1 What is Toulbar2

Toulbar2 is an exact black box discrete optimization solver targeted at solving cost function networks (CFN), thus solving the so-called "weighted Constraint Satisfaction Problem" or WCSP. Cost function networks can be simply described by a set of discrete variables each having a specific finite domain and a set of integer cost functions, each involving some of the variables. The WCSP is to find an assignment of all variables such that the sum of all cost functions is minimum and lest than a given upper bound often denoted as k or $\top$. Functions can be typically specified by sparse or full tables but also more concisely as specific functions called "global cost functions".

Using on the fly translation, Toulbar2 can also directly solve optimization problems on other graphical models such as Maximum probability Explanation (MPE) on Bayesian networks [4], and Maximum A Posteriori (MAP) on Markov random field [4]. It can also read partial weighted MaxSAT problems, Quadratic Pseudo Boolean problems (MAXCUT) as well as Linkage `.pre` pedigree files for genotyping error detection and correction.

Toulbar2 is exact. It will only report an optimal solution when it has both identified the solution and proved its optimality. Because it relies only on integer operations, addition and subtraction, it does not suffer from rounding errors. In the general case, the WCSP, MPE/BN, MAP/MRF, PWMaxSAT, QPBO or MAXCUT being all NP-hard problems and thus Toulbar2 may take exponential time to prove optimality. This is however a worst-case behavior and Toulbar2 has been shown to be able to solve to optimality problems with half a million non boolean variables defining a search space as large as $2^{829,440}$. It may also fail to solve in reasonable time problems with a search space smaller than 2^{264}.

Toulbar2 provides and uses by default an "anytime" algorithm [1] that tries to quickly provide good solutions together with an upper bound on the gap between the cost of each solution and the (unknown) optimal cost. Thus, even if it is unable to prove optimality, it will bound the quality of the solution provided.

Beyond the service of providing optimal solutions, Toulbar2 can also exhaustively enumerate solutions below a cost threshold and perform guaranteed approximate weighted counting of solutions. For stochastic graphical models, this means that Toulbar2 will compute the partition function (or the normalizing constant Z). These problems being #P-complete, Toulbar2 runtimes can quickly increase on such problems.

2 How do I install it ?

Toulbar2 is an open source solver distributed under the Gnu Public Library (GPL) as a set of C++ sources managed with git at `http://mulcyber.toulouse.inra.fr/projects/toulbar2`. If you want to use a released version, then you can download there binary archives as a shell archive, an rpm or a debian package that should be easy to use on most Linux systems as well as an autoinstalling executable for Windows.

If you want to compile it yourself, you will need a modern C++ compiler, CMake, Gnu MP Bignum library, a recent version of boost libraries and optionally the jemalloc memory management library. You can then clone TOULBAR2 on your machine and compile it by executing:

```
git clone http://mulcyber.toulouse.inra.fr/anonscm/git/toulbar2/toulbar2.git
cd toulbar/toulbar2
mkdir build
cd build
cmake ..
make
```

Finally, TOULBAR2 should be soon available in the debian-science section of the unstable/sid debian version. It should therefore be directly installable using:

```
sudo apt-get install toulbar2
```

If you want to try TOULBAR2 on crafted, random, or real problems, please look for benchmarks in the Cost Function Library.

3 Using it as a black box

Using TOULBAR2 is just a matter of having a properly formatted input file describing the cost function network, graphical model, PWMaxSAT, PBO or Linkage `.pre` file and executing:

```
toulbar2 [option parameters] <file>
```

and TOULBAR2 will start solving the optimization problem described in its file argument. By default, the extension of the file (either `.wcsp`, `.wcnf`, `.cnf`, `.qpbo`, `.uai`, `.LG`, `.pre` or `.bep`) is used to determine the nature of the file (see section 5). There is no specific order for the options or problem file. TOULBAR2 comes with decently optimized default option parameters. It is however often possible to set it up for different target than pure optimization or tune it for faster action using specific command line options.

