneighbors
- int finished
- int var_conflict
- int val_conflict

### 4.25.1 Detailed Description

Class NeighborhoodSearch : how the neighborhood is explored

### 4.25.2 Member Data Documentation

#### 4.25.2.1 finished

```
int NeighborhoodSearch::finished
```

behavior indicator when the neighborhood is exhausted and no neighbor has been accepted : 0 stagnation, 1 the 1st feasible move is selected, k the best feasible among k tried but not accepted moves is selected

#### 4.25.2.2 maxneighbors

`int NeighborhoodSearch::maxneighbors`

maximum number of explored neighbors

#### 4.25.2.3 minneighbors

`int NeighborhoodSearch::minneighbors`

minimum number of visited neighbors

#### 4.25.2.4 val_conflict

`int NeighborhoodSearch::val_conflict`

restriction indicator to best values of a variable (0 no restriction , 1 restriction)

Referenced by CSProblem::min_conflict_value().

#### 4.25.2.5 var_conflict

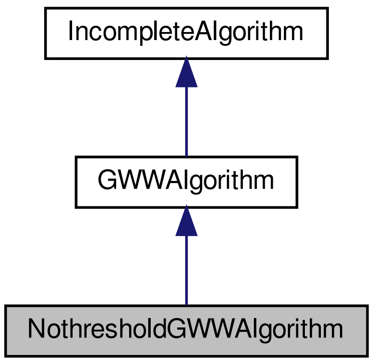
`int NeighborhoodSearch::var_conflict`

restriction indicator to variables participating in a conflict (0 no restriction, 1 restriction)

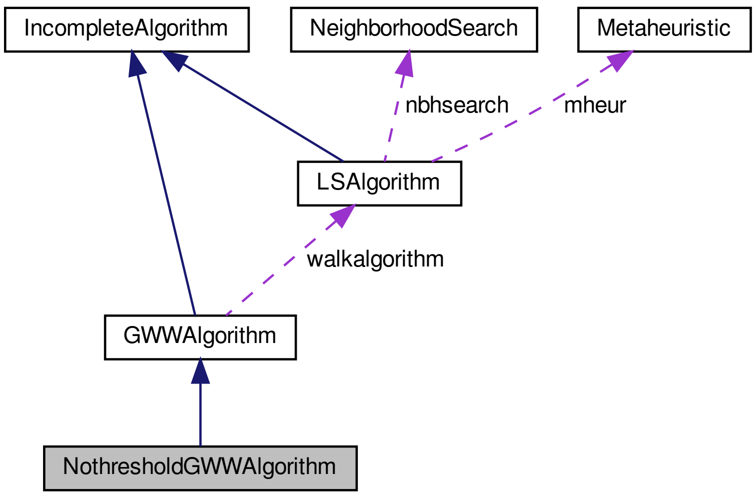
Referenced by CSProblem::min_conflict_value().

## 4.26 NothresholdGWWAlgorithm Class Reference

Inheritance diagram for NothresholdGWWAlgorithm:



Collaboration diagram for NothresholdGWWAlgorithm:



## Public Attributes

- int nbkilled

## Additional Inherited Members

### 4.26.1 Detailed Description

GWW without threshold management : 2 parameters : number of particles redistributed at each iteration , number of iterations

### 4.26.2 Member Data Documentation

#### 4.26.2.1 nbkilled

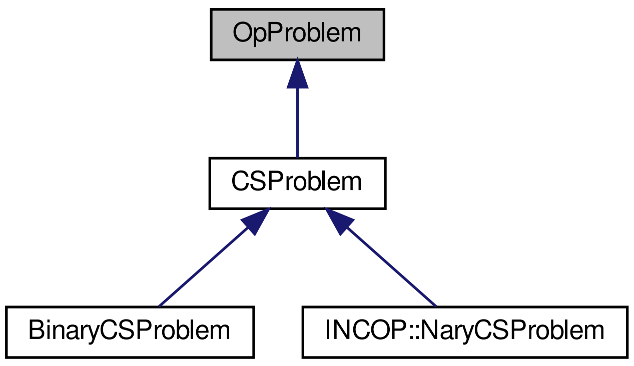
```
int NothresholdGWWAlgorithm::nbkilled
```

number of particles to be regrouped at each iteration

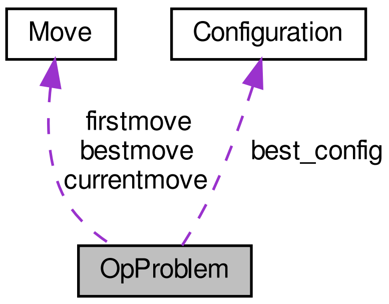
Referenced by GWWAlgorithm::regrouping(), SimulatedAnnealing::SimulatedAnnealing(), and GWWAlgorithm↩ ::thresholdcomputedelta().

