toulbar2 Documentation

Release 1.0.0

INRAE

Feb 28, 2022

CONTENTS

Quick access : *Downloads* | *Command line arguments* | . . .

toulbar2 is an exact solver for cost function networks.

CHAPTER

ONE

PRESENTATION

1.1 About toulbar2

toulbar2 is an open-source C++ solver for cost function networks. It solves various combinatorial optimization problems.

The constraints and objective function are factorized in local functions on discrete variables. Each function returns a cost (a finite positive integer) for any assignment of its variables. Constraints are represented as functions with costs in $\{0, \infty\}$ where ∞ is a large integer representing forbidden assignments. toulbar2 looks for a non-forbidden assignment of all variables that minimizes the sum of all functions.

Its engine uses a hybrid best-first branch-and-bound algorithm exploiting soft arc consistencies. It incorporates a parallel variable neighborhood search method for better performances. See *Publications*.

toulbar2 won several competitions on Max-CSP (CPAI08) and probabilistic graphical models (UAI 2008, 2010, 2014 MAP task).

toulbar2 is now also able to collaborate with ML code that can learn an additive graphical model (with constraints) from data (see example at cfn-learn).

1.2 Authors

toulbar2 was originally developed by Toulouse (INRAE MIAT) and Barcelona (UPC, IIIA-CSIC) teams, hence the name of the solver. Additional global cost functions were provided by the *Chinese University of Hong Kong* and Caen University (GREYC). It also includes codes from Marseille University (LSIS, tree decomposition heuristics) and *Ecole des Ponts ParisTech* (CERMICS/LIGM, INCOP local search solver).

A Python interface is now available. Install it with “python3 -m pip install pytoulbar2”. See examples in *Tutorials* using pytoulbar2 module. An older version is part of Numberjack (Insight - University College Cork). A portfolio approach dedicated to UAI format instances is available here.

toulbar2 is currently maintained by Simon de Givry (simon.de-givry@inrae.fr) and hosted on GitHub.

1.3 Citations

- 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

1.4 Acknowledgements

toulbar2 has been partly funded by the French *Agence Nationale de la Recherche* (projects STAL-DEC-OPT from 2006 to 2008, ANR-10-BLA-0214 Ficolofo from 2011 to 2014, and ANR-16-CE40-0028 DemoGraph from 2017 to 2021) and a PHC PROCORE project number 28680VH (from 2013 to 2015). It is currently supported by ANITI ANR-19-P3IA-0004 (2019-).

1.5 License

CHAPTER

TWO

RESOURCES

2.1 Downloads

Latest release toulbar2 binaries

Linux 64bit | MacOs 64bit | Windows 64bit

For more *other resources*

2.2 Open-source code

github

2.3 Documentation

Main documentation : `toulbar2`

API Reference : `Class Diagram` | `toulbar2 API Reference` | `pytoulbar2 API Reference`

Some extracts : `User manual` | `Reference manual` | `Tutorials` | `WCSP format` | `CFN format`

CHAPTER THREE

DOCUMENTATION

3.1 Manuals

Downloads : `User manual` | `Reference manual` | `WCSP format` | `CFN format`

3.1.1 User Manual

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" [Schiex2016a].

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 [koller2009], and Maximum A Posteriori (MAP) on Markov random field [koller2009]. 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 [Katsirelos2015a] 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. It can also apply a variable neighborhood search algorithm exploiting a problem decomposition [Ouali2017]. This algorithm is complete (if enough CPU-time is given) and it can be run in parallel using OpenMPI.

Beyond the service of providing optimal solutions, toulbar2 can also find a greedy sequence of diverse solutions [Ruffini2019a] or 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.

By exploiting the new toulbar2 python interface, with incremental solving capabilities, it is possible to learn a CFN from data and to combine it with mandatory constraints [Schiex2020b]. See examples at https://forgemia.inra.fr/thomas.schiex/cfn-learn.

How do I install it ?

toulbar2 is an open source solver distributed under the MIT license as a set of C++ sources managed with git at http://github.com/toulbar2/toulbar2. If you want to use a released version, then you can download there source archives of a specific release that should be easy to compile on most Linux systems.

If you want to compile the latest sources 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 and OpenMPI libraries. You can then clone toulbar2 on your machine and compile it by executing:

```
git clone https://github.com/toulbar2/toulbar2.git
cd toulbar2
mkdir build
cd build
# ccmake ..
cmake ..
make
```

Finally, toulbar2 is 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 benchmark Section. Other benchmarks coming from various discrete optimization languages are available at Genotoul EvalGM [Hurley2016b].

How do I test it ?

Some problem examples are available in the directory **toulbar2/validation**. After compilation with cmake, it is possible to run a series of tests using:

```
make test
```

For debugging toulbar2 (compile with flag `CMAKE_BUILD_TYPE="Debug"`), more test examples are available at Cost Function Library. The following commands run toulbar2 (executable must be found on your system path) on every problems with a 1-hour time limit and compare their optimum with known optima (in .ub files).

```
cd toulbar2
git clone https://forgemia.inra.fr/thomas.schiex/cost-function-library.git
./misc/script/runall.sh ./cost-function-library/trunk/validation
```

Other tests on randomly generated problems can be done where optimal solutions are verified by using an older solver toolbar (executable must be found on your system path).

```
cd toulbar2
git clone https://forgemia.inra.fr/thomas.schiex/toolbar.git
cd toolbar/toolbar
make toolbar
cd ../..
./misc/script/rungenerate.sh
```

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 **.cfn**, **.cfn.gz**, **.cfn.xz**, **.wcsp**, **.wcsp.gz**, **.wcsp.xz**, **.wcnf**, **.wcnf.gz**, **.wcnf.xz**, **.cnf**, **.cnf.gz**, **.cnf.xz**, **.qpbo**, **.qpbo.gz**, **.qpbo.xz**, **.opb**, **.opb.gz**, **.opb.xz**, **.uai**, **.uai.gz**, **.uai.xz**, **.LG**, **.LG.gz**, **.LG.xz**, **.pre** or **.bep**) is used to determine the nature of the file (see *Input File formats*). 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.

Quick start

- Download a binary weighted constraint satisfaction problem (WCSP) file `example.wcsp.xz`. Solve it with default options:

  ```
  toulbar2 EXAMPLES/example.wcsp.xz
  ```

  ```
  ```

- Solve a WCSP using INCOP, a local search method [idwalk:cp04] applied just after preprocessing, in order to find a good upper bound before a complete search:

  ```
  toulbar2 EXAMPLES/example.wcsp.xz -i
  ```

  ```
  ```

- Solve a WCSP with an initial upper bound and save its (first) optimal solution in filename “example.sol”:

  ```
  toulbar2 EXAMPLES/example.wcsp.xz -ub=28 -w=example.sol
  ```

  ```
  ```

- … and see this saved “example.sol” file:

  ```
  cat example.sol
  # each value corresponds to one variable assignment in problem file order
  ```

```

```

- Download a larger WCSP file `scen06.wcsp.xz`. Solve it using a limited discrepancy search strategy [Ginsberg1995] with a VAC integrality-based variable ordering [Trosser2020a] in order to speed-up the search for finding good upper bounds first (by default, toulbar2 uses another diversification strategy based on hybrid best-first search [Katsirelos2015a]):

```
toulbar2 EXAMPLES/scen06.wcsp.xz -l -vacint
```

```

```

- Download a cluster decomposition file `scen06.dec` (each line corresponds to a cluster of variables, clusters may overlap). Solve the previous WCSP using a variable neighborhood search algorithm (UDGVNS) [Ouali2017] during 10 seconds:

```
toulbar2 EXAMPLES/scen06.wcsp.xz EXAMPLES/scen06.dec -vns -time=10
```

```

```

- Download another difficult instance `scen07.wcsp.xz`. Solve it using a variable neighborhood search algorithm (UDGVNS) with maximum cardinality search cluster decomposition and absorption [Ouali2017] during 5 seconds:

```
toulbar2 EXAMPLES/scen07.wcsp.xz -vns -O=-1 -E -time=5
```

```

```

- Download file `404.wcsp.xz`. Solve it using Depth-First Brand and Bound with Tree Decomposition and HBFS (BTD-HBFS) [Schiex2006a] based on a min-fill variable ordering:

```
toulbar2 EXAMPLES/404.wcsp.xz -O=-3 -B=1
```

```

```

- Solve the same problem using Russian Doll Search exploiting BTD [Sanchez2009a]:

```
toulbar2 EXAMPLES/404.wcsp.xz -O=-3 -B=2
```

```

```

- Solve another WCSP using the original Russian Doll Search method [Verfaillie1996] with static variable ordering (following problem file) and soft arc consistency:

```
toulbar2 EXAMPLES/505.wcsp.xz -B=3 -j=1 -svo -k=1
```

```

```

- Solve the same WCSP using a parallel variable neighborhood search algorithm (UPDGVNS) with min-fill cluster decomposition [Ouali2017] using 4 cores during 5 seconds:

```
mpirun -n 4 toulbar2 EXAMPLES/505.wcsp.xz -vns -O=-3 -time=5
```

- Download a cluster decomposition file `example.dec` (each line corresponds to a cluster of variables, clusters may overlap). Solve a WCSP using a variable neighborhood search algorithm (UDGVNS) with a given cluster decomposition:

```
toulbar2 EXAMPLES/example.wcsp.xz EXAMPLES/example.dec -vns
```

- Solve a WCSP using a parallel variable neighborhood search algorithm (UPDGVNS) with the same cluster decomposition:

```
mpirun -n 4 toulbar2 EXAMPLES/example.wcsp.xz EXAMPLES/example.dec -vns
```

- Download file `example.order`. Solve a WCSP using BTD-HBFS based on a given (min-fill) reverse variable elimination ordering:

```
toulbar2 EXAMPLES/example.wcsp.xz EXAMPLES/example.order -B=1
```

- Download file `example.cov`. Solve a WCSP using BTD-HBFS based on a given explicit (min-fill path-) tree-decomposition:

```
toulbar2 EXAMPLES/example.wcsp.xz EXAMPLES/example.cov -B=1
```

- Download a Markov Random Field (MRF) file `pedigree9.uai.xz` in UAI format. Solve it using bounded (of degree at most 8) variable elimination enhanced by cost function decomposition in preprocessing [Favier2011a] followed by BTD-HBFS exploiting only small-size (less than four variables) separators:

```
toulbar2 EXAMPLES/pedigree9.uai.xz -O=-3 -p=-8 -B=1 -r=4
```

- Download another MRF file `GeomSurf-7-gm256.uai.xz`. Solve it using Virtual Arc Consistency (VAC) in preprocessing [Cooper2008] and exploit a VAC-based value [Cooper2010a] and variable [Trosser2020a] ordering heuristics:

```
toulbar2 EXAMPLES/GeomSurf-7-gm256.uai.xz -A -V -vacint
```

- Download another MRF file `1CM1.uai.xz`. Solve it by applying first an initial upper bound probing, and secondly, use a modified variable ordering heuristic based on VAC-integrality during search [Trosser2020a]:

```
toulbar2 EXAMPLES/1CM1.uai.xz -A=1000 -vacint -rasps -vacthr
```

- Download a weighted Max-SAT file `brock200_4.clq.wcnf.xz` in wcnf format. Solve it using a modified variable ordering heuristic [Schiex2014a]:

```
toulbar2 EXAMPLES/brock200_4.clq.wcnf.xz -m=1
```

- Download another WCSP file `latin4.wcsp.xz`. Count the number of feasible solutions:

```
toulbar2 EXAMPLES/latin4.wcsp.xz -a
```

- Find a greedy sequence of at most 20 diverse solutions with Hamming distance greater than 12 between any pair of solutions:

```
toulbar2 EXAMPLES/latin4.wcsp.xz -a=20 -div=12
```

- Download a crisp CSP file `GEOM40_6.wcsp.xz` (initial upper bound equal to 1). Count the number of solutions using #BTD [Favier2009a] using a min-fill variable ordering (warning, cannot use BTD to find all solutions in optimization):

```
toulbar2 EXAMPLES/GEOM40_6.wcsp.xz -O=-3 -a -B=1 -ub=1 -hbfs:
```

- Get a quick approximation of the number of solutions of a CSP with Approx#BTD [Favier2009a]:

```
toulbar2 EXAMPLES/GEOM40_6.wcsp.xz -O=-3 -a -B=1 -D -ub=1 -hbfs:
```

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.

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 [Givry2013a] (aka Soft Neighborhood Substitutability) preprocessing.

General control

-agap=[decimal] stops search if the absolute optimality gap reduces below the given value (provides guaranteed approximation) (default value is 0)

-rgap=[double] stops search if the relative optimality gap reduces below the given value (provides guaranteed approximation) (default value is 0)

-a=[integer] finds at most a given number of solutions with a cost strictly lower than the initial upper bound and stops, or if no integer is given, finds all solutions (or counts 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] gives 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.

-bt=[integer] gives a limit on the number of backtracks (9223372036854775807 by default)

-seed=[integer] random seed non-negative value or use current time if a negative value is given (default value is 1)

Preprocessing

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

-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 [Favier2011a] 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). See [Favier2011a].

-mst	find a maximum spanning tree ordering for DAC
-S	preprocessing only: performs singleton consistency (only in conjunction with option -A)

-M=[integer] preprocessing only: apply the Min Sum Diffusion algorithm (default is inactivated, with a number of iterations of 0). See [Cooper2010a].

-trws=[float] preprocessing only: enforces TRW-S until a given precision is reached (default value is 0.001). See Kolmogorov 2006.

--trws-order	replaces DAC order by Kolmogorov's TRW-S order.

–trws-n-iters=[integer] enforce at most N iterations of TRW-S (default value is 1000).

–trws-n-iters-no-change=[integer] stop TRW-S when N iterations did not change the lower bound up the given precision (default value is 5, -1=never).