4 Command line options

If you just execute:

```
toulbar2
```

TOULBAR2 will give you its (long) list of optional parameter which we now describe in more detail. If you don't known much about Constraint and Cost Function Programming, section 8 describes some of the inner working of TOULBAR2 to help you tune it to your requirements.

To deactivate a default command line option, just use the command-line option followed by "`:`". For example:

```
toulbar2 -dee: <file>
```

will disable the default Dead End Elimination [3] (aka Soft Neighborhood Substitutability) preprocessing.

4.1 General control

-a finds all solutions (or count the number of zero-cost satisfiable solutions in conjunction with BTD)

-D approximate satisfiable solution count with BTD

-logz computes log of probability of evidence (i.e. log partition function or log(Z) or PR task) for graphical models only (problem file extension .uai)

-timer=[integer] give a CPU time limit in seconds. TOULBAR2 will stop after the specified CPU time has been consumed. The time limit is a CPU user time limit, not wall clock time limit.

4.2 Preprocessing

-nopre deactivates all preprocessing options (equivalent to -e: -p: -t: -f: -dec: -n: -mst: -dee:)

-p=[integer] preprocessing only: general variable elimination of degree less than or equal to the given value (default value is -1)

-t=[integer] preprocessing only: simulates restricted path consistency by adding ternary cost functions on triangles of binary cost functions within a given maximum space limit (in MB)

-f=[integer] preprocessing only: variable elimination of functional (f=1) (resp. bijective (f=2)) variables (default value is 1)

-dec preprocessing only: pairwise decomposition of cost functions with arity ¿=3 into smaller arity cost functions (default option)

-n=[integer] preprocessing only: projects n-ary cost functions on all binary cost functions if n is lower than the given value (default value is 10)

-mst maximum spanning tree DAC ordering

-M=[integer] apply the Min Sum Diffusion algorithm (default is inactivated, with a number of iterations of 0). See [2].

4.3 Initial upper bounding

-l=[integer] limited discrepancy search, use a negative value to stop the search after the given absolute number of discrepancies has been explored (discrepancy bound = 4 by default)

-L=[integer] randomized (quasi-random variable ordering) search with restart (maximum number of nodes = 10000 by default)

-i=["string"] initial upper-bound found by INCOP local search solver.The string parameter is optional, using "0 1 3 idwa 100000 cv v 0 200 1 0 0" by default with the following meaning: stoppinglowerbound randomseed nbiterations method nbmoves neighborhoodchoice neighborhoodchoice2 minnbneighbors maxnbneighbors neighborhoodchoice3 autotuning tracemode.

-x=[(,i=a)*] assigns variable of index i to value a (multiple assignments are separated by a comma and no space) (without any argument, a complete assignment – used as initial upper bound and as value heuristic – read from default file "sol" or given as input filename with ".sol" extension)

4.4 Tree search algorithm and tree decomposition selection

-hbfs=[integer] hybrid best-first search, restarting from the root after a given number of backtracks (default value is 10000)

-open=[integer] hybrid best-first search limit on the number of open nodes (default value is -1)

-B=[integer] (0) DFBB, (1) BTD, (2) RDS-BTD, (3) RDS-BTD with path decomposition instead of tree decomposition (default value is 0)

-O=[filename] reads a variable elimination order from a file in order to build a tree decomposition (if BTD-like and/or variable elimination methods are used) and also a compatible DAC ordering

-O=[negative integer] build a tree decomposition (if BTD-like and/or variable elimination methods are used) and also a compatible DAC ordering using

- (-1) maximum cardinality search ordering,
- (-2) minimum degree ordering,
- (-3) minimum fill-in ordering,
- (-4) maximum spanning tree ordering (see -mst),
- (-5) reverse Cuthill-Mckee ordering,
- (-6) approximate minimum degree ordering

If not specified, then use the variable order in which variables appear in the problem file.