## 4.27 OpProblem Class Reference

Inheritance diagram for OpProblem:



Collaboration diagram for OpProblem:



#### Public Member Functions

- virtual void move_execution (Configuration ∗configuration, Move ∗move)
- virtual void incr_update_conflicts (IncrCSPConfiguration ∗configuration, Move ∗move)
- virtual void fullincr_update_conflicts (FullincrCSPConfiguration ∗configuration, Move ∗move)
- virtual void allocate_moves ()
- virtual Move ∗ create_move ()
- virtual void adjust_parameters (Configuration ∗configuration, int &maxneighbors, int &minneighbors)
- virtual void next_move (Configuration ∗configuration, Move ∗move, NeighborhoodSearch ∗nbhs)
- virtual void random_configuration (Configuration ∗configuration)
- virtual void best_config_analysis ()

- virtual void best_config_write ()
- virtual void best_config_verification ()
- virtual void init_population (Configuration ∗∗population, int populationsize)
- virtual Configuration ∗ create_configuration ()
- virtual Long compute_conflict (Configuration ∗configuration, int var, int val)
- virtual Long config_evaluation (Configuration ∗configuration)
- virtual Long move_evaluation (Configuration ∗configuration, Move ∗move)
- virtual int index2value (int index, int var)
- virtual int value2index (int value, int var)
- virtual void compute_var_conflict (Configuration ∗configuration)

**Public Attributes**

- Configuration ∗ best_config
- int nbvar
- int domainsize
- Long lower_bound
- Move ∗ currentmove
- Move ∗ firstmove
- Move ∗ bestmove

### 4.27.1 Detailed Description

Root class of Optimization Problems (minimization)

### 4.27.2 Member Function Documentation

#### 4.27.2.1 adjust_parameters()

```
virtual void OpProblem::adjust_parameters (
              Configuration * configuration,
              int & maxneighbors,
              int & minneighbors )  [virtual]
```

adjustment of the neighborhood parameters (when the size of the actual neighborhood is greater than maxneighbors)

Referenced by DynamicNeighborhoodSearch::dynamicmaxneighbors().

#### 4.27.2.2 allocate_moves()

```
void OpProblem::allocate_moves ( )  [virtual]
```

creation of 3 Move objects (currentmove,bestmove,firstmove)

Referenced by INCOP::NaryCSProblem::create_configuration().

#### 4.27.2.3 best_config_analysis()

```
virtual void OpProblem::best_config_analysis ( )  [virtual]
```

analysis of the best configuration

#### 4.27.2.4 best_config_verification()

```
void OpProblem::best_config_verification ( )  [virtual]
```

verification of the best solution (its cost is recomputed)

#### 4.27.2.5 best_config_write()

```
virtual void OpProblem::best_config_write ( )  [virtual]
```

writing the best solution

#### 4.27.2.6 compute_conflict()

```
virtual Long OpProblem::compute_conflict (
             Configuration * configuration,
             int var,
             int val )  [virtual]
```

computation of the participation of (var,val) to the configuration evaluation

Reimplemented in INCOP::NaryCSProblem.

Referenced by Configuration::get_conflicts_problem(), and Configuration::update_conflicts().

#### 4.27.2.7 compute_var_conflict()

```
virtual void OpProblem::compute_var_conflict (
             Configuration * configuration )  [virtual]
```

compute the variables participating to a conflict in the configuration

Reimplemented in CSProblem.

Referenced by LSAlgorithm::configurationmove().

#### 4.27.2.8 config_evaluation()

```
virtual Long OpProblem::config_evaluation (
             Configuration * configuration )  [virtual]
```

evaluation of a configuration

Reimplemented in INCOP::NaryCSProblem.

#### 4.27.2.9 create_configuration()

```
virtual Configuration* OpProblem::create_configuration ( )  [virtual]
```

create a configuration (the exact class depends on the problem and must defined in subclasses)

Reimplemented in INCOP::NaryCSProblem.

#### 4.27.2.10 create_move()

```
virtual Move* OpProblem::create_move ( )  [virtual]
```

creation of 1 Move object (the class of the Move depends on the problem) : this method is implemented in subclasses

#### 4.27.2.11 fullincr_update_conflicts()

```
virtual void OpProblem::fullincr_update_conflicts (
            FullincrCSPConfiguration * configuration,
            Move * move )  [virtual]
```

update of the conflict data structure (case FullincrCSPConfiguration)

Referenced by FullincrCSPConfiguration::get_conflicts().

#### 4.27.2.12 incr_update_conflicts()

```
virtual void OpProblem::incr_update_conflicts (
            IncrCSPConfiguration * configuration,
            Move * move )  [virtual]
```

update of the conflict data structure (case IncrCSPConfiguration)

Referenced by Configuration::update_conflicts().

#### 4.27.2.13 index2value()

```
virtual int OpProblem::index2value (
            int index,
            int var )  [virtual]
```

valueindex in the domain to value

#### 4.27.2.14 init_population()

```
virtual void OpProblem::init_population (
            Configuration ** population,
            int populationsize )  [virtual]
```

initialization of the population of size populationsize

#### 4.27.2.15 move_evaluation()

```
virtual Long OpProblem::move_evaluation (
            Configuration * configuration,
            Move * move )  [virtual]
```

evaluation of a configuration if the move is done

#### 4.27.2.16 move_execution()

```
void OpProblem::move_execution (
            Configuration * configuration,
            Move * move )  [virtual]
```

move execution (modification of the current configuration)

References Configuration::config, Configuration::get_conflicts_problem(), and Configuration::valuation.

Referenced by LSAlgorithm::configurationmove().

#### 4.27.2.17 next_move()

```
virtual void OpProblem::next_move (
            Configuration * configuration,
            Move * move,
            NeighborhoodSearch * nbhs )  [virtual]
```

next move to be tested (implemented in subclasses)

Referenced by LSAlgorithm::configurationmove().

#### 4.27.2.18 random_configuration()

```
virtual void OpProblem::random_configuration (
            Configuration * configuration )  [virtual]
```

random assignment of the variables of a configuration

#### 4.27.2.19 value2index()

```
virtual int OpProblem::value2index (
            int value,
            int var )  [virtual]
```

valueindex of value in its domain

Referenced by Configuration::update_conflicts().

