toulbar2 Reference Manual

Release 1.0.0

INRAE

Feb 28, 2022

CONTENTS

Cost Function Network Solver	toulbar2
Copyright	toulbar2 team
Source	https://github.com/toulbar2/toulbar2

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`.

CHAPTER

ONE

TOULBAR2

1.1 Exact optimization for cost function networks and additive graphical models

master: cpd:

1.2 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)

toulbar2 is now also able to collaborate with ML code that can learn an additive graphical model (with constraints) from data (see the associated paper, slides and video where it is shown how it can learn user preferences or how to play the Sudoku without knowing the rules). The current CFN learning code is available on GitHub.

1.3 Installation from binaries

You can install toulbar2 directly using the package manager in Debian and Debian derived Linux distributions (Ubuntu, Mint,. . .):

```
sudo apt-get update
sudo apt-get install toulbar2 toulbar2-doc
```

For the most recent binary or the Python API, compile from source.

1.4 Python interface

An alpha-release Python interface can be tested through pip on Linux and MacOS:

```
python3 -m pip install --upgrade pip
python3 -m pip install pytoulbar2
```

The first line is only useful for Linux distributions that ship "old" versions of pip.

Commands for compiling the Python API on Linux/MacOS with cmake (Python module in lib/*/pytb2.cpython*.so):

```
mkdir build
cd build
cmake -DPYTB2=ON ..
make
```

Move the cpython library and the experimental pytoulbar2.py python class wrapper in the folder of the python script that does "import pytoulbar2".

1.5 Download

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

```
git clone --branch 1.1.1 https://github.com/toulbar2/toulbar2.git
```

1.6 Installation from sources

Compilation requires git, cmake and a C++-11 capable compiler (in C++11 mode).

Required library:

- libgmp-dev

Recommended libraries (default use):

- libboost-graph-dev
- libboost-iostreams-dev
- libboost-serialization-dev
- zlib1g-dev
- liblzma-dev

Optional libraries:

- libxml2-dev
- libopenmpi-dev
- libboost-mpi-dev
- libjemalloc-dev

On MacOS, run ./misc/script/MacOS-requirements-install.sh to install the recommended libraries.

Commands for compiling toulbar2 on Linux/MacOS with cmake (binary in build/bin/*/toulbar2):

```
mkdir build
cd build
cmake ..
make
```

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

```
bash
cd src
echo '#define Toulbar_VERSION "1.1.0"' > ToulbarVersion.hpp
g++ -o toulbar2 -I. tb2*.cpp applis/*.cpp core/*.cpp globals/*.cpp incop/*.cpp search/*.
↪cpp utils/*.cpp vns/*.cpp ToulbarVersion.cpp -std=c++11 -O3 -DNDEBUG \
 -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -DWCSPFORMATONLY -lboost_graph -lboost_
↪iostreams -lboost_serialization -lgmp -lz -llzma -static
```

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

```
bash
cd src
echo '#define Toulbar_VERSION "1.1.0"' > ToulbarVersion.hpp
mpicxx -o toulbar2 -I. tb2*.cpp applis/*.cpp core/*.cpp globals/*.cpp incop/*.cpp search/
↪*.cpp utils/*.cpp vns/*.cpp ToulbarVersion.cpp -std=c++11 -O3 -DNDEBUG \
 -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -DOPENMPI -DWCSPFORMATONLY -lboost_graph -
↪lboost_iostreams -lboost_serialization -lboost_mpi -lgmp -lz -llzma
```

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

1.7 Authors

toulbar2 was originally developped by Toulouse (INRAE 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

1.8 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

1.9 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
- SCP branching (CPD branch): Fast search algorithms for Computational Protein Design. S Traoré, K Roberts, D Allouche, B Donald, I André, T Schiex, S Barbe. Journal of Computational Chemistry, 2016
- Guaranteed Free Energy computation (weighted model counting): Guaranteed Weighted Counting for Affinity Computation: Beyond Determinism and Structure C Viricel, D Simoncini, S Barbe, T Schiex. In Proc. of CP-16, Toulouse, France, 2016
- 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 ; Variable neighborhood search for graphical model energy minimization A Ouali, D Allouche, S de Givry, S Loudni, Y Lebbah, L Loukil, and P Boizumault Artificial Intelligence, 278(103194), 2020
- 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
- Greedy sequence of diverse solutions (div): Guaranteed diversity & quality for the Weighted CSP M Ruffini, J Vucinic, S de Givry, G Katsirelos, S Barbe, and T Schiex In Proc. of ICTAI-19, pages 18-25, Portland, OR, USA, 2019
- VAC integrality based variable heuristics and initial upper-bounding (vacint and rasps): Relaxation-Aware Heuristics for Exact Optimization in Graphical Models F Trösser, S de Givry and G Katsirelos In Proc. of CPAIOR-20, Vienna, Austria, 2020