–trws-n-iters-compute-ub=[integer] compute a basic upper bound every N steps during TRW-S (default value is 100)

Initial upper bounding

-l=[integer] limited discrepancy search [Ginsberg1995], 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/VNS restarts = 10000 by default)

-i=["string"] initial upper bound found by INCOP local search solver [idwalk:cp04]. 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)*] performs an elementary operation ('=':assign, '#':remove, '<':decrease, '>':increase) with value a on variable of index i (multiple operations are separated by a comma and no space) (without any argument, a complete assignment – used as initial upper bound and as a value heuristic – read from default file "sol" taken as a certificate or given directly as an additional input filename with ".sol" extension and without **-x**)

-ub=[decimal] gives an initial upper bound

-rasps=[integer] VAC-based upper bound probing heuristic (0: disable, >0: max. nb. of backtracks, 1000 if no integer given) (default value is 0)

-raspslds=[integer] VAC-based upper bound probing heuristic using LDS instead of DFS (0: DFS, >0: max. discrepancy) (default value is 0)

-raspsdeg=[integer] automatic threshold cost value selection for probing heuristic (default value is 10 degrees)

-raspsini	reset weighted degree variable ordering heuristic after doing upper bound probing

Tree search algorithms and tree decomposition selection

-hbfs=[integer] hybrid best-first search [Katsirelos2015a], 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 stored open nodes (default value is -1, i.e., no limit)

-B=[integer] (0) HBFS, (1) BTD-HBFS [Schiex2006a] [Katsirelos2015a], (2) RDS-BTD [Sanchez2009a], (3) RDS-BTD with path decomposition instead of tree decomposition [Sanchez2009a] (default value is 0)

-O=[filename] reads either a reverse variable elimination order (given by a list of variable indexes) from a file in order to build a tree decomposition (if BTD-like and/or variable elimination methods are used) or reads a valid tree decomposition directly (given by a list of clusters in topological order of a rooted forest, each line contains a cluster number, followed by a cluster parent number with -1 for the first/root(s) cluster(s), followed by a list of variable indexes). It is also used as a 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,
- (-7) default file 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 the 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 the 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=[float] merges leaf clusters with their fathers if small local treewidth (in conjunction with option "-e" and positive threshold value) or ratio of number of separator variables by number of cluster variables above a given threshold (in conjunction with option -vns) (default value is 0)

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

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

Variable neighborhood search algorithms

-vns unified decomposition guided variable neighborhood search [Ouali2017] (UDGVNS). A problem decomposition into clusters can be given as *.dec, *.cov, or *.order input files or using tree decomposition options such as -O. For a parallel version (UPDGVNS), use "mpirun -n [NbOfProcess] toulbar2 -vns problem.wcsp".

-vnsini=[integer] initial solution for VNS-like methods found: (-1) at random, (-2) min domain values, (-3) max domain values, (-4) first solution found by a complete method, (k=0 or more) tree search with k discrepancy max (-4 by default)

-ldsmin=[integer] minimum discrepancy for VNS-like methods (1 by default)

-ldsmax=[integer] maximum discrepancy for VNS-like methods (number of problem variables multiplied by maximum domain size -1 by default)

-ldsinc=[integer] discrepancy increment strategy for VNS-like methods using (1) Add1, (2) Mult2, (3) Luby operator (2 by default)

-kmin=[integer] minimum neighborhood size for VNS-like methods (4 by default)

-kmax=[integer] maximum neighborhood size for VNS-like methods (number of problem variables by default)

-kinc=[integer] neighborhood size increment strategy for VNS-like methods using: (1) Add1, (2) Mult2, (3) Luby operator (4) Add1/Jump (4 by default)

-best=[integer] stop VNS-like methods if a better solution is found (default value is 0)

Node processing & bounding options

-e=[integer] performs "on the fly" variable elimination of variable with small degree (less than or equal to a specified value, default is 3 creating a maximum of ternary cost functions). See [Larrosa2000].

-k=[integer] soft local consistency level (NC [Larrosa2002] with Strong NIC for global cost functions=0 [LL2009], (G)AC=1 [Schiex2000b] [Larrosa2002], D(G)AC=2 [CooperFCSP], FD(G)AC=3 [Larrosa2003], (weak) ED(G)AC=4 [Heras2005] [LL2010]) (default value is 4). See also [Cooper2010a] [LL2012asa].

-A=[integer] enforces VAC [Cooper2008] at each search node with a search depth less than a given value (default value is 0)

-V	VAC-based value ordering heuristic (default option)

-T=[decimal] threshold cost value for VAC (default value is 1)

-P=[decimal] threshold cost value for VAC during the preprocessing phase only (default value is 1)

-C=[float] multiplies all costs internally by this number when loading the problem (cannot be done with cfn format and probabilistic graphical models in uai/LG formats) (default value is 1)

-vacthr	automatic threshold cost value selection for VAC during search (must be combined with option -A)

-dee=[integer] restricted dead-end elimination [Givry2013a] (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 an optimal worst-case time complexity of DAC and EAC (can be slower in practice)

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 binary branching (by default) instead of n-ary branching. Uses binary branching for interval domains and small domains and dichotomic branching for large enumerated domains (see option -d).
-c	searches using binary branching with last conflict backjumping variable ordering heuristic [Lecoutre2009].

-q=[integer] use weighted degree variable ordering heuristic [boussemart2004] if the number of cost functions is less than the given value (default value is 1000000).

-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 [Schiex2014a] (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 the sorted domain based on unary costs.

-sortd	sorts domains in preprocessing based on increasing unary costs (works only for binary WCSPs).
-sortc	sorts constraints in preprocessing based on lexicographic ordering (1), decreasing DAC ordering (2 - default option), decreasing constraint tightness (3), DAC then tightness (4), tightness then DAC (5), randomly (6) or the opposite order if using a negative value.
-solr	solution-based phase saving (reuse last found solution as preferred value assignment in the value ordering heuristic) (default option).
-vacint	VAC-integrality/Full-EAC variable ordering heuristic (can be combined with option -A)

Diverse solutions

toulbar2 can search for a greedy sequence of diverse solutions with guaranteed local optimality and minimum pairwise Hamming distance [Ruffini2019a].

-div=[integer] minimum Hamming distance between diverse solutions (use in conjunction with -a=integer with a limit of 1000 solutions) (default value is 0)

-divm=[integer] diversity encoding method: 0:Dual 1:Hidden 2:Ternary (default value is 0)

-mdd=[integer] maximum relaxed MDD width for diverse solution global constraint (default value is 0)

-mddh=[integer] MDD relaxation heuristic: 0: random, 1: high div, 2: small div, 3: high unary costs (default value is 0)

Console output

-help	shows the default help message that toulbar2 prints when it gets no argument.

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

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

-s=[integer] shows each solution found during search. The solution is printed on one line, giving by default (-s=1) the value (integer) of each variable successively in increasing file order. For -s=2, the value name is used instead, and for -s=3, variable name=value name is printed instead.

File output

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

-w=[integer] 1: writes value numbers, 2: writes value names, 3: writes also variable names (default value is 1, this option can be used in combination with -w=filename).

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

-z=[integer] 1 or 3: saves original instance in 1-wcsp or 3-cfn format (1 by default), 2 or 4: saves after preprocessing in 2-wcsp or 4-cfn format (this option can be used in combination with -z=filename)

-x=[(,i[= # <>]a)*] performs an elementary operation ('=':assign, '#':remove, '<':decrease, '>':increase) with value a on variable of index i (multiple operations 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)

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 (greater than 1, default value is infinity)

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 number 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 number 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 number of ternary cost functions

  nary-{n}-{d}-{t1}-{p2}-{p3}…-{pn}-{seed}

  - pn is the number of n-ary cost functions

  salldiff-{n}-{d}-{t1}-{p2}-{p3}…-{pn}-{seed}

  - pn is the number of salldiff global cost functions (p2 and p3 still being used for the number of random binary and ternary cost functions). *salldiff* can be replaced by *gcc* or *regular* keywords with three possible forms (*e.g., sgcc, sgccdp, wgcc*) and by *knapsack*.

Input File formats

Notice that by default toulbar2 distinguishes file formats based on their extension. It is possible to read a file from a unix pipe using option `-stdin=[format]`; *e.g.*, `cat example.wcsp | toulbar2 --stdin=wcsp`

It is also possible to read and combine multiple problem files (warning, they must be all in the same format, either wcsp, cfn, or xml). Variables with the same name are merged (domains must be identical), otherwise the merge is based on variable indexes (wcsp format).

cfn format (.cfn, .cfn.gz, and .cfn.xz file extension)

With this JSON compatible format, it is possible:

- to give a name to variables and functions.
- to associate a local label to every value that is accessible inside toulbar2 (among others for heuristics design purposes).
- to use decimal and possibly negative costs.
- to solve both minimization and maximization problems.
- to debug your .cfn files: the parser gives a cause and line number when it fails.
- to use gzip'd or xz compressed files directly as input (.cfn.gz and .cfn.xz).
- to use dense descriptions for dense cost tables.

See a full description here (`CFN_format.pdf` file).

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 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 hard constraint with minimum capacity and a list of weights associated to the variables in the scope of the constraint (warning! it assumes Boolean 0/1 variables and is equivalent to a linear constraint with >= operator, use negative numbers to express the <= operator)
- 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 [Ficolofo2012]:
  - 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 1000
```

- clique cut ({x0,x1,x2,x3}) on Boolean variables such that value 1 is used at most once (warning! must add a clique of binary constraints too) :

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

(continues on next page)

(continued from previous page)

```
2 1 3 0 1 1 1 1000
2 2 3 0 1 1 1 1000
```

- knapsack constraint ({x0,x1,x2,x3}) on Boolean variables such that $2*x0+3*x1+4*x2+5*x3 >= 10$:

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

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
```

(continues on next page)

(continued from previous page)

```
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
```

UAI and 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
  ```

(continues on next page)

(continued from previous page)

```
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 corresponding 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
```

(continues on next page)

(continued from previous page)

```
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.)

In the LG format, probabilities are replaced by their logarithm.

- **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
```

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, which is the first integer in the clause. Weights must be greater than or equal to 1. Hard clauses have weight top and soft clauses have weight 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
```

Weighted Partial Max-SAT input format (.wcnf)

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

Example of Weighted Partial Max-SAT formula:

```
c
c comments Weighted 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
```

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. It can be controled by the option {tt -qpmult=[double]} which defines a coefficient multiplier for quadratic terms (default value is 2).

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 {em -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 $W_{ij} \neq 0$.

OPB format (.opb)

The OPB file format is used to express pseudo-Boolean satisfaction and optimization models. These models may only contain $0/1$ Boolean variables. The format is defined by an optional objective function followed by a set of linear constraints. Variables may be multiplied together in the objective function, but currently not in the constraints due to some restriction in the reader. The objective function must start with the **min:** or **max:** keyword followed by **coef_1 varname_1_1 varname_1_2 … coef2 varname_2_1 …** and end with a **;**. Linear constraints are composed in the same way, ended by a comparison operator (**<=**, **>=**, or **!=**) followed by the right-hand side coefficient and **;**. Each coefficient must be an integer beginning with its sign (**+** or **-** with no extra space). Comment lines start with a *.

An example with a quadratic objective and 7 linear constraints is:

```
max: +1 x1 x2 +2 x3 x4;
+1 x2 +1 x1 >= 1;
+1 x3 +1 x1 >= 1;
+1 x4 +1 x1 >= 1;
+1 x3 +1 x2 >= 1;
+1 x4 +1 x2 >= 1;
+1 x4 +1 x3 >= 1;
+2 x1 +2 x2 +2 x3 +2 x4 <= 7;
```

Internally, all integer costs are multiplied by a power of ten depending on the -precision option. For problems with big integers, try to reduce the precision (*e.g.*, use option -precision 0).

XCSP2.1 format (.xml)

CSP and weighted CSP in XML format XCSP 2.1, with constraints in extension only, can be read. See a description of this deprecated format here http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf.

Warning, toulbar2 must be compiled with a specific option XML in the cmake.

Linkage format (.pre)

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

Using it as a library

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

Using it from Python

A Python interface is now available. Compile toulbar2 with cmake option PYTB2 (and without MPI options) to generate a Python module **pytoulbar2** (in lib directory). See examples in `src/pytoulbar2.cpp` and *web/TUTORIALS* directory.

An older verion of toulbar2 was integrated inside Numberjack. See https://github.com/eomahony/Numberjack.

References

3.1.2 Reference Manual

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`.

toulbar2

Exact optimization for cost function networks and additive graphical models

master: cpd:

What is toulbar2?

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

toulbar2 won several competitions on deterministic and probabilistic graphical models:

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

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.

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.

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".

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
```

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).

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

Citing

Please use one of the following references for citing toulbar2:

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

What are the algorithms inside toulbar2?

- Soft arc consistency (AC): Arc consistency for Soft Constraints T. Schiex Proc. of CP'2000. Singapour, September 2000.
- More soft arc consistencies (NC, DAC, FDAC): In the quest of the best form of local consistency for Weighted CSP J. Larrosa & T. Schiex In Proc. of IJCAI-03. Acapulco, Mexico, 2003
- Soft existential arc consistency (EDAC): Existential arc consistency: Getting closer to full arc consistency in weighted csps S. de Givry, M. Zytnicki, F. Heras, and J. Larrosa In Proc. of IJCAI-05, Edinburgh, Scotland, 2005
- Depth-first Branch and Bound exploiting a tree decomposition (BTD): Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP S. de Givry, T. Schiex, and G. Verfaillie In Proc. of AAAI-06, Boston, MA, 2006
- Virtual arc consistency (VAC): Virtual arc consistency for weighted csp M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki In Proc. of AAAI-08, Chicago, IL, 2008
- Soft generalized arc consistencies (GAC, FDGAC): Towards Efficient Consistency Enforcement for Global Constraints in Weighted Constraint Satisfaction J. H. M. Lee and K. L. Leung In Proc. of IJCAI-09, Los Angeles, USA, 2010
- Russian doll search exploiting a tree decomposition (RDS-BTD): Russian doll search with tree decomposition M Sanchez, D Allouche, S de Givry, and T Schiex In Proc. of IJCAI'09, Pasadena (CA), USA, 2009
- Soft bounds arc consistency (BAC): Bounds Arc Consistency for Weighted CSPs M. Zytnicki, C. Gaspin, S. de Givry, and T. Schiex Journal of Artificial Intelligence Research, 35:593-621, 2009
- Counting solutions in satisfaction (#BTD, Approx_#BTD): Exploiting problem structure for solution counting A. Favier, S. de Givry, and P. Jégou In Proc. of CP-09, Lisbon, Portugal, 2009
- Soft existential generalized arc consistency (EDGAC): A Stronger Consistency for Soft Global Constraints in Weighted Constraint Satisfaction J. H. M. Lee and K. L. Leung In Proc. of AAAI-10, Boston, MA, 2010
- Preprocessing techniques (combines variable elimination and cost function decomposition): Pairwise decomposition for combinatorial optimization in graphical models A Favier, S de Givry, A Legarra, and T Schiex In Proc. of IJCAI-11, Barcelona, Spain, 2011
- Decomposable global cost functions (wregular, wamong, wsum): Decomposing global cost functions D Allouche, C Bessiere, P Boizumault, S de Givry, P Gutierrez, S Loudni, JP Métivier, and T Schiex In Proc. of AAAI-12, Toronto, Canada, 2012
- Pruning by dominance (DEE): Dead-End Elimination for Weighted CSP S de Givry, S Prestwich, and B O'Sullivan In Proc. of CP-13, pages 263-272, Uppsala, Sweden, 2013
- Hybrid best-first search exploiting a tree decomposition (HBFS): Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP D Allouche, S de Givry, G Katsirelos, T Schiex, and M Zytnicki In Proc. of CP-15, Cork, Ireland, 2015
- 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

Module Documentation

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>
    ...
    <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
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
```