### 4.27.3 Member Data Documentation

#### 4.27.3.1 best_config

```
Configuration* OpProblem::best_config
```

the best configuration found

Referenced by LSAlgorithm::configurationmove(), INCOP::NaryCSProblem::create_configuration(), and GWW↩
Algorithm::populationkeepbest().

#### 4.27.3.2 bestmove

```
Move* OpProblem::bestmove
```

the best move found in the neighborhood

Referenced by LSAlgorithm::configurationmove().

#### 4.27.3.3 currentmove

```
Move* OpProblem::currentmove
```

the current move being tested

Referenced by LSAlgorithm::configurationmove().

#### 4.27.3.4 domainsize

```
int OpProblem::domainsize
```

maximum domain size

Referenced by INCOP::NaryCSProblem::create_configuration(), and DynamicNeighborhoodSearch::dynamicmaxneighbors().

#### 4.27.3.5 firstmove

```
Move* OpProblem::firstmove
```

the first feasible move tried in the neighborhood

Referenced by LSAlgorithm::configurationmove().

#### 4.27.3.6 lower_bound

```
Long OpProblem::lower_bound
```

given lower bound , is used as a stop condition when it is reached

Referenced by INCOP::NaryCSProblem::create_configuration(), LSAlgorithm::isfeasible(), GWWAlgorithm↩ ::populationrandomwalk(), GWWAlgorithm::randomwalk(), and GWWAlgorithm::run().

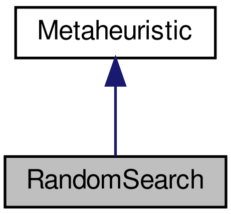
#### 4.27.3.7 nbvar

```
int OpProblem::nbvar
```

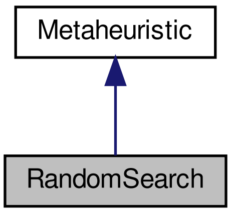
the number of variables

## 4.28 RandomSearch Class Reference

Inheritance diagram for RandomSearch:



Collaboration diagram for RandomSearch:



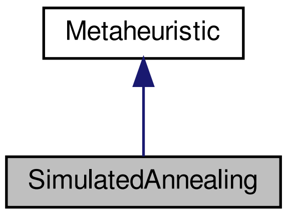
### Additional Inherited Members

#### 4.28.1 Detailed Description

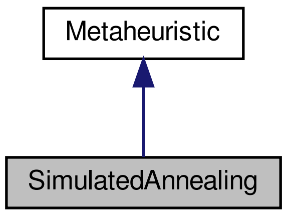
Random walk : every feasible neighbor is accepted

## 4.29 SimulatedAnnealing Class Reference

Inheritance diagram for SimulatedAnnealing:



Collaboration diagram for SimulatedAnnealing:



### Public Member Functions

- SimulatedAnnealing (double initialtemperature, int walklength)
- int acceptance (Move ∗move, Configuration ∗config)
- void executebeforemove (Move ∗move, Configuration ∗configuration, OpProblem ∗problem)

### Public Attributes

- double inittemperature
- double delta
- double temperature

#### 4.29.1 Detailed Description

Simulated Annealing : linear temperature descent from inittemperature to 0

#### 4.29.2 Constructor & Destructor Documentation

##### 4.29.2.1 SimulatedAnnealing()

```
SimulatedAnnealing::SimulatedAnnealing (
             double initialtemperature,
             int walklength )
```

Constructor : 2 parameters : initial temperature and walk length : the fixed temperature decrement is computed.

References GWWAlgorithm::elitism, GWWAlgorithm::lastmovedescent, GWWAlgorithm::nbiteration, Nothreshold↩
GWWAlgorithm::nbkilled, GWWAlgorithm::nomovestop, GWWAlgorithm::populationsize, GWWAlgorithm↩
::regrouptest, and GWWAlgorithm::walkalgorithm.

#### 4.29.3 Member Function Documentation

##### 4.29.3.1 acceptance()

```
int SimulatedAnnealing::acceptance (
             Move * move,
             Configuration * config )  [virtual]
```

Acceptance function of the temperature : classical simulated annealing formula for accepting a bad move : probability = exp (-temperature/evaluationdelta)

Reimplemented from Metaheuristic.

References Configuration::valuation.

##### 4.29.3.2 executebeforemove()

```
void SimulatedAnnealing::executebeforemove (
             Move * move,
             Configuration * configuration,
             OpProblem * problem )  [virtual]
```

the temperature is lowered by delta

Reimplemented from Metaheuristic.

### 4.29.4 Member Data Documentation

#### 4.29.4.1 delta

```
double SimulatedAnnealing::delta
```

constant step for lowering the temperature

#### 4.29.4.2 inittemperature

```
double SimulatedAnnealing::inittemperature
```

initial temperature

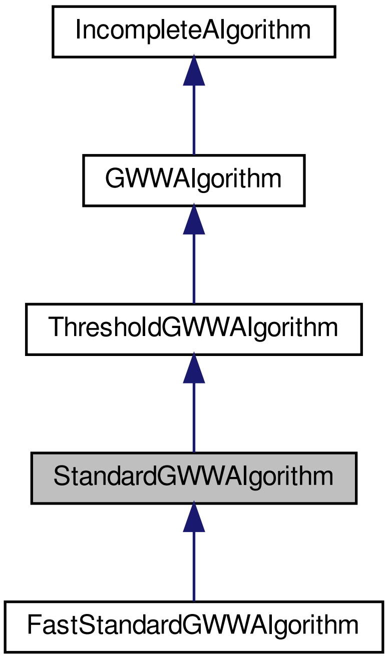
#### 4.29.4.3 temperature

```
double SimulatedAnnealing::temperature
```