-j=[integer] splits large clusters into a chain of smaller embedded clusters with a number of proper variables less than this number (use options ”-B=3 -j=1 -svo -k=1” for pure RDS, use value 0 for no splitting) (default value is 0).

-r=[integer] limit on maximum cluster separator size (merge cluster with its father otherwise, use a negative value for no limit) (default value is -1)

-X=[integer] limit on minimum number of proper variables in a cluster (merge cluster with its father otherwise, use a zero for no limit) (default value is 0)

-E merges leaf clusters with their fathers if small local treewidth (in conjunction with option ”-e”)

-R=[integer] choice for a specific root cluster number

-I=[integer] choice for solving only a particular rooted cluster subtree (with RDS-BTD only)

4.5 Node processing & bounding options

-e=[integer] performs “on the fly” variable elimination of variable with small degree (less than or equal specified value, default is 3 creating a maximum of ternary cost functions). See [5].

-k=[integer] soft local consistency level (NC with Strong NIC for global cost functions=0, (G)AC=1, D(G)AC=2, FD(G)AC=3, (weak) ED(G)AC=4) (default value is 4)

-A=[integer] enforces VAC at each search node with a search depth less than a given value (default value is 0)

-dee=[integer] restricted dead-end elimination (value pruning by dominance rule from EAC value (dee¿=1 and dee¡=3)) and soft neighborhood substitutability (in preprocessing (dee=2 or dee=4) or during search (dee=3)) (default value is 1)

-o ensures optimal worst-case time complexity of DAC and EAC (can be slower in practice)

4.6 Branching, variable and value ordering

-svo searches using a static variable ordering heuristic. The variable order value used will be the same order as the DAC order.

-b searches using only binary branching instead of the default that uses binary branching for interval domains and small domains and dichotomic branching for large enumerated domains.

-c searches using binary branching with last conflict backjumping variable ordering heuristic and dichotomic branching for large domains (default option).

-q=[integer] use weighted degree variable ordering heuristic if the number of cost functions is less than the given value (default value is 10000).

-var=[integer] searches by branching only on the first [given value] decision variables, assuming the remaining variables are intermediate variables that will be completely assigned by the decision variables (use a zero if all variables are decision variables, default value is 0)

-m=[integer] use a variable ordering heuristic that selects first variables such that the sum of the mean (m=1) or median (m=2) cost of all incident cost functions is maximum (in conjunction with weighted degree heuristic -q) (default value is 0: unused).

-d=[integer] searches using dichotomic branching. The default d=1 splits domains in the middle of domain range while d=2 splits domains in the middle of sorted unary costs.

-sortd sorts domains based on increasing unary costs (warning! works only for binary WCSPs).

4.7 Console output

-help shows the default help message that Toulbar2 prints when it gets no argument.

-v=[integer] sets the verbosity level (default 0).

-s shows each solution found during search. The solution is printed on one line, giving the value (integer) of each variable successively in increasing order.

4.8 File output

-w=[filename] writes last solution found in the specified filename (or "sol" if no parameter is given). The current directory is used is a relative path is used.

-z=[filename] saves problem in wcsp format in filename (or "problem.wcsp" if no parameter is given) writes also the graphviz dot file and the degree distribution of the input problem

-z=[integer] 1: saves original instance (by default), 2: saves after preprocessing

-Z=[integer] debug mode (save problem at each node if verbosity option -v=num ¿= 1 and -Z=num ¿=3)

-x=[(,i=a)*] assigns variable of index i to value a (multiple assignments are separated by a comma and no space) (without any argument, a complete assignment – used as initial upper bound and as value heuristic – read from default file "sol" or given as input filename with ".sol" extension)

4.9 Probability representation and numerical control

-precision=[integer] probability/real precision is a conversion factor (a power of ten) for representing fixed point numbers (default value is 7)

-epsilon=[float] approximation factor for computing the partition function (default value is 1000 representing $\varepsilon = \frac{1}{1000}$)

4.10 Random problem generation

-random=[bench profile] bench profile must be specified as follows.