(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
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
```

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.

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

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]);

    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;
```

(continues on next page)

(continued from previous page)

```
    // 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

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

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)

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.

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.

Virtual Arc Consistency enforcing

group **VAC**

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.

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.

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

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)

Warning: *Pour memo documents PRECEDENTS :* `User manual` | `Reference manual` | `WCSP Format` | `CFN Format`

3.2 Input formats

The available file formats (possibly compressed by gzip or xz, e.g., .cfn.gz, .wcsp.xz) are :

- Cost Function Network format (`.cfn` file extensions)
- Weighted Constraint Satisfaction Problem (`.wcsp` file extension)
- Probabilistic Graphical Model (.uai / .LG file extension ; the file format .LG is identical to .UAI except that we expect log-potentials)
- Weigthed Partial Max-SAT (.cnf/.wcnf file extension)
- Quadratic Unconstrained Pseudo-Boolean Optimization (`.qpbo` file extension)
- Pseudo-Boolean Optimization (.opb file extension)

3.3 Some examples

- A simple 2 variables maximization problem maximization.cfn in JSON-compatible CFN format, with decimal positive and negative costs.
- Random binary cost function network `example.wcsp`, with a specific variable ordering `example.order`, a tree decomposition `example.cov`, and a cluster decomposition `example.dec`
- Latin square 4x4 with random costs on each variable `latin4.wcsp`
- Radio link frequency assignment CELAR instances `scen06.wcsp`, `scen06.cov`, `scen06.dec`, `scen07.wcsp`

- Earth observation satellite management SPOT5 instances `404.wcsp` and `505.wcsp` with associated tree/cluster decompositions `404.cov`, `505.cov`, `404.dec`, `505.dec`
- Linkage analysis instance `pedigree9.uai`
- Computer vision superpixel-based image segmentation instance `GeomSurf-7-gm256.uai`
- Protein folding instance `1CM1.uai`
- Max-clique DIMACS instance `brock200_4.clq.wcnf`
- Graph 6-coloring instance `GEOM40_6.wcsp`

Many more instances available here and there.

3.4 Command line arguments

See *'Available options'* below :

```
c /toulbar2/build/bin/Linux/toulbar2  version : 1.1.1-139-gacc22b68-master␣
↪(1625740071), copyright (c) 2006-2020, toulbar2 team
*************************
* ToulBar2 Help Message *
*************************

Command line is:
toulbar2 problem_filename [options]

Available problem formats (specified by the filename extension) are:
   *.cfn : Cost Function Network format (see toulbar2 web site)
   *.wcsp : Weighted CSP format (see toulbar2 web site)
   *.wcnf : Weighted Partial Max-SAT format (see Max-SAT Evaluation)
   *.cnf : (Max-)SAT format
   *.qpbo : quadratic pseudo-Boolean optimization (unconstrained quadratic␣
↪programming) format (see also option -qpmult)
   *.opb : pseudo-Boolean optimization format
   *.uai : Bayesian network and Markov Random Field format (see UAI'08␣
↪Evaluation) followed by an optional evidence filename (performs MPE task,␣
↪see -logz for PR task, and write its solution in file .MPE or .PR using the␣
↪same directory as toulbar2)
   *.LG : Bayesian network and Markov Random Field format using logarithms␣
↪instead of probabilities
   *.xml : CSP and weighted CSP in XML format XCSP 2.1 (constraints in␣
↪extension only)
   *.pre : pedigree format (see doc/MendelSoft.txt for Mendelian error␣
↪correction)
   *.pre *.map : pedigree and genetic map formats (see doc/HaplotypeHalfSib.
↪txt for haplotype reconstruction in half-sib families)
   *.bep  : satellite scheduling format (CHOCO benchmark)

   *.order  : variable elimination order
   *.cov  : tree decomposition given by a list of clusters in topological␣
↪order of a rooted forest,
      each line contains a cluster number, then a cluster parent number with -
↪1 for the root(s) cluster(s), followed by a list of variable indexes
```
(continues on next page)

(continued from previous page)

```
   *.dec  : a list of overlapping clusters without the running intersection␣
↪property used by VNS-like methods,
      each line contains a list of variable indexes
   *.sol  : initial solution for the problem (given as initial upperbound plus␣
↪one and as default value heuristic, or only as initial upperbound if option -
↪x: is added)

Note: cfn, cnf, LG, qpbo, opb, uai, wcnf, wcsp formats can be read in gzip'd␣
↪or xz compressed format, e.g., toulbar2 problem.cfn.xz
Warning! File formats are recognized by filename extensions. To change the␣
↪default file format extension, use option --old_ext=".new" Examples: --cfn_
↪ext='.json' --wcspgz_ext='.wgz' --sol_ext='.sol2'

Available options are (use symbol ":" after an option to remove a default␣
↪option):
   -help : shows this help message
   -ub=[decimal] : initial problem upperbound (default value is␣
↪512409557603043100)
   -agap=[decimal] : stop search if the absolute optimality gap reduces below␣
↪the given value (provides guaranteed approximation) (default value is 0)
   -rgap=[double] : stop search if the relative optimality gap reduces below␣
↪the given value (provides guaranteed approximation) (default value is 0)
   -v=[integer] : verbosity level
   -s=[integer] : shows each solution found. 1 prints value numbers, 2 prints␣
↪value names, 3 prints also variable names (default 1)
   -w=[filename] : writes last/all solutions in filename (or "sol" if no␣
↪parameter is given)
   -w=[integer] : 1 writes value numbers, 2 writes value names, 3 writes also␣
↪variable names (default 1)
   -precision=[integer] defines the number of digits that should be␣
↪representable on probabilities in uai/pre files (default value is 7)
   -qpmult=[double] defines coefficient multiplier for quadratic terms␣
↪(default value is 2)
   -timer=[integer] : CPU time limit in seconds
   -bt=[integer] : limit on the number of backtracks (9223372036854775807 by␣
↪default)
   -seed=[integer] : random seed non-negative value or use current time if a␣
↪negative value is given (default value is 1)
   --stdin=[format] : read file from pipe ; e.g., cat example.wcsp | toulbar2 -
↪-stdin=wcsp
   -var=[integer] : searches by branching only on the first -the given value-␣
↪decision variables, assuming the remaining variables are intermediate␣
↪variables completely assigned by the decision variables (use a zero if all␣
↪variables are decision variables) (default value is 0)
   -b : searches using binary branching always instead of binary branching for␣
↪interval domains and n-ary branching for enumerated domains (default option)
   -svo : searches using a static variable ordering heuristic (same order as␣
↪DAC)
   -c : searches using binary branching with last conflict backjumping␣
↪variable ordering heuristic (default option)
   -q=[integer] : weighted degree variable ordering heuristic if the number of␣
↪cost functions is less than the given value (default value is 1000000)
```

(continues on next page)

(continued from previous page)

```
  -m=[integer] : variable ordering heuristic based on mean (m=1) or median␣
↪(m=2) costs (in conjunction with weighted degree heuristic -q) (default␣
↪value is 0)
  -d=[integer] : searches using dichotomic branching (d=1 splitting in the␣
↪middle of domain range, d=2 splitting in the middle of sorted unary costs)␣
↪instead of binary branching when current domain size is strictly greater␣
↪than 10 (default value is 1)
  -sortd : sorts domains based on increasing unary costs (warning! works only␣
↪for binary WCSPs)
  -sortc : sorts constraints based on lexicographic ordering (1), decreasing␣
↪DAC ordering (2), decreasing constraint tightness (3), DAC then tightness␣
↪(4), tightness then DAC (5), randomly (6) or the opposite order if using a␣
↪negative value (default value is 2)
  -solr : solution-based phase saving (default option)
  -e=[integer] : boosting search with variable elimination of small degree␣
↪(less than or equal to 3) (default value is 3)
  -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
  -nopre : removes all preprocessing options (equivalent to -e: -p: -t: -f: -
↪dec: -n: -mst: -dee: -trws:)
  -o : ensures optimal worst-case time complexity of DAC and EAC (can be␣
↪slower in practice)
  -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)
  -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)
  -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/VNS restarts = 10000 by default)
  -i=["string"] : initial upperbound found by INCOP local search solver.
      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
  -vns : unified decomposition guided variable neighborhood search (a problem␣
↪decomposition can be given as *.dec, *.cov, or *.order input files or using␣
↪tree decomposition options such as -O)
```

(continues on next page)

(continued from previous page)

```
  -vnsini=[integer] : initial solution for VNS-like methods found (-1) at␣
↪random, (-2) min domain values, (-3) max domain values, (-4) first solution␣
↪found by a complete method, (k=0 or more) tree search with k discrepancy max␣
↪(-4 by default)
  -ldsmin=[integer] : minimum discrepancy for VNS-like methods (1 by default)
  -ldsmax=[integer] : maximum discrepancy for VNS-like methods (number of␣
↪problem variables multiplied by maximum domain size -1 by default)
  -ldsinc=[integer] : discrepancy increment strategy for VNS-like methods␣
↪using (1) Add1, (2) Mult2, (3) Luby operator (2 by default)
  -kmin=[integer] : minimum neighborhood size for VNS-like methods (4 by␣
↪default)
  -kmax=[integer] : maximum neighborhood size for VNS-like methods (number of␣
↪problem variables by default)
  -kinc=[integer] : neighborhood size increment strategy for VNS-like methods␣
↪using (1) Add1, (2) Mult2, (3) Luby operator (4) Add1/Jump (4 by default)
  -best=[integer] : stop VNS-like methods if a better solution is found␣
↪(default value is 0)

  -z=[filename] : saves problem in wcsp (by default) or cfn format (see␣
↪below) in filename (or "problem.wcsp/.cfn"  if no parameter is given)
                  writes also the  graphviz dot file  and the degree␣
↪distribution of the input problem (wcsp format only)
  -z=[integer] : 1 or 3: saves original instance in 1-wcsp or 3-cfn format (1␣
↪by default), 2 or 4: saves after preprocessing in 2-wcsp or 4-cfn format␣
↪(this option can be combined with the previous one)
  -Z=[integer] : debug mode (save problem at each node if verbosity option -
↪v=num >= 1 and -Z=num >=3)
  -x=[(,i[=#<>]a)*] : performs an elementary operation ('=':assign, '#
↪':remove, '<':decrease, '>':increase) with value a on variable of index i␣
↪(multiple operations 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" taken as a certificate or given as␣
↪input filename with ".sol" extension)

  -M=[integer] : preprocessing only: Min Sum Diffusion algorithm (default␣
↪number of iterations is 0)
  -A=[integer] : enforces VAC at each search node with a search depth less␣
↪than the absolute value of a given value, if negative value then VAC is not␣
↪performed inside depth-first search of hybrid best-first search (default␣
↪value is 0)
  -T=[decimal] : threshold cost value for VAC (default value is 1)
  -P=[decimal] : threshold cost value for VAC during the preprocessing phase␣
↪(default value is 1)
  -C=[float] : multiplies all costs internally by this number when loading␣
↪the problem (default value is 1)
  -S : preprocessing only: performs singleton consistency (only in␣
↪conjunction with option "-A")
  -V : VAC-based value ordering heuristic (default option)
  -vacint : VAC-integrality/Full-EAC variable ordering heuristic
  -vacthr : automatic threshold cost value selection for VAC during search
  -rasps=[integer] : VAC-based upper bound probing heuristic (0: disable, >0:␣
↪max. nb. of backtracks) (default value is 0)
```

(continues on next page)

(continued from previous page)

```
  -raspslds=[integer] : VAC-based upper bound probing heuristic using LDS␣
↪instead of DFS (0: DFS, >0: max. discrepancy) (default value is 0)
  -raspsdeg=[integer] : automatic threshold cost value selection for probing␣
↪heuristic (default value is 10°)
  -raspsini : reset weighted degree variable ordering heuristic after doing␣
↪upper bound probing
  -trws=[float] : enforces TRW-S in preprocessing until a given precision is␣
↪reached (default value is -1)
  --trws-order : replaces DAC order by Kolmogorov's TRW-S order
  --trws-n-iters=[integer] : enforce at most N iterations of TRW-S (default␣
↪value is 1000)
  --trws-n-iters-no-change=[integer] : stop TRW-S when N iterations did not␣
↪change the lower bound up the given precision (default value is 5, -1=never)
  --trws-n-iters-compute-ub=[integer] : compute UB every N steps in TRW-S␣
↪(default value is 100)

  -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 or directly a valid tree␣
↪decomposition (given by a list of clusters in topological order of a rooted␣
↪forest, each line contains a cluster number,
      followed by a cluster parent number with -1 for the root(s) cluster(s),␣
↪followed by a list of variable indexes) from a file used for BTD-like and␣
↪variable elimination methods, and also 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,
                         (-7) default file ordering (the same if this option␣
↪is missing, i.e. use the variable order in which variables appear in the␣
↪problem file)
                         (-8) lexicographic ordering of variable names.
  -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=[float] : merges leaf clusters with their fathers if small local␣
↪treewidth (in conjunction with option "-e" and positive threshold value) or␣
↪ratio of number of separator variables by number of cluster variables above␣
↪a given threshold (in conjunction with option "-vns") (default value is 0)
  -R=[integer] : choice for a specific root cluster number
  -I=[integer] : choice for solving only a particular rooted cluster subtree␣
↪(with RDS-BTD only)
```

(continues on next page)

(continued from previous page)