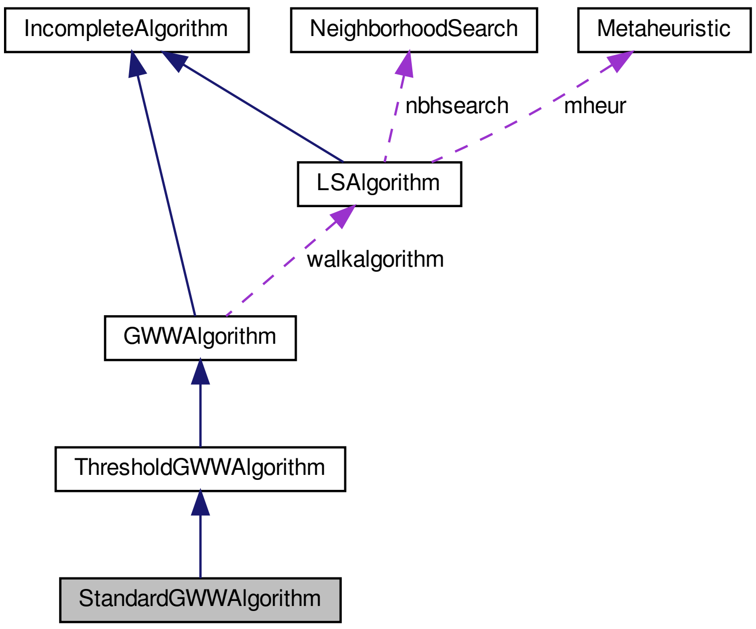
current temperature

## 4.30 StandardGWWAlgorithm Class Reference

Inheritance diagram for StandardGWWAlgorithm:



Collaboration diagram for StandardGWWAlgorithm:



#### Public Attributes

- double thresholddescent
- Long thresholdmin

#### Additional Inherited Members

### 4.30.1 Detailed Description

Standard GWW : threshold descent with a fixed rate

### 4.30.2 Member Data Documentation

#### 4.30.2.1 thresholddescent

```
double StandardGWWAlgorithm::thresholddescent
```

threshold descent constant rate

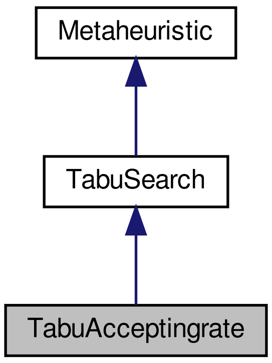
#### 4.30.2.2 thresholdmin

```
Long StandardGWWAlgorithm::thresholdmin
```

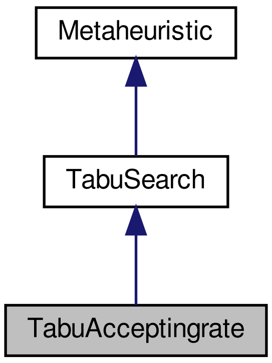
minimum of the threshold (corresponds generally to a known lowerbound)

## 4.31 TabuAcceptingrate Class Reference

Inheritance diagram for TabuAcceptingrate:



Collaboration diagram for TabuAcceptingrate:



### Public Member Functions

- int acceptance (Move ∗move, Configuration ∗config)

#### Public Attributes

- float Pd
- float P0

### 4.31.1 Detailed Description

Special Tabu search with complementary acceptance condition depending on the move direction

### 4.31.2 Member Function Documentation

#### 4.31.2.1 acceptance()

```
int TabuAcceptingrate::acceptance (
            Move * move,
            Configuration * config )  [virtual]
```

Acceptance condition : non tabu and probabilities depending on the move direction

Reimplemented from TabuSearch.

References Configuration::valuation.

### 4.31.3 Member Data Documentation

#### 4.31.3.1 P0

```
float TabuAcceptingrate::P0
```

probability of acceptance of a move with same cost

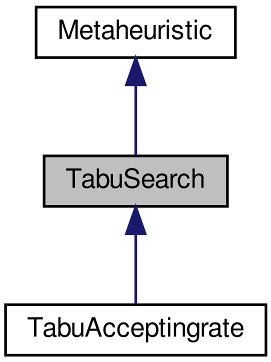
#### 4.31.3.2 Pd

```
float TabuAcceptingrate::Pd
```

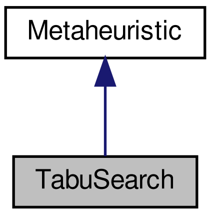
probability of acceptance of a worsening move

## 4.32 TabuSearch Class Reference

Inheritance diagram for TabuSearch:



Collaboration diagram for TabuSearch:



### Public Member Functions

- int acceptance (Move ∗move, Configuration ∗config)
- int nontabumove (Move ∗move)
- void executebeforemove (Move ∗move, Configuration ∗configuration, OpProblem ∗problem)
- void reinit (OpProblem ∗problem)

### Public Attributes

- int tabulength
- list< Move ∗ > move_list

#### 4.32.1 Detailed Description

Walk with using a tabu list : this list of moves is implemented by a list<Move∗> structure , the actual class of the moves depend on the problems

#### 4.32.2 Member Function Documentation

##### 4.32.2.1 acceptance()

```
int TabuSearch::acceptance (
            Move ∗ move,
            Configuration ∗ config )  [virtual]
```

acceptance of a move : not in the tabulist (the aspiration criterion of a best is in the configurationmove algorithm)

Reimplemented from Metaheuristic.