- n and d are respectively the number of variable and the maximum domain size of the random problem.

  bin-n-d-t1-p2-seed

  - t1 is the tightness in percentage % of random binary cost functions
  - p2 is the num of binary cost functions to include
  - the seed parameter is optional

  binsub-n-d-t1-p2-p3-seed binary random & submodular cost functions

  - t1 is the tightness in percentage % of random cost functions
  - p2 is the num of binary cost functions to include
  - p3 is the percentage % of submodular cost functions among p2 cost functions (plus 10 permutations of two randomly-chosen values for each domain)

  tern-n-d-t1-p2-p3-seed

  - p3 is the num of ternary cost functions

  nary-n-d-t1-p2-p3...-pn-seed

  - pn is the num of n-ary cost functions

  salldiff-n-d-t1-p2-p3...-pn-seed

  - pn is the num of salldiff global cost functions (p2 and p3 still being used for the number of random binary and ternary cost functions)

5 Input File formats

Notice that by default `toulbar2` distinguishes file formats based on their extension.

5.1 wcsp format (.wcsp file extension)

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 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 hard 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
```

- 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
```

5.1.1 CPD final stanza

TODO

5.2 UAI LG formats (.uai .LG)

It is a simple text file format specified below to describe probabilistic graphical model instances. The format is a generalization of the Ergo file format initially developed by Noetic Systems Inc. for their Ergo software.

- **Structure**

  A file in the UAI format consists of the following two parts, in that order:

  `<Preamble>`

  `<Function tables>`

  The contents of each section (denoted < ... > above) are described in the following:

- **Preamble**

  The preamble starts with one line denoting the type of network. This will be either BAYES (if the network is a Bayesian network) or MARKOV (in case of a Markov network). This is followed by a line containing the number of variables. The next line specifies each variable's domain size, one at a time, separated by whitespace (note that this implies an order on the variables which will be used throughout the file).

  The fourth line contains only one integer, denoting the number of functions in the problem (conditional probability tables for Bayesian networks, general factors for Markov networks). Then, one function per line, the scope of each function is given as follows: The first integer in each line specifies the size of the function's scope, followed by the actual indexes of the variables in the scope. The order of this list is not restricted, except when specifying a conditional probability table (CPT) in a Bayesian network, where the child variable has to come last. Also note that variables are indexed starting with 0.

For instance, a general function over variables 0, 5 and 11 would have this entry:

```
3 0 5 11
```

A simple Markov network preamble with three variables and two functions might for instance look like this:

```
MARKOV
3
2 2 3
2
2 0 1
3 0 1 2
```

The first line denotes the Markov network, the second line tells us the problem consists of three variables, let's refer to them as X, Y, and Z. Their domain size is 2, 2, and 3 respectively (from the third line). Line four specifies that there are 2 functions. The scope of the first function is X,Y, while the second function is defined over X,Y,Z.

An example preamble for a Belief network over three variables (and therefore with three functions) might be:

```
BAYES
3
2 2 3
3
1 0
2 0 1
2 1 2
```

The first line signals a Bayesian network. This example has three variables, let's call them X, Y, and Z, with domain size 2, 2, and 3, respectively (from lines two and three). Line four says that there are 3 functions (CPTs in this case). The scope of the first function is given in line five as just X (the probability P(X)), the second one is defined over X and Y (this is (Y | X)). The third function, from line seven, is the CPT P(Z | Y). We can therefore deduce that the joint probability for this problem factors as P(X,Y,Z) = P(X).P(Y | X).P(Z | Y).

- **Function tables**

In this section each function is specified by giving its full table (i.e, specifying the function value for each tuple). The order of the functions is identical to the one in which they were introduced in the preamble.