```
   -a=[integer] : finds at most a given number of solutions with a cost␣
↪strictly lower than the initial upper bound and stops, or if no integer is␣
↪given, finds all solutions (or counts the number of zero-cost satisfiable␣
↪solutions in conjunction with BTD)
   -div=[integer] : minimum Hamming distance between diverse solutions (use in␣
↪conjunction with -a=integer with a limit of 1000 solutions) (default value␣
↪is 0)
   -divm=[integer] : diversity encoding method: 0:Dual 1:Hidden 2:Ternary␣
↪3:Knapsack (default value is 0)
   -mdd=[integer] : maximum relaxed MDD width for diverse solution global␣
↪constraint (default value is 0)
   -mddh=[integer] : MDD relaxation heuristic: 0: random, 1: high div, 2:␣
↪small div, 3: high unary costs (default value is 0)
   -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)
   -epsilon=[float] : approximation factor for computing the partition␣
↪function (greater than 1, default value is inf)

   -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)
---------------------------
Alternatively one can call the random problem generator with the following␣
↪options:

   -random=[bench profile]  : bench profile must be specified as follow :
                         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 (and␣
↪will overwrite -seed)
   or:
       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)
 or:
      tern-{n}-{d}-{t1}-{p2}-{p3}-{seed}  p3 is the num of ternary cost␣
↪functions
```

(continues on next page)

(continued from previous page)

```
 or:
      nary-{n}-{d}-{t1}-{p2}-{p3}...-{pn}-{seed}  pn is the num of n-ary cost␣
↪functions
 or:
      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)
---------------------------
```

CHAPTER

FOUR

PUBLICATIONS

4.1 Related publications

4.1.1 What are the algorithms inside toulbar2 ?

- **Soft arc consistencies (NC, AC, 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, Pasadena (CA), USA, 2009.

- **Russian doll search exploiting a tree decomposition (RDS-BTD)**

  Russian doll search with tree decomposition, M Sanchez, D Allouche, S de Givry, and T Schiex, In Proc. of IJCAI-09, Pasadena (CA), USA, 2009.

- **Soft bounds arc consistency (BAC)**

  Bounds Arc Consistency for Weighted CSPs, M. Zytnicki, C. Gaspin, S. de Givry, and T. Schiex, Journal of Artificial Intelligence Research, 35:593-621, 2009.

- **Counting solutions in satisfaction (#BTD, Approx_#BTD)**

  Exploiting problem structure for solution counting, A. Favier, S. de Givry, and P. Jégou, In Proc. of CP-09, Lisbon, Portugal, 2009.

- **Soft existential generalized arc consistency (EDGAC)**

  A Stronger Consistency for Soft Global Constraints in Weighted Constraint Satisfaction, J. H. M. Lee and K. L. Leung, In Proc. of AAAI-10, Boston, MA, 2010 .

- **Preprocessing techniques (combines variable elimination and cost function decomposition)**

  Pairwise decomposition for combinatorial optimization in graphical models, A Favier, S de Givry, A Legarra, and T Schiex, In Proc. of IJCAI-11, Barcelona, Spain, 2011.

- **Decomposable global cost functions (wregular, wamong, wsum)**

  Decomposing global cost functions, D Allouche, C Bessiere, P Boizumault, S de Givry, P Gutierrez, S Loudni, JP Métivier, and T Schiex, In Proc. of AAAI-12, Toronto, Canada, 2012.

- **Pruning by dominance (DEE)**

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

- **Hybrid best-first search exploiting a tree decomposition (HBFS)**

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

- **Unified parallel decomposition guided variable neighborhood search (UDGVNS/UPDGVNS)**

  Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization, A Ouali, D Allouche, S de Givry, S Loudni, Y Lebbah, F Eckhardt, and L Loukil, In Proc. of UAI-17, pages 550-559, Sydney, Australia, 2017.

  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, 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-bound probing (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.

4.1.2 toulbar2 for Combinatorial Optimization in Life Sciences

- **Computational Protein Design**

  Designing Peptides on a Quantum Computer, Vikram Khipple Mulligan, Hans Melo, Haley Irene Merritt, Stewart Slocum, Brian D. Weitzner, Andrew M. Watkins, P. Douglas Renfrew, Craig Pelissier, Paramjit S. Arora, and Richard Bonneau, bioRxiv, 2019.

  Computational design of symmetrical eight-bladed β-propeller proteins, Noguchi, H., Addy, C., Simoncini, D., Wouters, S., Mylemans, B., Van Meervelt, L., Schiex, T., Zhang, K., Tameb, J., and Voet, A., IUCrJ, 6(1), 2019.

  Positive Multi-State Protein Design, Jelena Vučinić, David Simoncini, Manon Ruffini, Sophie Barbe, Thomas Schiex, Bioinformatics, 2019.

  Cost function network-based design of protein-protein interactions: predicting changes in binding affinity, Clément Viricel, Simon de Givry, Thomas Schiex, and Sophie Barbe, Bioinformatics, 2018.

  Algorithms for protein design, Pablo Gainza, Hunter M Nisonoff, Bruce R Donald, Current Opinion in Structural Biology, 39:6-26, 2016.

Fast search algorithms for computational protein design, Seydou Traoré, Kyle E Roberts, David Allouche, Bruce R Donald, Isabelle André, Thomas Schiex, and Sophie Barbe, Journal of computational chemistry, 2016.

Comparing three stochastic search algorithms for computational protein design: Monte Carlo, replica exchange Monte Carlo, and a multistart, steepest-descent heuristic, David Mignon, Thomas Simonson, Journal of computational chemistry, 2016.

Protein sidechain conformation predictions with an mmgbsa energy function, Thomas Gaillard, Nicolas Panel, and Thomas Simonson, Proteins: Structure, Function, and Bioinformatics, 2016.

Improved energy bound accuracy enhances the efficiency of continuous protein design, Kyle E Roberts and Bruce R Donald, Proteins: Structure, Function, and Bioinformatics, 83(6):1151-1164, 2015.

Guaranteed discrete energy optimization on large protein design problems, D. Simoncini, D. Allouche, S. de Givry, C. Delmas, S. Barbe, and T. Schiex, Journal of Chemical Theory and Computation, 2015.

Computational protein design as an optimization problem, David Allouche, Isabelle André, Sophie Barbe, Jessica Davies, Simon de Givry, George Katsirelos, Barry O'Sullivan, Steve Prestwich, Thomas Schiex, and Seydou Traoré, Journal of Artificial Intelligence, 212:59-79, 2014.

A new framework for computational protein design through cost function network optimization, Seydou Traoré, David Allouche, Isabelle André, Simon de Givry, George Katsirelos, Thomas Schiex, and Sophie Barbe, Bioinformatics, 29(17):2129-2136, 2013.

- **Genetics**

  Optimal haplotype reconstruction in half-sib families, Aurélie Favier, Jean-Michel Elsen, Simon de Givry, and Andrès Legarra, ICLP-10 workshop on Constraint Based Methods for Bioinformatics, Edinburgh, UK, 2010.

  Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques, Marti Sanchez, Simon de Givry, and Thomas Schiex, Constraints, 13(1-2):130-154, 2008. See also Mendelsoft integrated in the QTLmap Quantitative Genetics platform from INRA GA dept.

- **RNA motif search**

  Darn! a weighted constraint solver for RNA motif localization, Matthias Zytnicki, Christine Gaspin, and Thomas Schiex, Constraints, 13(1-2):91-109, 2008.

- **Agronomy**

  Solving the crop allocation problem using hard and soft constraints, Mahuna Akplogan, Simon de Givry, Jean-Philippe Métivier, Gauthier Quesnel, Alexandre Joannon, and Frédérick Garcia, RAIRO - Operations Research, 47:151-172, 2013.

4.1.3 Other publications mentioning toulbar2

- **Constraint Satisfaction, Distributed Constraint Optimization**

  Graph Based Optimization For Multiagent Cooperation, Arambam James Singh, Akshat Kumar, In Proc. of AAMAS, 2019.

  Probabilistic Inference Based Message-Passing for Resource Constrained DCOPs, Supriyo Ghosh, Akshat Kumar, Pradeep Varakantham, In Proc. of IJCAI, 2015.

  SAT-based MaxSAT algorithms, Carlos Ansótegui and Maria Luisa Bonet and Jordi Levy, Artificial Intelligence, 196:77-105, 2013.

Local Consistency and SAT-Solvers, P. Jeavons and J. Petke, Journal of Artificial Intelligence Research, 43:329-351, 2012.

- **Data Mining and Machine Learning**

  Pushing Data in CP Models Using Graphical Model Learning and Solving, Céline Brouard, Simon de Givry, and Thomas Schiex, In Proc. of CP-20, Louvain-la-neuve, Belgium, 2020.

  A constraint programming approach for mining sequential patterns in a sequence database, Jean-Philippe Métivier, Samir Loudni, and Thierry Charnois, In Proc. of the ECML/PKDD Workshop on Languages for Data Mining and Machine Learning, Praha, Czech republic, 2013.

- **Timetabling, planning and POMDP**

  Solving a Judge Assignment Problem Using Conjunctions of Global Cost Functions, S de Givry, J.H.M. Lee, K.L. Leung, and Y.W. Shum, In Proc. of CP-14, pages 797-812, Lyon, France, 2014.

  Optimally solving Dec-POMDPs as continuous-state MDPs, Jilles Steeve Dibangoye, Christopher Amato, Olivier Buffet, and François Charpillet, In Proc. of IJCAI, pages 90-96, 2013.

  A weighted csp approach to cost-optimal planning, Martin C Cooper, Marie de Roquemaurel, and Pierre Régnier, Ai Communications, 24(1):1-29, 2011.

  Point-based backup for decentralized POMDPs: Complexity and new algorithms, Akshat Kumar and Shlomo Zilberstein, In Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, 1:1315-1322, 2010.

- **Inference, Sampling, and Diagnostic**

  Mohamed-Hamza Ibrahim, Christopher Pal and Gilles Pesant, Leveraging cluster backbones for improving MAP inference in statistical relational models, In Ann. Math. Artif. Intell. 88, No. 8, 907-949, 2020.

  C. Viricel, D. Simoncini, D. Allouche, S. de Givry, S. Barbe, and T. Schiex, Approximate counting with deterministic guarantees for affinity computations, In Proc. of Modeling, Computation and Optimization in Information Systems and Management Sciences - MCO'15, Metz, France, 2015.

  Discrete sampling with universal hashing, Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman, In Proc. of NIPS, pages 2085-2093, 2013.

  Compiling ai engineering models for probabilistic inference, Paul Maier, Dominik Jain, and Martin Sachenbacher, In KI 2011: Advances in Artifcial Intelligence, pages 191-203, 2011.

  Diagnostic hypothesis enumeration vs. probabilistic inference for hierarchical automata models, Paul Maier, Dominik Jain, and Martin Sachenbacher, In Proc. of the International Workshop on Principles of Diagnosis, Murnau, Germany, 2011.

- **Computer Vision and Energy Minimization**

  Exact MAP-inference by Confining Combinatorial Search with LP Relaxation, Stefan Haller, Paul Swoboda, Bogdan Savchynskyy, In Proc. of AAAI, 2018.

- **Computer Music**

  Exploiting structural relationships in audio music signals using markov logic networks, Hélène Papadopoulos and George Tzanetakis, In Proc. of 38th International Conference on Acoustics, Speech, and Signal Processing (ICASSP), pages 4493-4497, Canada, 2013.

  Modeling chord and key structure with markov logic, Hélène Papadopoulos and George Tzanetakis, In Proc. of the Society for Music Information Retrieval (ISMIR), pages 121-126, 2012.

- **Inductive Logic Programming**

Extension of the top-down data-driven strategy to ILP, Erick Alphonse and Céline Rouveirol, In Proc. of Inductive Logic Programming, pages 49-63, 2007.

- **Other domains**

  An automated model abstraction operator implemented in the multiple modeling environment MOM, Peter Struss, Alessandro Fraracci, and D Nyga, In Proc. of the 25th International Workshop on Qualitative Reasoning, Barcelona, Spain, 2011.

  Modeling Flowchart Structure Recognition as a Max-Sum Problem, Martin Bresler, Daniel Prusa, Václav Hlavác, In Proc. of International Conference on Document Analysis and Recognition, Washington, DC, USA, 1215-1219, 2013.

4.2 Specific Events

- tutorial on cost function networks at CP2020 (teaser, part1, part2 videos, and script
- tutorial on cost function networks at PFIA 2019 (part1, part2, demo), Toulouse, France, July 4th, 2019.
- talk on toulbar2 latest algorithmic features at ISMP 2018, Bordeaux, France, July 6, 2018.
- toulbar2 projects meeting at CP 2016, Toulouse, France, September 5, 2016.

CHAPTER

FIVE

TUTORIALS

5.1 Weighted n-queen problem

5.1.1 Brief description

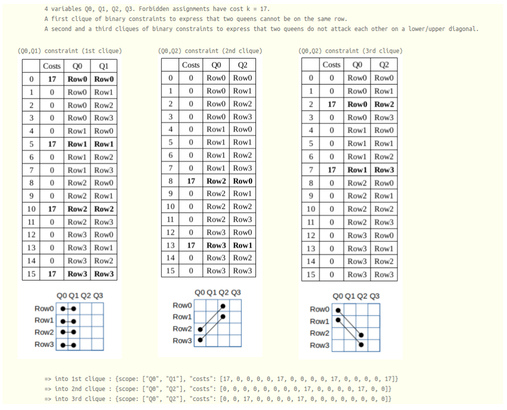
The problem consists in assigning N queens on a NxN chessboard with random weights in (1..N) associated to every cell such that each queen does not attack another queen and the sum of the weights of queen's selected cells is minimized.

5.1.2 CFN model

A solution must have only one queen per column and per row. We create N variables for every column with domain size N to represent the selected row for each queen. A clique of binary constraints is used to express that two queens cannot be on the same row. Forbidden assignments have cost k=N**2+1. Two other cliques of binary constraints are used to express that two queens do not attack each other on a lower/upper diagonal.

5.1.3 Example for N=4 in JSON .cfn format

More details :