Reimplemented in TabuAcceptingrate.

References Configuration::valuation.

##### 4.32.2.2 executebeforemove()

```
void TabuSearch::executebeforemove (
            Move ∗ move,
            Configuration ∗ configuration,
            OpProblem ∗ problem )  [virtual]
```

updating of the tabulist which is managed as a FIFO of maximum length tabulength

Reimplemented from Metaheuristic.

References Move::computetabumove().

##### 4.32.2.3 nontabumove()

```
int TabuSearch::nontabumove (
            Move ∗ move )
```

test of non presence in the tabulist (use of eqmove method)

##### 4.32.2.4 reinit()

```
void TabuSearch::reinit (
            OpProblem ∗ problem )  [virtual]
```

the tabu list is cleared

Reimplemented from Metaheuristic.

### 4.32.3 Member Data Documentation

#### 4.32.3.1 move_list

```
list<Move*> TabuSearch::move_list
```

tabu list : implemented FIFO

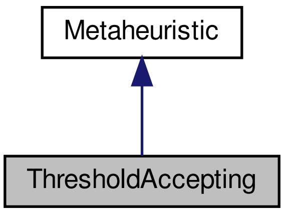
#### 4.32.3.2 tabulength

```
int TabuSearch::tabulength
```

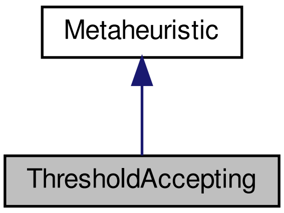
maximum length of the tabulist

## 4.33 ThresholdAccepting Class Reference

Inheritance diagram for ThresholdAccepting:



Collaboration diagram for ThresholdAccepting:



### Public Member Functions

- ThresholdAccepting (double maxthreshold, int walklength)
- int acceptance (Move ∗move, Configuration ∗config)
- void executebeforemove (Move ∗move, Configuration ∗configuration, OpProblem ∗problem)
- void reinit (OpProblem ∗problem)

### Public Attributes

- double thresholdinit
- double delta
- double thresholdaccept

## 4.33.1 Detailed Description

Threshold accepting Metaheuristics : a move must no deteriorate the evaluation more than the current threshod : the threshold goes down linearly from thresholdinit to 0

## 4.33.2 Constructor & Destructor Documentation

### 4.33.2.1 ThresholdAccepting()

```
ThresholdAccepting::ThresholdAccepting (
             double maxthreshold,
             int walklength )
```

constructor : two arguments : maxthreshold the initial threshold and walklength , it computes a constant step for lowering the threshold

## 4.33.3 Member Function Documentation

### 4.33.3.1 acceptance()

```
int ThresholdAccepting::acceptance (
             Move ∗ move,
             Configuration ∗ config )  [virtual]
```

acceptance condition : being under or at the threshold

Reimplemented from Metaheuristic.

References Configuration::valuation.

#### 4.33.3.2 executebeforemove()

```
void ThresholdAccepting::executebeforemove (
            Move * move,
            Configuration * configuration,
            OpProblem * problem ) [virtual]
```

the threshold is lowered by delta

Reimplemented from Metaheuristic.

#### 4.33.3.3 reinit()

```
void ThresholdAccepting::reinit (
            OpProblem * problem ) [virtual]
```

the threshold is initialized at thresholdinit

Reimplemented from Metaheuristic.

### 4.33.4 Member Data Documentation

#### 4.33.4.1 delta

```
double ThresholdAccepting::delta
```

constant step to lower the threshold

#### 4.33.4.2 thresholdaccept

```
double ThresholdAccepting::thresholdaccept
```

current value of the threshold

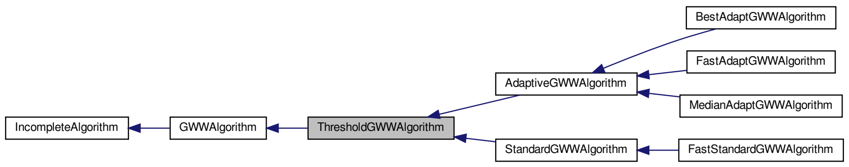
#### 4.33.4.3 thresholdinit

```
double ThresholdAccepting::thresholdinit
```

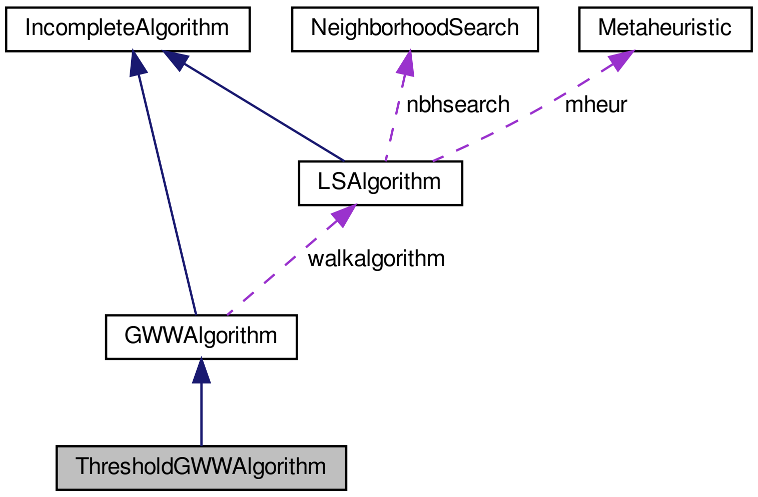
initial threshold

## 4.34 ThresholdGWWAlgorithm Class Reference

Inheritance diagram for ThresholdGWWAlgorithm:



Collaboration diagram for ThresholdGWWAlgorithm:



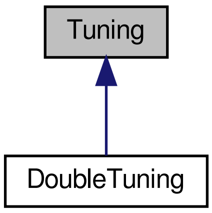
### Additional Inherited Members

### 4.34.1 Detailed Description

Abstract class : GWW managing a threshold

## 4.35 Tuning Class Reference

Inheritance diagram for Tuning:



### 4.35.1 Detailed Description

Automatic tuning of a local search algorithm with one parameter

## 4.36 WeightedCSP Class Reference

### Public Member 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 ∗ 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*
- virtual Cost finiteUb () const =0

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

  *updates infinite costs in all cost functions accordingly to the problem global lower and upper bounds*
- virtual bool enumerated (int varIndex) const =0

  *true if the variable has an enumerated domain*
- virtual string getName (int varIndex) const =0
- 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*
- virtual unsigned int getDomainSize (int varIndex) const =0

  *current domain size*
- virtual bool getEnumDomain (int varIndex, Value ∗array)=0

  *gets current domain values in an array*
- virtual bool getEnumDomainAndCost (int varIndex, ValueCost ∗array)=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 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)=0

  *assigns a set of variables at once and propagates (used by Local Search methods such as Large Neighborhood Search)*
- 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 (int varIndex)=0

  *initialize weighted degree heuristic*
- 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*
- 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 int getMaxDomainSize () const =0

  *maximum initial 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.*
- 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, Value ∗d, int dsize)=0

  *create an enumerated variable with its domain values*

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

  *create an interval variable with its domain bounds*
- virtual void postUnary (int xIndex, vector< Cost > &costs)=0
- virtual void postNaryConstraintTuple (int ctrindex, Value ∗tuple, int arity, Cost cost)=0
- virtual int postUnary (int xIndex, Value ∗d, int dsize, Cost penalty)=0
- virtual int postGlobalConstraint (int ∗scopeIndex, int arity, const string &gcname, istream &file, int ∗constrcounter=NULL, bool mult=true)=0
- virtual int postWAmong (int ∗scopeIndex, int arity, const string &semantics, const string &propagator, Cost baseCost, const vector< Value > &values, int lb, int ub)=0

  *post a soft among cost function*
- virtual void postWAmong (int ∗scopeIndex, int arity, string semantics, Cost baseCost, Value ∗values, int nb←Values, int lb, int ub)=0
- virtual void postWVarAmong (int ∗scopeIndex, int arity, string semantics, Cost baseCost, Value ∗values, int nbValues, 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 (int ∗scopeIndex, int arity, const string &semantics, const string &propagator, Cost baseCost, int nbStates, const vector< WeightedObj< int > > &initial_States, const vector< WeightedObj< int > > &accepting_States, const vector< DFATransition > &Wtransitions)=0

  *post a soft or weighted regular cost function*
- 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 (int ∗scopeIndex, int arity, const string &semantics, const string &propagator, Cost baseCost)=0

  *post a soft alldifferent cost function*
- 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< BoundedObj< Value > > &values)=0

  *post a soft global cardinality cost function*
- virtual void postWGcc (int ∗scopeIndex, int arity, string semantics, Cost baseCost, Value ∗values, int nb←Values, 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)*
- 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*
- virtual int postMST (int ∗scopeIndex, int arity, const string &semantics, const string &propagator, Cost base←Cost)=0

  *post a Spanning Tree hard constraint*
- 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)*
- virtual void postWSum (int ∗scopeIndex, int arity, string semantics, Cost baseCost, string comparator, int rightRes)=0

  *post a soft linear constraint with unit coefficients*
- 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)*
- virtual vector< vector< int > > ∗ getListSuccessors ()=0

  *generating additional variables vector created when berge decomposition are included in the WCSP*
- 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_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`*)*
- 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 const vector< Value > & getSolution (Cost ∗cost_ptr=NULL)=0

  *returns current best solution and its 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 (ostream &os)=0

  *prints current best solution*
- virtual void printSolution (FILE ∗f)=0

  *prints current best solution*
- 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*

#### Static Public Member Functions

- static WeightedCSP ∗ makeWeightedCSP (Cost upperBound, void ∗solver=NULL)

  *Weighted CSP factory.*

### 4.36.1 Detailed Description

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::make↩ EnumeratedVariable)
Cost functions are referenced by their lexicographic index number (as returned by *eg* WeightedCSP::post↩ BinaryConstraint)

### 4.36.2 Member Function Documentation

#### 4.36.2.1 assignLS()

```
virtual void WeightedCSP::assignLS (
            vector< int > & varIndexes,
            vector< Value > & newValues ) [pure virtual]
```

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 |

#### 4.36.2.2 cartProd()

```
virtual void WeightedCSP::cartProd (
            BigInteger & cartesianProduct ) [pure virtual]
```

Cartesian product of current domain sizes.

