toulbar2 Reference Manual

Release 1.0.0

INRAE

Apr 21, 2022

CONTENTS

1 Introduction **1**

2 Exact optimization for cost function networks and additive graphical models **2**
- 2.1 What is toulbar2? . . . 2
- 2.2 Installation from binaries . . . 2
- 2.3 Python interface . . . 3
- 2.4 Download . . . 3
- 2.5 Installation from sources . . . 3

3 Modules **5**
- 3.1 Variable and cost function modeling . . . 5
- 3.2 Solving cost function networks . . . 6
- 3.3 Output messages, verbosity options and debugging . . . 11
- 3.4 Preprocessing techniques . . . 12
- 3.5 Variable and value search ordering heuristics . . . 12
- 3.6 Soft arc consistency and problem reformulation . . . 12
- 3.7 Virtual Arc Consistency enforcing . . . 13
- 3.8 NC bucket sort . . . 13
- 3.9 Variable elimination . . . 13
- 3.10 Propagation loop . . . 14
- 3.11 Backtrack management . . . 15

4 Libraries **16**

CHAPTER

ONE

INTRODUCTION

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

TWO

EXACT OPTIMIZATION FOR COST FUNCTION NETWORKS AND ADDITIVE GRAPHICAL MODELS

2.1 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.

2.2 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.

2.3 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".

2.4 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
```

2.5 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).

 toulbar2 is currently maintained by Simon de Givry, INRAE - MIAT, Toulouse, France (simon.de-givry@inrae.fr)

CHAPTER

THREE

MODULES

3.1 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)

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

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 : 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

3.2 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>
#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]);
```

(continues on next page)

(continued from previous page)

```
    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 : another code example in ./src/toulbar2test.cpp

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

toulbar2test.cpp

toulbar2test.cpp

```
/**
 * Test toulbar2 API
 */

#include "toulbar2lib.hpp"

#include "core/tb2wcsp.hpp"
#include "vns/tb2vnsutils.hpp"
#include "vns/tb2dgvns.hpp"
#ifdef OPENMPI
#include "vns/tb2cpdgvns.hpp"
#include "vns/tb2rpdgvns.hpp"
```

(continues on next page)

(continued from previous page)

```
#endif
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// INCOP default command line option
const string Incop_cmd = "0 1 3 idwa 100000 cv v 0 200 1 0 0";

int main(int argc, char* argv[])
{
#ifdef OPENMPI
    mpi::environment env; // equivalent to MPI_Init via the constructor and␣
↪MPI_finalize via the destructor
    mpi::communicator world;
#endif

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

#ifdef OPENMPI
    if (world.rank() == WeightedCSPSolver::MASTER)
        ToulBar2::verbose = -1; // change to 0 or higher values to see more␣
↪trace information
    else
        ToulBar2::verbose = -1;
#else
    ToulBar2::verbose = -1; // change to 0 or higher values to see more trace␣
↪information
#endif

    // uncomment if Virtual Arc Consistency (equivalent to Augmented DAG␣
↪algorithm) enable
    //        ToulBar2::vac = 1; // option -A
    //        ToulBar2::vacValueHeuristic = true; // option -V
    // uncomment if partial Limited Discrepancy Search enable
    //        ToulBar2::lds = 1;  // option -l=1
    // uncomment if INCOP local search enable
    //        ToulBar2::incop_cmd = Incop_cmd; // option -i
    // uncomment the following lines if variable neighborhood search enable
    //ToulBar2::lds = 4;
    //ToulBar2::restart = 10000;
    //#ifdef OPENMPI
    //     if (world.size() > 1) {
    //             ToulBar2::searchMethod = RPDGVNS;
    //             ToulBar2::vnsParallel = true;
    //             ToulBar2::vnsNeighborVarHeur = MASTERCLUSTERRAND;
    //             ToulBar2::vnsParallelSync = false;
    //     } else {
    //             ToulBar2::searchMethod = DGVNS;
    //             ToulBar2::vnsNeighborVarHeur = CLUSTERRAND;
    //     }
```

(continues on next page)

(continued from previous page)

```
    //#else
    //        ToulBar2::searchMethod = DGVNS;
    //        ToulBar2::vnsNeighborVarHeur = CLUSTERRAND;
    //**or**
    //  ToulBar2::searchMethod = VNS;
    //  ToulBar2::vnsNeighborVarHeur = RANDOMVAR;
    //#endif