```
4 variables Q0, Q1, Q2, Q3. Forbidden assignments have cost k = 17.
A first clique of binary constraints to express that two queens cannot be on the same row.
A second and a third cliques of binary constraints to express that two queens do not attack each other on a lower/upper diagonal.
```

(Q0,Q1) constraint (1st clique)

	Costs	Q0	Q1
0	**17**	**Row0**	**Row0**
1	0	Row0	Row1
2	0	Row0	Row2
3	0	Row0	Row3
4	0	Row1	Row0
5	**17**	**Row1**	**Row1**
6	0	Row1	Row2
7	0	Row1	Row3
8	0	Row2	Row0
9	0	Row2	Row1
10	**17**	**Row2**	**Row2**
11	0	Row2	Row3
12	0	Row3	Row0
13	0	Row3	Row1
14	0	Row3	Row2
15	**17**	**Row3**	**Row3**

(Q0,Q2) constraint (2nd clique)

	Costs	Q0	Q2
0	0	Row0	Row0
1	0	Row0	Row1
2	0	Row0	Row2
3	0	Row0	Row3
4	0	Row1	Row0
5	0	Row1	Row1
6	0	Row1	Row2
7	0	Row1	Row3
8	**17**	**Row2**	**Row0**
9	0	Row2	Row1
10	0	Row2	Row2
11	0	Row2	Row3
12	0	Row3	Row0
13	**17**	**Row3**	**Row1**
14	0	Row3	Row2
15	0	Row3	Row3

(Q0,Q2) constraint (3rd clique)

	Costs	Q0	Q2
0	0	Row0	Row0
1	0	Row0	Row1
2	**17**	**Row0**	**Row2**
3	0	Row0	Row3
4	0	Row1	Row0
5	0	Row1	Row1
6	0	Row1	Row2
7	**17**	**Row1**	**Row3**
8	0	Row2	Row0
9	0	Row2	Row1
10	0	Row2	Row2
11	0	Row2	Row3
12	0	Row3	Row0
13	0	Row3	Row1
14	0	Row3	Row2
15	0	Row3	Row3

```
=> into 1st clique : {scope: ["Q0", "Q1"], "costs": [17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17]}
=> into 2nd clique : {scope: ["Q0", "Q2"], "costs": [0, 0, 0, 0, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0]}
=> into 3rd clique : {scope: ["Q0", "Q2"], "costs": [0, 0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 0, 0, 0, 0]}
```

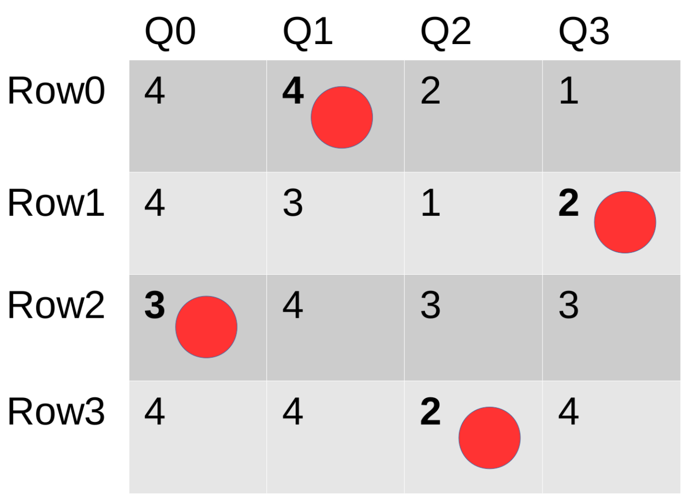
```
{
  problem: { "name": "4-queen", "mustbe": "<17" },
  variables: {"Q0":["Row0", "Row1", "Row2", "Row3"], "Q1":["Row0", "Row1",
↪"Row2", "Row3"],
```

(continues on next page)

(continued from previous page)

```
              "Q2":["Row0", "Row1", "Row2", "Row3"], "Q3":["Row0", "Row1",
↪"Row2", "Row3"]},
  functions: {
    {scope: ["Q0", "Q1"], "costs": [17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0,␣
↪0, 0, 17]},
    {scope: ["Q0", "Q2"], "costs": [17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0,␣
↪0, 0, 17]},
    {scope: ["Q0", "Q3"], "costs": [17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0,␣
↪0, 0, 17]},
    {scope: ["Q1", "Q2"], "costs": [17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0,␣
↪0, 0, 17]},
    {scope: ["Q1", "Q3"], "costs": [17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0,␣
↪0, 0, 17]},
    {scope: ["Q2", "Q3"], "costs": [17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0,␣
↪0, 0, 17]},

    {scope: ["Q0", "Q1"], "costs": [0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0,
↪ 17, 0]},
    {scope: ["Q0", "Q2"], "costs": [0, 0, 0, 0, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17,
↪ 0, 0]},
    {scope: ["Q0", "Q3"], "costs": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 17, 0,␣
↪0, 0]},
    {scope: ["Q1", "Q2"], "costs": [0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0,
↪ 17, 0]},
    {scope: ["Q1", "Q3"], "costs": [0, 0, 0, 0, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17,
↪ 0, 0]},
    {scope: ["Q2", "Q3"], "costs": [0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0,
↪ 17, 0]},

    {scope: ["Q0", "Q1"], "costs": [0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0,␣
↪0, 0, 0]},
    {scope: ["Q0", "Q2"], "costs": [0, 0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 0, 0,
↪ 0, 0]},
    {scope: ["Q0", "Q3"], "costs": [0, 0, 0, 17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0]},
    {scope: ["Q1", "Q2"], "costs": [0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0,␣
↪0, 0, 0]},
    {scope: ["Q1", "Q3"], "costs": [0, 0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 0, 0,
↪ 0, 0]},
    {scope: ["Q2", "Q3"], "costs": [0, 17, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 0,␣
↪0, 0, 0]},

    {scope: ["Q0"], "costs": [4, 4, 3, 4]},
    {scope: ["Q1"], "costs": [4, 3, 4, 4]},
    {scope: ["Q2"], "costs": [2, 1, 3, 2]},
    {scope: ["Q3"], "costs": [1, 2, 3, 4]}}
}
```

Optimal solution with cost 11 for the 4-queen example :



	Q0	Q1	Q2	Q3
Row0	4	**4** ●	2	1
Row1	4	3	1	**2** ●
Row2	**3** ●	4	3	3
Row3	4	4	**2** ●	4

5.1.4 Python model generator

The following code using python3 interpreter will generate the previous example if called without argument. Otherwise the first argument is the number of queens N (e.g. "python3 queens.py 8").

Note: Notice that the first lines of code (import and functions flatten and cfn) are needed by all the other tutorial examples.

queens.py

```
import sys
from random import randint, seed
seed(123456789)

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(problem["variables"]):
```

(continues on next page)

(continued from previous page)

```
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        for j,x in enumerate(e.get("scope")):
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
        print('], ', end='')
        if e.get("type") is not None:
            print('"type:" %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
                        print('"%s"' % c, end='')
                    else:
```

(continues on next page)

(continued from previous page)

```
                        print('%s' % c, end='')
                print(']', end='')
        print('}', end='')
    print('}\n}')

def model(N, k):
    Var = ["Q" + str(i) for i in range(N)]
    Queen = {
        "name": str(N) + "-queen",
        "variables": [(Var[i], ["Row" + str(j) for j in range(N)]) for i in␣
↪range(N)],
        "functions":
            [
             # permutation constraints expressed by a clique of binary␣
↪constraints
             [{"scope": [Var[i], Var[j]], "costs": [0 if a != b else k for a␣
↪in range(N) for b in range(N)]} for i in range(N) for j in range(N) if (i <␣
↪j)],
             # upper diagonal constraints
             [{"scope": [Var[i], Var[j]], "costs": [0 if a + i != b + j else k␣
↪for a in range(N) for b in range(N)]} for i in range(N) for j in range(N) if␣
↪(i < j)],
             # lower diagonal constraints
             [{"scope": [Var[i], Var[j]], "costs": [0 if a - i != b - j else k␣
↪for a in range(N) for b in range(N)]} for i in range(N) for j in range(N) if␣
↪(i < j)],
             # random unary costs
             [{"scope": [Var[i]], "costs": [randint(1,N) for a in range(N)]}␣
↪for i in range(N)]
            ]
         }
    return Queen

if __name__ == '__main__':
    # read parameters
    N = int(sys.argv[1]) if len(sys.argv) > 1 else 4
    # infinite cost
    k = N**2+1
    # dump problem into JSON .cfn format for minimization
    cfn(model(N, k), True, k)
    # or for maximization
    #cfn(model(N, -k), False, -k)
```

5.2 Weighted latin square problem

5.2.1 Brief description

The problem consists in assigning a value from 0 to N-1 to every cell of a NxN chessboard. Each row and each column must be a permutation of N values. For each cell, a random weight in (1…N) is associated to every domain value. The objective is to find a complete assignment where the sum of the weights associated to the selected values for the cells is minimized.

5.2.2 CFN model

We create NxN variables for all cells with domain size N. A hard AllDifferent global cost function is used to model a permutation for every row and every column. Random weights are generated for every cell and domain value. Forbidden assignments have cost k=N**3+1.

5.2.3 Example for N=4 in JSON .cfn format

```
{
  problem: { "name": "LatinSquare4", "mustbe": "<65" },
  variables: {"X0_0": 4, "X0_1": 4, "X0_2": 4, "X0_3": 4, "X1_0": 4, "X1_1": 4,
↪ "X1_2": 4, "X1_3": 4, "X2_0": 4, "X2_1": 4, "X2_2": 4, "X2_3": 4, "X3_0": 4,
↪ "X3_1": 4, "X3_2": 4, "X3_3": 4},
  functions: {
    {scope: ["X0_0", "X0_1", "X0_2", "X0_3"], "type:" salldiff, "params": {
↪"metric": "var", "cost": 65}},
    {scope: ["X1_0", "X1_1", "X1_2", "X1_3"], "type:" salldiff, "params": {
↪"metric": "var", "cost": 65}},
    {scope: ["X2_0", "X2_1", "X2_2", "X2_3"], "type:" salldiff, "params": {
↪"metric": "var", "cost": 65}},
    {scope: ["X3_0", "X3_1", "X3_2", "X3_3"], "type:" salldiff, "params": {
↪"metric": "var", "cost": 65}},

    {scope: ["X0_0", "X1_0", "X2_0", "X3_0"], "type:" salldiff, "params": {
↪"metric": "var", "cost": 65}},
    {scope: ["X0_1", "X1_1", "X2_1", "X3_1"], "type:" salldiff, "params": {
↪"metric": "var", "cost": 65}},
    {scope: ["X0_2", "X1_2", "X2_2", "X3_2"], "type:" salldiff, "params": {
↪"metric": "var", "cost": 65}},
    {scope: ["X0_3", "X1_3", "X2_3", "X3_3"], "type:" salldiff, "params": {
↪"metric": "var", "cost": 65}},

    {scope: ["X0_0"], "costs": [4, 4, 3, 4]},
    {scope: ["X0_1"], "costs": [4, 3, 4, 4]},
    {scope: ["X0_2"], "costs": [2, 1, 3, 2]},
    {scope: ["X0_3"], "costs": [1, 2, 3, 4]},
    {scope: ["X1_0"], "costs": [3, 1, 3, 3]},
    {scope: ["X1_1"], "costs": [4, 1, 1, 1]},
    {scope: ["X1_2"], "costs": [4, 1, 1, 3]},
    {scope: ["X1_3"], "costs": [4, 4, 1, 4]},
    {scope: ["X2_0"], "costs": [1, 3, 3, 2]},
```

(continues on next page)

(continued from previous page)

```
    {scope: ["X2_1"], "costs": [2, 1, 3, 1]},
    {scope: ["X2_2"], "costs": [3, 4, 2, 2]},
    {scope: ["X2_3"], "costs": [2, 3, 1, 3]},
    {scope: ["X3_0"], "costs": [3, 4, 4, 2]},
    {scope: ["X3_1"], "costs": [3, 2, 4, 4]},
    {scope: ["X3_2"], "costs": [4, 1, 3, 4]},
    {scope: ["X3_3"], "costs": [4, 4, 4, 3]}}
}
```

Optimal solution with cost 35 for the latin 4-square example (in red, weights associated to the selected values) :

4, 4, 3, 4	4, 3, 4, 4	2, 1, 3, 2	1, 2, 3, 4
3	2	0	1
3, 1, 3, 3	4, 1, 1, 1	4, 1, 1, 3	4, 4, 1, 4
1	3	2	0
1, 3, 3, 2	2, 1, 3, 1	3, 4, 2, 2	2, 3, 1, 3
0	1	3	2
3, 4, 4, 2	3, 2, 4, 4	4, 1, 3, 4	4, 4, 4, 3
2	0	1	3

5.2.4 Python model generator

The following code using python3 interpreter will generate the previous example if called without argument. Otherwise the first argument is the dimension N of the chessboard (e.g. “python3 latinsquare.py 6”).

latinsquare.py

```
import sys
from random import randint, seed
seed(123456789)

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
```

(continues on next page)

(continued from previous page)

```
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(problem["variables"]):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        for j,x in enumerate(e.get("scope")):
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
        print('], ', end='')
        if e.get("type") is not None:
            print('"type:" %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
```

(continues on next page)

(continued from previous page)