CHAPTER

TWO

MODULE DOCUMENTATION

2.1 Weighted Constraint Satisfaction Problem file format (wcsp)

group **wcspformat**

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>
```

- 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>
...
```

(continues on next page)

(continued from previous page)

```
<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>
```

- 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)
  - knapsack *capacity* (*weight*)* to express a reverse knapsack constraint (i.e., a linear constraint on 0/1 variables with >= operator) with capacity and weights are positive or negative integer coefficients (use negative numbers to express a linear constraint with <= operator)
  - knapsackp *capacity* (*nb_values* (*value weight*)*)* to express a reverse knapsack constraint with for each variable the list of values to select the item in the knapsack with their corresponding weight
- 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
  - wdiverse *distance* (*value*)* to express a hard diversity constraint using a dual encoding such that there is a given minimum Hamming distance to a given variable assignment
  - whdiverse *distance* (*value*)* to express a hard diversity constraint using a hidden encoding such that there is a given minimum Hamming distance to a given variable assignment
  - wtdiverse *distance* (*value*)* to express a hard diversity constraint using a ternary encoding such that there is a given minimum Hamming distance to a given variable assignment

    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);

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
```

- knapsack constraint ($2 * x0 + 3 * x1 + 4 * x2 + 5 * x3 >= 10$) on four Boolean 0/1 variables:

```
4 0 1 2 3 -1 knapsack 10 2 3 4 5
```

- knapsackp constraint ($2 * (x0 = 0) + 3 * (x1 = 1) + 4 * (x2 = 2) + 5 * (x3 = 0 \vee x3 = 1) >= 10$) on four {0,1,2}-domain variables:

```
4 0 1 2 3 -1 knapsackp 10 1 0 2 1 1 3 1 2 4 2 0 5 1 5
```

- 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 ==
```

- wdiverse({x0,x1,x2,x3}) hard constraint on four variables with minimum Hamming distance of 2 to the value assignment (1,1,0,0):

```
4 0 1 2 3 -1 wdiverse 2 1 1 0 0
```

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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

Note: domain values range from zero to *size-1*

Note: a negative domain size is interpreted as a variable with an interval domain in $[0, -size - 1]$

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.

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

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

Warning: The decomposition of wsum and wvarsum may use an exponential size (sum of domain sizes).

Warning: *list_size1* and *list_size2* must be equal in *ssame*.

Warning: Cost functions defined in intention cannot be shared.

2.2 Variable and cost function modeling

group **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)

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

Note: 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)

Note: 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: 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.

Warning: Each global cost function may have less than three propagators implemented

Warning: Current implementation of toulbar2 has limited solving facilities for monolithic global cost functions (no BTD-like methods nor variable elimination)

Warning: 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)

Warning: Before modeling the problem using make and post, call ::tb2init method to initialize toulbar2 global variables

Warning: After modeling the problem using make and post, call *WeightedCSP::sortConstraints* method to initialize correctly the model before solving it

2.3 Solving cost function networks

group **solving**

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>
```

(continues on next page)

(continued from previous page)

```
#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
    initCosts(); // last check for compatibility issues between ToulBar2 options␣
↪and Cost data-type
    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

2.4 Output messages, verbosity options and debugging

group **`verbosity`**

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)

Note: `toulbar2` will output an help message giving available options if run without any parameters

2.5 Preprocessing techniques

group **preprocessing**

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)

2.6 Variable and value search ordering heuristics

group **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.

See also:

Last Conflict Based Reasoning . Christophe Lecoutre, Lakhdar Sais, Sebastien Tabary, Vincent Vidal. Proc. of ECAI 2006, pages 133-137. Trentino, Italy, 2006.

2.7 Soft arc consistency and problem reformulation

group **softac**

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.

See also:

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

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.

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.

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.

2.8 Virtual Arc Consistency enforcing

group `VAC`

2.9 NC bucket sort

group `ncbucket`

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.

2.10 Variable elimination

group `varelim`

- *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)

Note: 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

Warning: 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.

2.11 Propagation loop

group **`propagation`**

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

2.12 Backtrack management

group **`backtrack`**

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::storeInt for storable integer values ::StoreInt (number of non assigned variables in nary cost functions, etc)
- Store::storeCost for storable costs ::StoreCost (inside cost functions, etc)
- Store::storeDomain for enumerated domains (to manage holes inside domains)
- Store::storeIndexList for integer lists (to manage edge connections in global cost functions)
- 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: Current storable data management is not multi-threading safe! (Store is a static virtual class relying on StoreBasic<T> static members)