For each function table, first the number of entries is given (this should be equal to the product of the domain sizes of the variables in the scope). Then, one by one, separated by whitespace, the values for each assignment to the variables in the function's scope are enumerated. Tuples are implicitly assumed in ascending order, with the last variable in the scope as the 'least significant'.

To illustrate, we continue with our Bayesian network example from above, let's assume the following conditional probability tables:

X	$P(X)$
0	0.436
1	0.564

X	Y	$P(Y\|X)$
0	0	0.128
0	1	0.872
1	0	0.920
1	1	0.080

Y	Z	$P(Z\|Y)$
0	0	0.210
0	1	0.333
0	2	0.457
1	0	0.811
1	1	0.000
1	2	0.189

The correspoding function tables in the file would then look like this:

```
2
 0.436 0.564

4
 0.128 0.872
 0.920 0.080

6
 0.210 0.333 0.457
 0.811 0.000 0.189
```

(Note that line breaks and empty lines are effectively just whitespace, exactly like plain spaces " ". They are used here to improve readability.)

- **Summary**

  To sum up, a problem file consists of 2 sections: the preamble and the full the function tables, the names and the labels

  For our Markov network example above, the full file could be:

```
MARKOV
3
2 2 3
2
2 0 1
3 0 1 2

4
 4.000 2.400
 1.000 0.000

12
 2.2500 3.2500 3.7500
 0.0000 0.0000 10.0000
 1.8750 4.0000 3.3330
 2.0000 2.0000 3.4000
```

  Here is the full Bayesian network example from above:

```
BAYES
3
2 2 3
3
1 0
2 0 1
2 1 2

2
 0.436 0.564

4
 0.128 0.872
 0.920 0.080

6
 0.210 0.333 0.457
 0.811 0.000 0.189
```

- **Expressing evidence**

  Evidence is specified in a separate file. This file has the same name as the original problems file but an added .evid extension at the end. For instance, problem.uai will have evidence in problem.uai.evid.

  The file simply starts with a line specifying the number of evidence variables. This is followed by the pairs of variable and value indexes for each observed variable, one pair per line. The indexes correspond to the ones implied by the original problem file.

If, for our above example, we want to specify that variable Y has been observed as having its first value and Z with its second value, the file example.uai.evid would contain the following:

```
2
 1 0
 2 1
```

5.3 (Partial Weighted) MaxSAT format

Max-SAT input format (.cnf)

The input file format for Max-SAT will be in DIMACS format:

```
c
c comments Max-SAT
c
p cnf 3 4
1 -2 0
-1 2 -3 0
-3 2 0
1 3 0
```

- The file can start with comments, that is lines beginning with the character 'c'.
- Right after the comments, there is the line "p cnf nbvar nbclauses" indicating that the instance is in CNF format; nbvar is the number of variables appearing in the file; nbclauses is the exact number of clauses contained in the file.
- Then the clauses follow. Each clause is a sequence of distinct non-null numbers between -nbvar and nbvar ending with 0 on the same line. Positive numbers denote the corresponding variables. Negative numbers denote the negations of the corresponding variables.

Weighted Max-SAT input format (.wcnf)

In Weighted Max-SAT, the parameters line is "p wcnf nbvar nbclauses". The weights of each clause will be identified by the first integer in each clause line. The weight of each clause is an integer greater than or equal to 1.

Example of Weighted Max-SAT formula:

```
c
c comments Weighted Max-SAT
c
p wcnf 3 4
10 1 -2 0
3 -1 2 -3 0
8 -3 2 0
5 1 3 0
```

Partial Max-SAT input format (.wcnf)

In Partial Max-SAT, the parameters line is "p wcnf nbvar nbclauses top". We associate a weight with each clause, wich is the first integer in the clause. Weigths must be greater than or equal to 1. Hard clauses have weigth top and soft clauses have weigth 1. We assume that top is a weight always greater than the sum of the weights of violated soft clauses.