```
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
                        print('"%s"' % c, end='')
                    else:
                        print('%s' % c, end='')
                print(']', end='')
        print('}', end='')
    print('}\n}')

def model(N, k):
    Var = {(i,j): "X" + str(i) + "_" + str(j) for i in range(N) for j in␣
↪range(N)}
    LatinSquare = {
        "name": "LatinSquare" + str(N),
        "variables": [(Var[(i,j)], N) for i in range(N) for j in range(N)],
        "functions":
            [# permutation constraints on rows
                [{"scope": [Var[(i,j)] for j in range(N)], "type": "salldiff",
↪"params": {"metric": "var", "cost": k}} for i in range(N)],
             # permutation constraints on columns
             [{"scope": [Var[(i,j)] for i in range(N)], "type": "salldiff",
↪"params": {"metric": "var", "cost": k}} for j in range(N)],
              # random unary costs on every cell
             [{"scope": [Var[(i,j)]], "costs": [randint(1, N) for a in␣
↪range(N)]} for i in range(N) for j in range(N)]
            ]
    }
    return LatinSquare

if __name__ == '__main__':
    # read parameters
    N = int(sys.argv[1]) if len(sys.argv) > 1 else 4
    # infinite cost
    k = N**3+1
    # dump problem into JSON .cfn format for minimization
    cfn(model(N, k), True, k)
```

5.3 Radio link frequency assignment problem

5.3.1 Brief description

The problem consists in assigning frequencies to radio communication links in such a way that no interferences occurs. Domains are set of integers (non necessarily consecutives). Two types of constraints occur: (I) the absolute difference between two frequencies should be greater than a given number d_i (| x - y | > d_i), or (II) the absolute difference between two frequencies should exactly be equal to a given number d_i (| x - y | = d_i). Different deviations d_i, i in 0..4, may exist for the same pair of links. d_0 corresponds to hard constraints while higher deviations are soft constraints that can be violated with an associated cost a_i. Moreover, pre-assigned frequencies may be known for some links which are either hard or soft preferences (mobility cost b_i, i in 0..4). The goal is to minimize the weighted sum of violated constraints. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.P. Constraints (1999) 4: 79.

5.3.2 CFN model

We create N variables for every radio link with a given integer domain. Hard and soft binary cost functions express interference constraints with possible deviations. Unary cost functions are used to model mobility costs.

5.3.3 Data

Original data files can be download from the cost function library FullRLFAP. Their format is described here. You can try a small example CELAR6-SUB1 (`var.txt`, `dom.txt`, `ctr.txt`, `cst.txt`) with optimum value equal to 2669.

5.3.4 Python model generator

The following code using python3 interpreter will generate the corresponding cost function network (e.g. "python3 rlfap.py var.txt dom.txt ctr.txt cst.txt").

rlfap.py

```
import sys

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
```

(continues on next page)

(continued from previous page)

```
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(problem["variables"]):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        for j,x in enumerate(e.get("scope")):
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
        print('], ', end='')
        if e.get("type") is not None:
            print('"type:" %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
```

(continues on next page)

(continued from previous page)

```
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
                        print('"%s"' % c, end='')
                    else:
                        print('%s' % c, end='')
                print(']', end='')
            print('}', end='')
        print('}\n}')

class Data:
    def __init__(self, var, dom, ctr, cst):
        self.var = list()
        self.dom = {}
        self.ctr = list()
        self.cost = {}
        self.nba = {}
        self.nbb = {}
        self.top = 0

        stream = open(var)
        for line in stream:
            if len(line.split())>=4:
                (varnum, vardom, value, mobility) = line.split()[:4]
                self.var.append((int(varnum), int(vardom), int(value),␣
↪int(mobility)))
                self.nbb["b" + str(mobility)] = self.nbb.get("b" +␣
↪str(mobility), 0) + 1
            else:
                (varnum, vardom) = line.split()[:2]
                self.var.append((int(varnum), int(vardom)))

        stream = open(dom)
        for line in stream:
            domain = line.split()[:]
            self.dom[int(domain[0])] = [int(f) for f in domain[2:]]

        stream = open(ctr)
        for line in stream:
            (var1, var2, dummy, operand, deviation, weight) = line.split()[:6]
            self.ctr.append((int(var1), int(var2), operand, int(deviation),␣
↪int(weight)))
            self.nba["a" + str(weight)] = self.nba.get("a" + str(weight), 0) +␣
↪1

        stream = open(cst)
        for line in stream:
            if len(line.split()) == 3:
                (aorbi, eq, cost) = line.split()[:3]
                if (eq == "="):
                    self.cost[aorbi] = int(cost)
                    self.top += int(cost) * self.nba.get(aorbi, self.nbb.
↪get(aorbi, 0))
```

(continues on next page)

(continued from previous page)

```
def model(data):
    Var = {e[0]: "X" + str(e[0]) for e in data.var}
    Domain = {e[0]: e[1] for e in data.var}
    RLFAP = {
        "name": "RLFAP",
        "variables": [(Var[e[0]], ["f" + str(f) for f in data.dom[e[1]]]) for␣
↪e in data.var],
        "functions":
            [# hard and soft interference
             [{"scope": [Var[var1], Var[var2]], "costs": [0 if ((operand==">"␣
↪and abs(a - b)>deviation) or (operand=="=" and abs(a - b)==deviation)) else␣
↪data.cost.get("a"+str(weight),data.top) for a in data.dom[Domain[var1]] for␣
↪b in data.dom[Domain[var2]]]} for (var1, var2, operand, deviation, weight)␣
↪in data.ctr],
             # mobility costs
             [{"scope": [Var[e[0]]], "defaultcost": data.cost.get("b
↪"+str(e[3]),data.top), "costs": ["f" + str(e[2]), 0] if e[2] in data.
↪dom[e[1]] else []} for e in data.var if len(e)==4]
            ]
    }
    return RLFAP

if __name__ == '__main__':
    # read parameters
    if len(sys.argv) < 5: exit('Command line arguments are filenames: var.txt␣
↪dom.txt ctr.txt cst.txt')
    data = Data(sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4])
    # dump problem into JSON .cfn format for minimization
    cfn(model(data), True, data.top)
```

5.4 Frequency assignment problem with polarization

5.4.1 Brief description

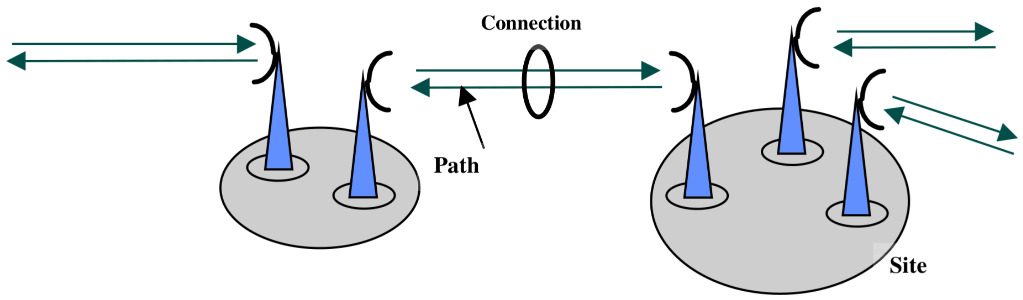
The previously-described *Radio link frequency assignment problem* has been extended to take into account polarization constraints and user-defined relaxation of electromagnetic compatibility constraints. The problem is to assign a pair (frequency,polarization) to every radio communication link (also called a path). Frequencies are integer values taken in finite domains. Polarizations are in {-1,1}. Constraints are :

- (I) two paths must use equal or different frequencies (*f_i=f_j* or *f_i<>f_j*),
- (II) the absolute difference between two frequencies should exactly be equal or different to a given number e (*|f_i-f_j|=e* or *|f_i-f_j|<>e*),
- (III) two paths must use equal or different polarizations (*p_i=p_j* or *p_i<>p_j*),
- (IV) the absolute difference between two frequencies should be greater at a relaxation level l (0 to 10) than a given number g_l (resp. d_l) if polarization are equal (resp. different) (*|f_i-f_j|>=g_l* if *p_i=p_j* else *|f_i-f_j|>=d_l*), with *g_(l-1)>g_l*, *d_(l-1)>d_l*, and usually *g_l>d_l*.

Constraints (I) to (III) are mandatory constraints, while constraints (IV) can be relaxed. The goal is to find a feasible assignment with the smallest relaxation level l and which minimizes the number of violations of

(IV) at lower levels. See ROADEF Challenge 2001.

Physical description and mathematical formulation



5.4.2 CFN model

In order to benefit from soft local consistencies on binary cost functions, we create a single variable to represent a pair (frequency,polarization) for every radio link.

5.4.3 Data

Original data files can be download from ROADEF or fapp. Their format is described here. You can try a small example `exemple1.in` (resp. `exemple2.in`) with optimum 523 at relaxation level 3 with 1 violation at level 2 and 3 below (resp. 13871 at level 7 with 1 violation at level 6 and 11 below). See ROADEF Challenge 2001 results.

5.4.4 Python model generator

The following code using python3 interpreter will generate the corresponding cost function network (e.g. “python3 fapp.py exemple1.in 3”). You can also compile `fappeval.c` using “gcc -o fappeval fappeval.c” and download `sol2fapp.awk` in order to evaluate the solutions (e.g., “python3 fapp.py exemple1.in 3 | toulbar2 –stdin=cfn -s=3 | awk -f ./sol2fapp.awk - exemple1”).

fapp.py

```
import sys

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["metric", "cost", "bounds", "vars1", "vars2", "nb_
↪states", "starts", "ends", "transitions", "nb_symbols", "nb_values", "start",
↪ "terminals", "non_terminals", "min", "max", "values", "defaultcost", "tuples
↪", "comparator", "to"]
```

(continues on next page)

(continued from previous page)

```
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(problem["variables"]):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        for j,x in enumerate(e.get("scope")):
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
        print('], ', end='')
        if e.get("type") is not None:
            print('"type:" %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
```

(continues on next page)

(continued from previous page)

```
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
                        print('"%s"' % c, end='')
                    else:
                        print('%s' % c, end='')
                print(']', end='')
            print('}', end='')
        print('}\n}')

class Data:
    def __init__(self, filename, k):
        self.var = list()
        self.dom = {}
        self.ctr = list()
        self.softeq = list()
        self.softne = list()
        self.nbsoft = 0
        self.top = 1
        self.cst = 0

        stream = open(filename)
        for line in stream:
            if len(line.split())==3 and line.split()[0]=="DM":
                (DM, dom, freq) = line.split()[:3]
                if self.dom.get(int(dom)) is None:
                    self.dom[int(dom)] = [int(freq)]
                else:
                    self.dom[int(dom)].append(int(freq))

            if len(line.split()) == 4 and line.split()[0]=="TR":
                (TR, route, dom, polarisation) = line.split()[:4]
                if int(polarisation) is 0:
                    self.var.append((int(route), [(f,-1) for f in self.
↪dom[int(dom)]] + [(f,1) for f in self.dom[int(dom)]]))
                if int(polarisation) is -1:
                    self.var.append((int(route), [(f,-1) for f in self.
↪dom[int(dom)]]))
                if int(polarisation) is 1:
                    self.var.append((int(route), [(f,1) for f in self.
↪dom[int(dom)]]))

            if len(line.split())==6 and line.split()[0]=="CI":
                (CI, route1, route2, vartype, operator, deviation) = line.
↪split()[:6]
                self.ctr.append((int(route1), int(route2), vartype, operator,␣
↪int(deviation)))

            if len(line.split())==14 and line.split()[0]=="CE":
                (CE, route1, route2, s0, s1, s2, s3, s4, s5, s6, s7, s8, s9,␣
↪s10) = line.split()[:14]
                self.softeq.append((int(route1), int(route2), [int(s) for s in␣
↪[s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10]]))
```

(continues on next page)

(continued from previous page)

```
                self.nbsoft += 1

            if len(line.split())==14 and line.split()[0]=="CD":
                (CD, route1, route2, s0, s1, s2, s3, s4, s5, s6, s7, s8, s9,␣
↪s10) = line.split()[:14]
                self.softne.append((int(route1), int(route2), [int(s) for s in␣
↪[s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10]]))
#                 self.nbsoft += 1

        self.cst = 10*k*self.nbsoft**2
        self.top += self.cst
        self.top += 10*self.nbsoft**2

def model(data, k):
    Var = {e[0]: "X" + str(e[0]) for e in data.var}
    Domain = {e[0]: e[1] for e in data.var}
    FAPP = {
        "name": "FAPP",
        "variables": [(Var[e[0]], ["f" + str(f) + "p" + str(p) for (f,p) in␣
↪e[1]]) for e in data.var],
        "functions":
            [# hard constraints
             [{"scope": [Var[route1], Var[route2]], "costs": [0 if ((operand==
↪"I" and abs((f1 if vartype=="F" else p1) - (f2 if vartype=="F" else p2)) !=␣
↪deviation)
                                                                        or␣
↪(operand=="E" and abs((f1 if vartype=="F" else p1) - (f2 if vartype=="F"␣
↪else p2)) == deviation)) else data.top
                                                              for (f1,p1) in␣
↪Domain[route1] for (f2,p2) in Domain[route2]]}
              for (route1, route2, vartype, operand, deviation) in data.ctr],
             # soft equality constraints
             [{"scope": [Var[route1], Var[route2]], "costs": [0 if p1!=p2 or␣
↪abs(f1 - f2) >= deviations[i] else (data.top if i>=k else (1 if i<k-1 else␣
↪10*data.nbsoft))
                                                              for (f1, p1) in␣
↪Domain[route1] for (f2, p2) in Domain[route2]]}
                for i in range(11) for (route1, route2, deviations) in data.
↪softeq],
             # soft inequality constraints
             [{"scope": [Var[route1], Var[route2]], "costs": [0 if p1==p2 or␣
↪abs(f1 - f2) >= deviations[i] else (data.top if i>=k else (1 if i<k-1 else␣
↪10*data.nbsoft))
                                                              for (f1, p1) in␣
↪Domain[route1] for (f2, p2) in Domain[route2]]}
                for i in range(11) for (route1, route2, deviations) in data.
↪softne],
             # constant cost to be added corresponding to the relaxation level
             {"scope": [], "defaultcost": data.cst, "costs": []}
             ]
    }
    return FAPP
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    # read parameters
    if len(sys.argv) < 2: exit('Command line argument is problem data filename␣
↪and relaxation level')
    k = int(sys.argv[2])
    data = Data(sys.argv[1], k)
    # dump problem into JSON .cfn format for minimization
    cfn(model(data, k), True, data.top)
```

5.5 Mendelian error detection problem

5.5.1 Brief description

The problem is to detect marker genotyping incompatibilities (Mendelian errors only) in complex pedigrees. The input is a pedigree data with partial observed genotyping data at a single locus. The problem is to assign genotypes (unordered pairs of alleles) to all individuals such that they are compatible with the Mendelian law of heredity and with the maximum number of genotyping data. Sanchez, M., de Givry, S. and Schiex, T. Constraints (2008) 13:130.

5.5.2 CFN model

We create N variables for every individual genotype with domain being all possible unordered pairs of existing alleles. Hard ternary cost functions express mendelian law of heredity. Unary cost functions are used to model potential genotyping errors.

5.5.3 Data

Original data files can be download from the cost function library pedigree. Their format is described here. You can try a small example simple.pre (`simple.pre`) with optimum value equal to 1.

5.5.4 Python model generator

The following code using python3 interpreter will generate the corresponding cost function network (e.g. “python3 mendel.py simple.pre”).

mendel.py