    // create a problem with three 0/1 variables
    initCosts(); // last check for compatibility issues between ToulBar2␣
↪options and Cost data-type
    WeightedCSPSolver* solver = WeightedCSPSolver::makeWeightedCSPSolver(MAX_
↪COST);
    int x = solver->getWCSP()->makeEnumeratedVariable("x", 0, 1); // note that␣
↪for efficiency issue, I assume domain values start at zero (otherwise remove␣
↪flag -DWCSPFORMATONLY in Makefile)
    int y = solver->getWCSP()->makeEnumeratedVariable("y", 0, 1);
    int z = solver->getWCSP()->makeEnumeratedVariable("z", 0, 1);

    // add random unary cost functions on each variable
    mysrand(getpid());
    {
        vector<Cost> costs(2, 0);
        costs[0] = randomCost(0, 100);
        costs[1] = randomCost(0, 100);
        solver->getWCSP()->postUnary(x, costs);
        costs[0] = randomCost(0, 100);
        costs[1] = randomCost(0, 100);
        solver->getWCSP()->postUnary(y, costs);
        costs[0] = randomCost(0, 100);
        costs[1] = randomCost(0, 100);
        solver->getWCSP()->postUnary(z, costs);
    }

    // add binary cost functions (Ising) on each pair of variables
    {
        vector<Cost> costs;
        for (unsigned int i = 0; i < 2; i++) {
            for (unsigned int j = 0; j < 2; j++) {
                costs.push_back((solver->getWCSP()->toValue(x, i) == solver->
↪getWCSP()->toValue(y, j)) ? 0 : 30); // penalizes by a cost=30 if variables␣
↪are assigned to different values
            }
        }
        solver->getWCSP()->postBinaryConstraint(x, y, costs);
        solver->getWCSP()->postBinaryConstraint(x, z, costs);
        solver->getWCSP()->postBinaryConstraint(y, z, costs);
    }

    // add a ternary hard constraint (x+y=z)
    {
        vector<Cost> costs;
```

(continues on next page)

(continued from previous page)

```
        for (unsigned int i = 0; i < 2; i++) {
            for (unsigned int j = 0; j < 2; j++) {
                for (unsigned int k = 0; k < 2; k++) {
                    costs.push_back((solver->getWCSP()->toValue(x, i) + solver-
↪>getWCSP()->toValue(y, j) == solver->getWCSP()->toValue(z, k)) ? 0 : MAX_
↪COST);
                }
            }
        }
        solver->getWCSP()->postTernaryConstraint(x, y, z, costs);
    }

    solver->getWCSP()->sortConstraints(); // must be done before the search

    //        int verbose = ToulBar2::verbose;
    //        ToulBar2::verbose = 5;  // high verbosity to see the cost␣
↪functions
    //        solver->getWCSP()->print(cout);
    //        ToulBar2::verbose = verbose;

    //tb2checkOptions();
    if (solver->solve()) {
#ifdef OPENMPI
        if (world.rank() == WeightedCSPSolver::MASTER) {
#endif
            // show optimal solution
            vector<Value> sol;
            Cost optimum = solver->getSolution(sol);
            cout << "Optimum=" << optimum << endl;
            cout << "Solution: x=" << sol[x] << " ,y=" << sol[y] << " ,z=" <<␣
↪sol[z] << endl;
#ifdef OPENMPI
        }
#endif
    } else {
#ifdef OPENMPI
        if (world.rank() == WeightedCSPSolver::MASTER) {
#endif
            cout << "No solution found!" << endl;
#ifdef OPENMPI
        }
#endif
    }
    // cout << "Problem lower bound: " << solver->getWCSP()->getLb() << endl; /
↪/ initial problem lower bound possibly enhanced by value removals at the␣
↪root during search

    delete solver;
    return 0;
}

/* Local Variables: */
```

(continues on next page)

(continued from previous page)

```
/* c-basic-offset: 4 */
/* tab-width: 4 */
/* indent-tabs-mode: nil */
/* c-default-style: "k&r" */
/* End: */
```

3.3 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

3.4 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)

3.5 Variable and value search ordering heuristics

group **heuristics**

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

See : *Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers* . Emir Demirovic, Geoffrey Chu, and Peter Stuckey. Proc. of CP-18, pages 99–108. Lille, France, 2018.

3.6 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.

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 : *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 : *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 Virtual Arc Consistency enforcing

group **VAC**

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

3.8 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.

3.9 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.

3.10 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

3.11 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)

CHAPTER

FOUR

LIBRARIES

- C++ Library : see “C++ Library of toulbar2” document.
- Python Library : see “Python Library of toulbar2” document.