**Parameters**

| | |
|---|---|
| *cartesianProduct* | result obtained by the GNU Multiple Precision Arithmetic Library GMP |

#### 4.36.2.3 dump()

```
virtual void WeightedCSP::dump (
             ostream & os,
             bool original = true )  [pure virtual]
```

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 |

#### 4.36.2.4 finiteUb()

```
virtual Cost WeightedCSP::finiteUb ( ) const  [pure virtual]
```

computes the worst-case assignment finite cost (sum of maximum finite cost over all cost functions plus one)

**Returns**

the worst-case assignment finite cost

**Warning**

current problem should be completely loaded and propagated before calling this function

#### 4.36.2.5 getName()

```
virtual string WeightedCSP::getName (
             int varIndex ) const  [pure virtual]
```

**Note**

by default, variables names are integers, starting at zero

#### 4.36.2.6 getValue()

```
virtual Value WeightedCSP::getValue (
             int varIndex ) const  [pure virtual]
```

current assigned value

**Warning**

undefined if not assigned yet

#### 4.36.2.7 increaseLb()

```
virtual void WeightedCSP::increaseLb (
             Cost addLb )  [pure virtual]
```

increases problem lower bound thanks to *eg* soft local consistencies

**Parameters**

| | |
|---|---|
| *addLb* | increment value to be **added** to the problem lower bound |

#### 4.36.2.8 postGlobalConstraint()

```
virtual int WeightedCSP::postGlobalConstraint (
            int * scopeIndex,
            int arity,
            const string & gcname,
            istream & file,
            int * constrcounter = NULL,
            bool mult = true )  [pure virtual]
```

#### 4.36.2.9 postMaxWeight()

```
virtual int WeightedCSP::postMaxWeight (
            int * scopeIndex,
            int arity,
            const string & semantics,
            const string & propagator,
            Cost baseCost,
            const vector< WeightedVarValPair > weightFunction )  [pure virtual]
```

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. |

#### 4.36.2.10 postMST()

```
virtual int WeightedCSP::postMST (
            int * scopeIndex,
            int arity,
            const string & semantics,
            const string & propagator,
            Cost baseCost )  [pure virtual]
```

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) |

#### 4.36.2.11 postNaryConstraintTuple()

```
virtual void WeightedCSP::postNaryConstraintTuple (
             int ctrindex,
             Value * tuple,
             int arity,
             Cost cost )  [pure virtual]
```

**Warning**

must call WeightedCSP::postNaryConstraintEnd after giving cost tuples

#### 4.36.2.12 postUnary() [1/2]

```
virtual void WeightedCSP::postUnary (
             int xIndex,
             vector< Cost > & costs )  [pure virtual]
```

#### 4.36.2.13 postUnary() [2/2]

```
virtual int WeightedCSP::postUnary (
             int xIndex,
             Value * d,
             int dsize,
             Cost penalty )  [pure virtual]
```

**Warning**

must call WeightedCSP::sortConstraints after all cost functions have been posted (see WeightedCSP::sort↩Constraints)

#### 4.36.2.14 postWAllDiff() [1/2]

```
virtual int WeightedCSP::postWAllDiff (
             int * scopeIndex,
             int arity,
             const string & semantics,
             const string & propagator,
             Cost baseCost )  [pure virtual]
```

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 |

#### 4.36.2.15 postWAllDiff() [2/2]

```
virtual void WeightedCSP::postWAllDiff (
            int * scopeIndex,
            int arity,
            string semantics,
            Cost baseCost )  [pure virtual]
```

#### 4.36.2.16 postWAmong() [1/2]

```
virtual int WeightedCSP::postWAmong (
            int * scopeIndex,
            int arity,
            const string & semantics,
            const string & propagator,
            Cost baseCost,
            const vector< Value > & values,
            int lb,
            int ub )  [pure virtual]
```

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* |

#### 4.36.2.17 postWAmong() [2/2]

```
virtual void WeightedCSP::postWAmong (
             int * scopeIndex,
             int arity,
             string semantics,
             Cost baseCost,
             Value * values,
             int nbValues,
             int lb,
             int ub )  [pure virtual]
```

#### 4.36.2.18 postWGcc() [1/2]

```
virtual int WeightedCSP::postWGcc (
             int * scopeIndex,
             int arity,
             const string & semantics,
             const string & propagator,
             Cost baseCost,
             const vector< BoundedObj< Value > > & values )  [pure virtual]
```

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 BoundedObj, specifying the lower and upper bounds of each value, restricting the number of variables can be assigned to them |

#### 4.36.2.19 postWGcc() [2/2]

```
virtual void WeightedCSP::postWGcc (
             int * scopeIndex,
             int arity,
             string semantics,
             Cost baseCost,
             Value * values,
             int nbValues,
             int * lb,
             int * ub )  [pure virtual]
```

#### 4.36.2.20 postWGrammarCNF()

```
virtual int WeightedCSP::postWGrammarCNF (
             int * scopeIndex,
             int arity,
             const string & semantics,
             const string & propagator,
             Cost baseCost,
             int nbSymbols,
             int startSymbol,
             const vector< CFGProductionRule > WRuleToTerminal )  [pure virtual]
```

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: <br>• 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. <br>• if *order* in *CFGProductionRule* is set to 1, it is classified as A -> BC, where A,B,C the index of the nonterminal symbols. <br>• 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. <br>• 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. <br>• if *order* in *CFGProductionRule* is set to values greater than 3, it is ignored. |

#### 4.36.2.21 postWOverlap()

```
virtual void WeightedCSP::postWOverlap (
             int * scopeIndex,
             int arity,
             string semantics,
             Cost baseCost,
             string comparator,
             int rightRes )  [pure virtual]
```