```
import sys

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
```

(continues on next page)

(continued from previous page)

```
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(problem["variables"]):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        for j,x in enumerate(e.get("scope")):
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
        print('], ', end='')
        if e.get("type") is not None:
            print('"type:" %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
```

(continues on next page)

(continued from previous page)

```
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
                        print('"%s"' % c, end='')
                    else:
                        print('%s' % c, end='')
                print(']', end='')
        print('}', end='')
    print('}\n}')

class Data:
    def __init__(self, ped):
        self.id = list()
        self.father = {}
        self.mother = {}
        self.alleles = {}
        self.freq = {}
        self.obs = 0

        stream = open(ped)
        for line in stream:
            (locus, id, father, mother, sex, allele1, allele2) = line.
↪split()[:]
            self.id.append(int(id))
            self.father[int(id)] = int(father)
            self.mother[int(id)] = int(mother)
            self.alleles[int(id)] = (int(allele1), int(allele2)) if␣
↪int(allele1) < int(allele2) else (int(allele2), int(allele1))
            if int(allele1) != 0: self.freq[int(allele1)] = self.freq.
↪get(int(allele1), 0) + 1
            if int(allele2) != 0: self.freq[int(allele2)] = self.freq.
↪get(int(allele2), 0) + 1
            if int(allele1) != 0 or int(allele2) != 0: self.obs += 1

def model(data, k):
    Var = {g: "g" + str(g) for g in data.id}
    Domain = ["a" + str(a1) + "a" + str(a2)  for a1 in data.freq for a2 in␣
↪data.freq if a1 <= a2]
    Mendel = {
        "name": "Mendel",
        "variables": [(Var[g], Domain) for g in data.id],
        "functions":
            [# mendelian law of heredity
             [{"scope": [Var[data.father[g]], Var[data.mother[g]], Var[g]],
```

(continues on next page)

(continued from previous page)

```
               "costs": [0 if (a1 in (p1,p2) and a2 in (m1,m2)) or (a2 in (p1,
↪p2) and a1 in (m1,m2)) else k
                         for p1 in data.freq for p2 in data.freq
                         for m1 in data.freq for m2 in data.freq
                         for a1 in data.freq for a2 in data.freq if p1 <= p2␣
↪and m1 <= m2 and a1 <= a2]}
              for g in data.id if data.father.get(g, 0) != 0 and data.mother.
↪get(g, 0) != 0],
             # observation costs
             [{"scope": [Var[g]],
               "costs": [0 if (a1,a2) == data.alleles[g] else 1 for a1 in data.
↪freq for a2 in data.freq if a1 <= a2]}
              for g in data.id if data.alleles[g][0] != 0 and data.
↪alleles[g][1] != 0]
            ]
    }
    return Mendel

if __name__ == '__main__':
    # read parameters
    if len(sys.argv) < 2: exit('Command line arguments are PEDFILE filename:␣
↪simple.pre')
    data = Data(sys.argv[1])
    # dump problem into JSON .cfn format for minimization
    cfn(model(data, data.obs + 1), True, data.obs + 1)
```

5.6 Block modeling problem

5.6.1 Brief description

This is a clustering problem, occuring in social network analysis. The problem is to divide a given graph G into k clusters such that the interactions between clusters can be summarized by a k*k 0/1 matrix M: if M[i,j]=1 then all the nodes in cluster i should be connected to all the nodes in cluster j in G, else if M[i,j]=0 then there should be no edge between the nodes in G. The goal is to find a k-clustering and the associated matrix M minimizing the number of erroneous edges. A Mattenet, I Davidson, S Nijssen, P Schaus. Generic Constraint-Based Block Modeling Using Constraint Programming. CP 2019, pp656-673, Stamford, CT, USA.

5.6.2 CFN model

We create N variables for every node of the graph with domain size k. We add k*k Boolean variables for representing M. For all triplets of two nodes u, v, and one matrix cell M[i,j], we have a ternary cost function which returns a cost of 1 if node u is assigned to cluster i, v to j, and M[i,j]=1 but (u,v) is not in G, or M[i,j]=0 and (u,v) in G. In order to break symmetries, we constrain the first k-1 node variables to be assigned to cluster index less than or equal to their index

5.6.3 Data

You can try a small example `simple.mat` with optimum value equal to 0 for 3 clusters.

Perfect solution for the small example with k=3 (Mattenet et al, CP 2019)

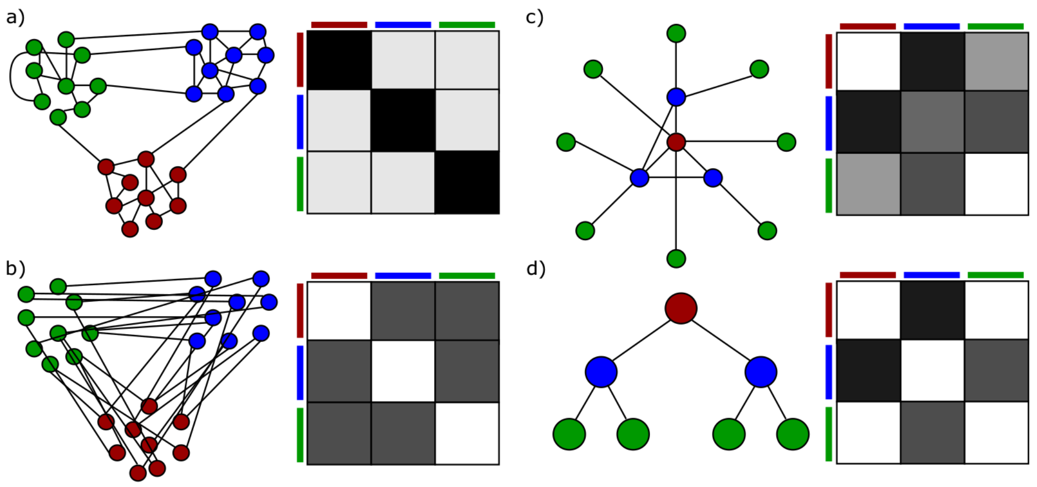
$$\begin{tikzcd}
1 \arrow[r] \arrow[dr] & 3 \arrow[r] \arrow[loop above] \arrow[d, leftrightarrow] & 5 \\
2 \arrow[r] \arrow[ur] & 4 \arrow[ur] \arrow[loop right] &
\end{tikzcd}$$

$$G = \left(\begin{array}{cc|cc|c}
\mathbf{0} & \mathbf{0} & 1 & 1 & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & 1 & 1 & \mathbf{0} \\
\hline
\mathbf{0} & \mathbf{0} & 1 & 1 & 1 \\
\mathbf{0} & \mathbf{0} & 1 & 1 & 1 \\
\hline
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}
\end{array}\right)$$

$$\begin{tikzcd}
\{1,2\} \arrow[r] & \{3,4\} \arrow[r] \arrow[loop above] & \{5\}
\end{tikzcd}$$

$$M = \begin{pmatrix}
\mathbf{0} & 1 & \mathbf{0} \\
\mathbf{0} & 1 & 1 \\
\mathbf{0} & \mathbf{0} & \mathbf{0}
\end{pmatrix}$$

More examples with 3 clusters (Stochastic Block Models [Funke and Becker, Plos One 2019])



See other examples, such as PoliticalActor and more, here : `100.mat` | `150.mat` | `200.mat` | `30.mat` | `50.mat` | `hartford_drug.mat` | `kansas.mat` | `politicalactor.mat` | `sharpstone.mat` | `transatlantic.mat`.

5.6.4 Python model generator

The following code using python3 interpreter will generate the corresponding cost function network (e.g. “python3 blockmodel.py simple.mat 3”). Download the AWK script `sol2block.awk` to pretty print the results (e.g., “python3 blockmodel.py simple.mat 3 | toulbar2 –stdin=cfn -s=3 | awk -f ./sol2block.awk”).

`blockmodel.py`

```
import sys

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(flatten(problem["variables"])):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        scope = {}
        for j,x in enumerate(e.get("scope")):
            if (x in scope): sys.exit(str(e) + '\nError: scope of function ' +␣
↪str(i) + ' with the same variable twice is forbidden!')
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
```

(continues on next page)

(continued from previous page)

```
            scope[x]=j
        print('], ', end='')
        if e.get("type") is not None:
            print('"type:" %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
                        print('"%s"' % c, end='')
                    else:
                        print('%s' % c, end='')
                print(']', end='')
        print('}', end='')
    print('}\n}')

class Data:
    def __init__(self, filename, k):
        lines = open(filename).readlines()
        self.n = len(lines)
        self.matrix = [[int(e) for e in l.split(' ')] for l in lines]
        self.top = 1 + self.n*self.n

def model(data, K):
    Var = [(chr(65 + i) if data.n < 28 else "x" + str(i)) for i in range(data.
↪n)] # Political actor or any instance
#    Var = ["ron","tom","frank","boyd","tim","john","jeff","jay","sandy","jerry
↪","darrin","ben","arnie"] # Transatlantic
#    Var = ["justin","harry","whit","brian","paul","ian","mike","jim","dan",
↪"ray","cliff","mason","roy"] # Sharpstone
```

(continues on next page)

(continued from previous page)

```
#    Var = ["Sherrif","CivilDef","Coroner","Attorney","HighwayP","ParksRes",
↪"GameFish","KansasDOT","ArmyCorps","ArmyReserve","CrableAmb","FrankCoAmb",
↪"LeeRescue","Shawney","BurlPolice","LyndPolice","RedCross","TopekaFD","CarbFD
↪","TopekaRBW"] # Kansas
    BlockModeling = {
        "name": "BlockModel_N" + str(data.n) + "_K" + str(K),
        "variables": [[("M_" + str(u) + "_" + str(v), 2) for u in range(K) for␣
↪v in range(K)],
                      [(Var[i], K)  for i in range(data.n)]],
        "functions":
            [
                # objective function
                [{"scope": ["M_" + str(u) + "_" + str(v), Var[i], Var[j]],
                  "costs": [1 if (u == k and v == l and data.matrix[i][j] != m)
                            else 0
                            for m in range(2)
                            for k in range(K)
                            for l in range(K)]}
                 for u in range(K) for v in range(K) for i in range(data.n)␣
↪for j in range(data.n) if i != j],

                # self-loops
                [{"scope": ["M_" + str(u) + "_" + str(u), Var[i]],
                  "costs": [1 if (u == k and data.matrix[i][i] != m)
                            else 0
                            for m in range(2)
                            for k in range(K)]}
                 for u in range(K) for i in range(data.n)],

                # breaking partial symmetries by fixing first (K-1) domain␣
↪variables to be assigned to cluster less than or equal to their index
                [{"scope": [Var[l]],
                 "costs": [data.top if k > l else 0 for k in range(K)]}
                 for l in range(K-1)]
            ]
    }
    return BlockModeling

if __name__ == '__main__':
    # read parameters
    if len(sys.argv) < 2: exit('Command line argument is problem data filename␣
↪and number of blocks')
    K = int(sys.argv[2])
    data = Data(sys.argv[1], K)
    # dump problem into JSON .cfn format for minimization by toulbar2 solver
    cfn(model(data, K), True, data.top)
```

We improve the previous model by sorting node variables by decreasing out degree and removing the lower triangular matrix of M if the input graph is undirected (symmetric adjacency matrix).

`blockmodel2.py`

```
import sys

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(flatten(problem["variables"])):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        scope = {}
        for j,x in enumerate(e.get("scope")):
            if (x in scope): sys.exit(str(e) + '\nError: scope of function ' +␣
↪str(i) + ' with the same variable twice is forbidden!')
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
            scope[x]=j
        print('], ', end='')
        if e.get("type") is not None:
            print('"type:" %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
```

(continues on next page)

(continued from previous page)

```
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
                        print('"%s"' % c, end='')
                    else:
                        print('%s' % c, end='')
                print(']', end='')
        print('}', end='')
    print('}\n}')

class Data:
    def __init__(self, filename, k):
        lines = open(filename).readlines()
        self.n = len(lines)
        self.matrix = [[int(e) for e in l.split(' ')] for l in lines]
        self.top = 1 + self.n*self.n

def model(data, K):
    symmetric = all([data.matrix[i][j] == data.matrix[j][i] for i in␣
↪range(data.n) for j in range(data.n) if j>i])
    Var = [(chr(65 + i) if data.n < 28 else "x" + str(i)) for i in range(data.
↪n)]

    # sort node variables by decreasing out degree
    degree = [(i, sum(data.matrix[i])) for i in range(data.n)]
    degree.sort(key=lambda tup: -tup[1])
    indexes = [e[0] for e in degree]

    BlockModeling = {
        "name": "BlockModel_N" + str(data.n) + "_K" + str(K) + "_Sym" +␣
↪str(symmetric),
```

(continues on next page)

(continued from previous page)

```
        # order node variables before matrix M variables
        # order matrix M variables starting from the main diagonal and moving␣
↪away progressively
        # if input graph is symmetric then keep only the upper triangular␣
↪matrix of M
        "variables": [[("M_" + str(u) + "_" + str(u), 2) for u in range(K)],
                      [("M_" + str(u) + "_" + str(v), 2) for d in range(K) for␣
↪u in range(K) for v in range(K)
                       if u != v and (not symmetric or u < v) and abs(u - v)␣
↪== d],
                      [(Var[indexes[i]], K)  for i in range(data.n)]],
        "functions":
            [
                # objective function
                # if input graph is symmetric then cost tables are also␣
↪symmetric wrt node variables
                [{"scope": ["M_" + str(u) + "_" + str(v), Var[indexes[i]],␣
↪Var[indexes[j]]],
                  "costs": [1 if (((u == k and v == l) or (symmetric and u ==␣
↪l and v == k))
                                   and data.matrix[indexes[i]][indexes[j]] != m)
                            else 0
                            for m in range(2)
                            for k in range(K)
                            for l in range(K)]}
                 for u in range(K) for v in range(K) for i in range(data.n)␣
↪for j in range(data.n)
                 if i != j and (not symmetric or u <= v)],

                # self-loops
                [{"scope": ["M_" + str(u) + "_" + str(u), Var[indexes[i]]],
                  "costs": [1 if (u == k and data.
↪matrix[indexes[i]][indexes[i]] != m)
                            else 0
                            for m in range(2)
                            for k in range(K)]}
                 for u in range(K) for i in range(data.n)],