Example of Partial Max-SAT formula:

```
c
c comments Partial Max-SAT
c
p wcnf 4 5 15
15 1 -2 4 0
15 -1 -2 3 0
1 -2 -4 0
1 -3 2 0
1 1 3 0
```

Weigthed Partial Max-SAT input format (.wcnf)

In Weigthed Partial Max-SAT, the parameters line is "p wcnf nbvar nbclauses top". We associate a weight with each clause, wich is the first integer in the clause. Weigths must be greater than or equal to 1. Hard clauses have weigth top and soft clauses have a weigth smaller than top. We assume that top is a weight always greater than the sum of the weights of violated soft clauses.

Example of Weigthed Partial Max-SAT formula:

```
c
c comments Weigthed Partial Max-SAT
c
p wcnf 4 5 16
16 1 -2 4 0
16 -1 -2 3 0
8 -2 -4 0
4 -3 2 0
3 1 3 0
```

5.4 QPBO format (.qpbo)

In the quadratic pseudo-Boolean optimization (unconstrained quadratic programming) format, the goal is to minimize or maximize the quadratic function:

$$X' * W * X = \sum_{i=1}^{N} \sum_{j=1}^{N} W_{ij} * X_i * X_j$$

where W is a symmetric squared $N \times N$ matrix expressed by all its non-zero half ($i \leq j$) squared matrix coefficients, X is a vector of N binary variables with domain values in $\{0, 1\}$ or $\{1, -1\}$, and X' is the transposed vector of X.

Note that for two indices $i \neq j$, coefficient $W_{ij} = W_{ji}$ (symmetric matrix) and it appears twice in the previous sum. Note also that coefficients can be positive or negative and are real float numbers. They are converted to fixed-point real numbers by multiplying them by $10^{precision}$ (see option *-precision* to modify it, default value is 7). Infinite coefficients are forbidden.

Notice that depending on the sign of the number of variables in the first text line, the domain of all variables is either $\{0, 1\}$ or $\{1, -1\}$.

Warning! The encoding in Weighted CSP of variable domain $\{1, -1\}$ associates for each variable value the following index: value 1 has index 0 and value -1 has index 1 in the solutions found by toulbar2. The encoding of variable domain $\{0, 1\}$ is direct.

Qpbo is a file text format:

- First line contains the number of variables N and the number of non-zero coefficients M.

  If N is negative then domain values are in $\{1, -1\}$, otherwise $\{0, 1\}$. If M is negative then it will maximize the quadratic function, otherwise it will minimize it.

- Followed by $|M|$ lines where each text line contains three values separated by spaces: position index i (integer belonging to $[1, |N|]$), position index j (integer belonging to $[1, |N|]$), coefficient W_{ij} (float number) such that $i \leq j$ and $W_{ij} \neq 0$

5.5 Linkage format (.pre)

See `mendelsoft` companion software at `http://www.inra.fr/mia/T/MendelSoft` for pedigree correction. See also `https://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/HaplotypeInference` for haplotype inference in half-sib families.

6 Using it as a library

See `toulbar2` reference manual which describes the libtb2.so C++ library API.

7 Using it from Python/Numberjack

See `http://numberjack.ucc.ie`.

8 How does it work

TODO

References

[1] D Allouche, S de Givry, G Katsirelos, T Schiex, and M Zytnicki. Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP. In *Proc. of CP-15*, pages 12–28, Cork, Ireland, 2015.

[2] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7–8):449–478, 2010.

[3] S de Givry, S Prestwich, and B O'Sullivan. Dead-End Elimination for Weighted CSP. In *Proc. of CP-13*, pages 263–272, Uppsala, Sweden, 2013.

[4] D Koller and N Friedman. *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009.

[5] J. Larrosa. Boosting search with variable elimination. In *Principles and Practice of Constraint Programming - CP 2000*, volume 1894 of *LNCS*, pages 291–305, Singapore, September 2000.