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 |

#### 4.36.2.22 postWRegular() [1/2]

```
virtual int WeightedCSP::postWRegular (
            int * scopeIndex,
            int arity,
            const string & semantics,
            const string & propagator,
            Cost baseCost,
            int nbStates,
            const vector< WeightedObj< int > > & initial_States,
            const vector< WeightedObj< int > > & accepting_States,
            const vector< DFATransition > & Wtransitions ) [pure virtual]
```

post a soft or 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 WeightedObj specifying the starting states with weight |
| *accepting_States* | a vector of WeightedObj specifying the final states |
| *Wtransitions* | a vector of (weighted) transitions |

**Warning**

Weights are ignored in the current implementation of DAG and flow-based propagators

#### 4.36.2.23 postWRegular() [2/2]

```
virtual void WeightedCSP::postWRegular (
             int * scopeIndex,
             int arity,
             int nbStates,
             vector< pair< int, Cost > > initial_States,
             vector< pair< int, Cost > > accepting_States,
             int ** Wtransitions,
             vector< Cost > transitionsCosts )  [pure virtual]
```

#### 4.36.2.24 postWSame() [1/2]

```
virtual int WeightedCSP::postWSame (
             int * scopeIndexG1,
             int arityG1,
             int * scopeIndexG2,
             int arityG2,
             const string & semantics,
             const string & propagator,
             Cost baseCost )  [pure virtual]
```

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. |

#### 4.36.2.25 postWSame() [2/2]

```
virtual void WeightedCSP::postWSame (
             int * scopeIndex,
             int arity,
             string semantics,
             Cost baseCost )  [pure virtual]
```

#### 4.36.2.26 postWSum()

```
virtual void WeightedCSP::postWSum (
            int * scopeIndex,
            int arity,
            string semantics,
            Cost baseCost,
            string comparator,
            int rightRes )  [pure virtual]
```

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 |

#### 4.36.2.27 read_uai2008()

```
virtual void WeightedCSP::read_uai2008 (
            const char * fileName )  [pure virtual]
```

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)

#### 4.36.2.28 setInfiniteCost()

```
virtual void WeightedCSP::setInfiniteCost ( )  [pure virtual]
```

updates infinite costs in all cost functions accordingly to the problem global lower and upper bounds

**Warning**

to be used in preprocessing only

#### 4.36.2.29 sortConstraints()

```
virtual void WeightedCSP::sortConstraints ( ) [pure virtual]
```

sorts the list of cost functions associated to each variable based on smallest problem variable indexes

**Warning**

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

**Note**

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

## 4.37 WeightedCSPSolver Class Reference

### Public Member 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 netwrok 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 global-name="")=0

  *create a random WCSP, see WeightedCSP::read_random*
- virtual bool solve ()=0

  *simplifies and solves to optimality the problem*
- virtual Cost narycsp (string cmd, vector< Value > &solution)=0

  *solves the current problem using INCOP local search solver by Bertrand Neveu*
- 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 $\forall i, posx[i] \leq posy[i]$)*
- virtual void dump_wcsp (const char ∗fileName, bool original=true)=0

  *output current problem in a file*
- virtual void read_solution (const char ∗fileName, bool updateValueHeuristic=true)=0

  *read a solution from a file*
- virtual void parse_solution (const char ∗certificate)=0

  *read a solution from a string (see ToulBar2 option -x)*
- virtual Cost getSolution (vector< Value > &solution)=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)*

#### Static Public Member Functions

- static WeightedCSPSolver ∗ makeWeightedCSPSolver (Cost initUpperBound)

  *WeightedCSP Solver factory.*

### 4.37.1 Detailed Description

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)

### 4.37.2 Member Function Documentation

#### 4.37.2.1 dump_wcsp()

```
virtual void WeightedCSPSolver::dump_wcsp (
             const char * fileName,
             bool original = true )  [pure virtual]
```

output current problem in a file

**See also**

WeightedCSP::dump

Referenced by makeWeightedCSPSolver().

#### 4.37.2.2 narycsp()

```
virtual Cost WeightedCSPSolver::narycsp (
             string cmd,
             vector< Value > & solution )  [pure virtual]
```

solves the current problem using INCOP local search solver by Bertrand Neveu

**Returns**

best solution cost found

**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) |

**Warning**

cannot solve problems with global cost functions

**Note**

side-effects: updates current problem upper bound and propagates, best solution saved (using WCSP::set↩ BestValue)

Referenced by makeWeightedCSPSolver().

#### 4.37.2.3 solve()

```
virtual bool WeightedCSPSolver::solve ( )  [pure virtual]
```

simplifies and solves to optimality the problem

**Returns**

false if there is no solution found

**Warning**

after solving, the current problem has been modified by various preprocessing techniques
DO NOT READ VALUES OF ASSIGNED VARIABLES USING WeightedCSP::getValue (temporally wrong assignments due to variable elimination in preprocessing) BUT USE WeightedCSPSolver::getSolution INS↩ TEAD

Referenced by makeWeightedCSPSolver().

#### 4.37.2.4 solve_symmax2sat()

```
virtual bool WeightedCSPSolver::solve_symmax2sat (
            int n,
            int m,
            int * posx,
            int * posy,
            double * cost,
            int * sol )  [pure virtual]
```

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 $\forall i, posx[i] \leq posy[i]$)

**Note**

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

**See also**

::solvesymmax2sat_ for Fortran call

# Index