                # breaking partial symmetries by fixing first (K-1) domain␣
↪variables to be assigned to cluster less than or equal to their index
                [{"scope": [Var[indexes[l]]],
                  "costs": [data.top if k > l else 0 for k in range(K)]}
                 for l in range(K-1)]
            ]
    }
    return BlockModeling

if __name__ == '__main__':
    # read parameters
    if len(sys.argv) < 2: exit('Command line argument is problem data filename␣
↪and number of blocks')
    K = int(sys.argv[2])
```

(continues on next page)

(continued from previous page)

```
    data = Data(sys.argv[1], K)
    # dump problem into JSON .cfn format for minimization by toulbar2 solver
    cfn(model(data, K), True, data.top)
```

5.7 Airplane landing problem

5.7.1 Brief description (from CHOCO-SOLVER)

Given a set of planes and runways, the objective is to minimize the total weighted deviation from the target landing time for each plane. We consider only a single runway. There are costs associated with landing either earlier or later than a target landing time for each plane. Each plane has to land within its predetermined time window such that separation times between all pairs of planes are satisfied. J.E. Beasley, M. Krishnamoorthy, Y.M. Sharaiha and D. Abramson. Scheduling aircraft landings - the static case. Transportation Science, vol.34, 2000.

5.7.2 CFN model

We create N variables for every plane landing time. Binary cost functions express separation times between pairs of planes. Unary cost functions represent the weighted deviation for each plane.

5.7.3 Data

Original data files can be download from the cost function library airland. Their format is described here. You can try a small example `airland1.txt` with optimum value equal to 700.

5.7.4 Python model and solve

The following code uses the pytoulbar2 module to generate the cost function network and solve it (e.g. "python3 airland.py airland1.txt"). Compile toulbar2 with "cmake -DPYTB2=ON . ; make" and copy the resulting module in pytoulbar2 folder "cp lib/Linux/pytb2.cpython* pytoulbar2".

`airland.py`

```

import sys
import pytoulbar2

f = open(sys.argv[1], 'r').readlines()

tokens = []
for l in f:
    tokens += l.split()

pos = 0

def token():
   global pos, tokens
```

(continues on next page)

(continued from previous page)

```
    if (pos == len(tokens)):
        return None
    s = tokens[pos]
    pos += 1
    return int(float(s))

N = token()
token() # skip freeze time

LT = []
PC = []
ST = []

for i in range(N):
    token()  # skip appearance time
# Times per plane: {earliest landing time, target landing time, latest landing␣
↪time}
    LT.append([token(), token(), token()])

# Penalty cost per unit of time per plane:
# [for landing before target, after target]
    PC.append([token(), token()])

# Separation time required after i lands before j can land
    ST.append([token() for j in range(N)])

top = 99999

Problem = pytoulbar2.CFN(top)
for i in range(N):
    Problem.AddVariable('x' + str(i), range(LT[i][0],LT[i][2]+1))

for i in range(N):
    Problem.AddFunction([i], [PC[i][0]*abs(a-LT[i][1]) for a in range(LT[i][0],
↪ LT[i][2]+1)])

for i in range(N):
    for j in range(i+1,N):
        Problem.AddFunction([i, j], [top*(a+ST[i][j]>b and b+ST[j][i]>a) for a␣
↪in range(LT[i][0], LT[i][2]+1) for b in range(LT[j][0], LT[j][2]+1)])

Problem.Dump('airplane.cfn')
Problem.NoPreprocessing()
Problem.Solve()
```

5.8 Warehouse location problem

5.8.1 Brief description

See a problem description in CSPLib-034. We are dealing with the uncapacitated case only for the moment.

5.8.2 CFN model

We create Boolean variables for the warehouses (i.e., open or not) and integer variables for the stores (with domain size the number of warehouses). Channeling constraints link both of them. The objective function is linear and decomposed into one unary cost function per variable (maintenance and supply costs).

5.8.3 Data

Original data files can be download from the cost function library warehouses. Their format is described here.

5.8.4 Python model generator

The following code using python3 interpreter will generate the corresponding cost function network with a user given floating-point precision (e.g. "python3 warehouse.py cap44.txt 5").

warehouse.py

```
import sys

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(flatten(problem["variables"])):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
```

(continues on next page)

(continued from previous page)

```
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        scope = {}
        for j,x in enumerate(e.get("scope")):
            if (x in scope): sys.exit(str(e) + '\nError: scope of function ' +␣
↪str(i) + ' with the same variable twice is forbidden!')
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
            scope[x]=j
        print('], ', end='')
        if e.get("type") is not None:
            print('"type": %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
```

(continues on next page)

(continued from previous page)

```
                        print('"%s"' % c, end='')
                    else:
                        print('%s' % c, end='')
                print(']', end='')
            print('}', end='')
        print('}\n}')

class Data:
    def __init__(self, filename):
        lines = open(filename).readlines()
        tokens = flatten([[e for e in l.split()] for l in lines])
        p = 0
        self.n = int(tokens[p])
        p += 1
        self.m = int(tokens[p])
        p += 1
        self.top = 1.  # sum of all costs plus one
        self.CostW = []  # maintenance cost of warehouses
        self.Capacity = []  # capacity limit of warehouses (not used)
        for i in range(self.n):
            self.Capacity.append(int(tokens[p]))
            p += 1
            self.CostW.append(float(tokens[p]))
            p += 1
        self.top += sum(self.CostW)
        self.Demand = []  # demand for each store (not used)
        self.CostS = []  # supply cost matrix
        for j in range(self.m):
            self.Demand.append(int(tokens[p]))
            p += 1
            self.CostS.append([])
            for i in range(self.n):
                self.CostS[j].append(float(tokens[p]))
                p += 1
            self.top += sum(self.CostS[-1])

def model(data):
    Warehouse = ["w" + str(i) for i in range(data.n)]
    Store = ["s" + str(i) for i in range(data.m)]
    Model = {
        "name": "Warehouse_" + str(data.n) + "_" + str(data.m),
        "variables": [[(e, 2) for e in Warehouse],
                      [(e, data.n)  for e in Store]],
        "functions":
            [
                # maintenance costs
                [{"scope": [Warehouse[i]],
                  "costs": [0, data.CostW[i]]}
                 for i in range(data.n)],
                # supply costs
                [{"scope": [Store[i]],
                  "costs": data.CostS[i]}
```

(continues on next page)

(continued from previous page)

```
                 for i in range(data.m)],
                # channeling constraints between warehouses and stores
                [{"scope": [Warehouse[i], Store[j]],
                  "costs": [(data.top if (a == 0 and b == i) else 0) for a in␣
↪range(2) for b in range(data.n)]}
                 for i in range(data.n) for j in range(data.m)]
            ]
    }
    return Model

if __name__ == '__main__':
    # read parameters
    if len(sys.argv) < 2: exit('Command line argument is problem data filename␣
↪and number of precision digits after the floating point')
    data = Data(sys.argv[1])
    # dump problem into JSON .cfn format for minimization
    cfn(model(data), True, data.top, int(sys.argv[2]))
```

5.8.5 Python model and solve using pytoulbar2

The following code uses the pytoulbar2 module to generate the cost function network and solve it (e.g. “python3 warehouse2.py cap44.txt 1” found optimum value equal to 10349757). Other instances are available here in cfn format. Compile toulbar2 with “cmake -DPYTB2=ON . ; make” and copy the resulting module in pytoulbar2 folder “cp lib/Linux/pytb2.cpython* pytoulbar2”.

warehouse2.py

```

import sys
import pytoulbar2

f = open(sys.argv[1], 'r').readlines()

precision = int(sys.argv[2])  # used to convert cost values from float to␣
↪integer

tokens = []
for l in f:
    tokens += l.split()

pos = 0


def token():
    global pos, tokens
    if pos == len(tokens):
        return None
    s = tokens[pos]
    pos += 1
    return s
```

(continues on next page)

(continued from previous page)

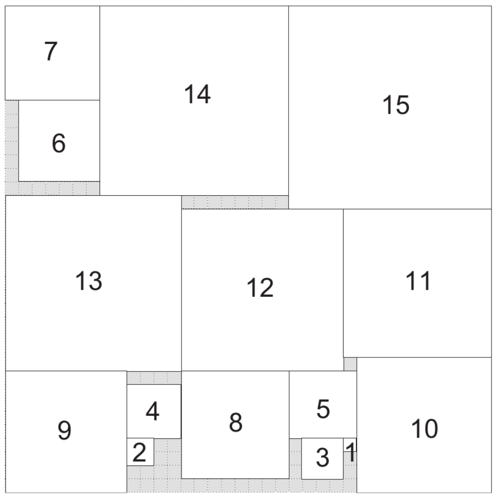
```
N = int(token())  # number of warehouses
M = int(token())  # number of stores

top = 1  # sum of all costs plus one

CostW = []  # maintenance cost of warehouses
Capacity = []  # capacity limit of warehouses (not used)

for i in range(N):
    Capacity.append(token())
    CostW.append(int(float(token()) * 10.**precision))

top += sum(CostW)

Demand = []  # demand for each store
CostS = [[] for i in range(M)]  # supply cost matrix

for j in range(M):
    Demand.append(int(token()))
    for i in range(N):
        CostS[j].append(int(float(token()) * 10.**precision))
    top += sum(CostS[-1])

# create a new empty cost function network
Problem = pytoulbar2.CFN(top)
# add warehouse variables
for i in range(N):
    Problem.AddVariable('w' + str(i), range(2))
# add store variables
for j in range(M):
    Problem.AddVariable('s' + str(j), range(N))
# add maintenance costs
for i in range(N):
    Problem.AddFunction([i], [0, CostW[i]])
# add supply costs for each store
for j in range(M):
    Problem.AddFunction([N+j], CostS[j])
# add channeling constraints between warehouses and stores
for i in range(N):
    for j in range(M):
        Problem.AddFunction([i, N+j], [(top if (a == 0 and b == i) else 0) for␣
↪a in range(2) for b in range(N)])

Problem.Dump('warehouse.cfn')
Problem.Option.FullEAC = False
Problem.Option.showSolutions = False
Problem.Solve()
```

5.9 Square packing problem

5.9.1 Brief description

Find a packing of squares of size 1×1, 2×2,. . . , NxN into a given container square SxS without overlaps. See a problem description in CSPLib-009. Results up to N=56 are given here.

Optimal solution for 15 squares packed into a 36x36 square (Fig. taken from Takehide Soh)



5.9.2 CFN model

We create an integer variable of domain size (S-i)x(S-i) for each square i in [0,N-1] of size i+1 representing its top-left position in the container. Its value modulo (S-i) gives the x-coordinate, whereas its value divided by (S-i) gives the y-coordinate. We have binary constraints to forbid any overlapping pair of squares. We make the problem a pure satisfaction problem by fixing S. The initial upper bound is 1.

5.9.3 Python model generator

The following code using python3 interpreter will generate the corresponding cost function network (e.g. “python3 square.py 3 5”).

square.py

```
import sys
from random import seed
seed(123456789)

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
```

(continues on next page)

(continued from previous page)

```
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(flatten(problem["variables"])):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        scope = {}
        for j,x in enumerate(e.get("scope")):
            if (x in scope): sys.exit(str(e) + '\nError: scope of function ' +␣
↪str(i) + ' with the same variable twice is forbidden!')
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
            scope[x]=j
        print('], ', end='')
        if e.get("type") is not None:
            print('"type": %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
```

(continues on next page)

(continued from previous page)

```
                        first = False
                    print ('}', end='')
                else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
            if e.get("defaultcost") is not None:
                print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
            if e.get("costs") is not None:
                print('"costs": ', end='')
                if isinstance(e.get("costs"), str):
                    print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
                else:
                    print('[', end='')
                    for j,c in enumerate(e.get("costs")):
                        if j > 0: print(', ', end='')
                        if isinstance(c, str) and not c.isdigit():
                            print('"%s"' % c, end='')
                        else:
                            print('%s' % c, end='')
                    print(']', end='')
            print('}', end='')
        print('}\n}')

def model(N, S, top):
    Var = ["sq" + str(i+1) for i in range(N)]
    Model = {
        "name": "SquarePacking" + str(N) + "_" + str(S),
        "variables": [(Var[i], (S-i)*(S-i)) for i in range(N)],
        "functions":
            [
             # no overlapping constraint
             [{"scope": [Var[i], Var[j]], "costs": [(0 if ((a%(S-i)) + i + 1
↪<= (b%(S-j))) or ((b%(S-j)) + j + 1 <= (a%(S-i))) or (int(a/(S-i)) + i + 1
↪<= int(b/(S-j))) or (int(b/(S-j)) + j + 1 <= int(a/(S-i))) else top) for a␣
↪in range((S-i)*(S-i)) for b in range((S-j)*(S-j))]} for i in range(N) for j␣
↪in range(N) if (i < j)]
            ]
         }
    return Model

if __name__ == '__main__':
    # read parameters
    N = int(sys.argv[1])
    S = int(sys.argv[2])
    # infinite cost
    top = 1
    # dump problem into JSON .cfn format for minimization
    cfn(model(N, S, top), True, top)
```

5.9.4 Python model and solve using pytoulbar2

The following code uses the pytoulbar2 module to generate the cost function network and solve it (e.g. “python3 square2.py 3 5”). Compile toulbar2 with “cmake -DPYTB2=ON . ; make” and copy the resulting module in pytoulbar2 folder “cp lib/Linux/pytb2.cpython* pytoulbar2”.

square2.py

```
import sys
from random import seed
seed(123456789)

import pytoulbar2

N = int(sys.argv[1])
S = int(sys.argv[2])

top = 1

Problem = pytoulbar2.CFN(top)

for i in range(N):
    Problem.AddVariable('sq' + str(i+1), range((S-i)*(S-i)))

for i in range(N):
    for j in range(i+1,N):
        Problem.AddFunction([i, j], [0 if ((a%(S-i)) + i + 1 <= (b%(S-j))) or␣
↪((b%(S-j)) + j + 1 <= (a%(S-i))) or (int(a/(S-i)) + i + 1 <= int(b/(S-j)))␣
↪or (int(b/(S-j)) + j + 1 <= int(a/(S-i))) else top for a in range((S-i)*(S-
↪i)) for b in range((S-j)*(S-j))])

#Problem.Dump('square.cfn')
Problem.Option.FullEAC = False
Problem.Option.showSolutions = True
Problem.Solve()
```

5.9.5 C++ program using libtb2.so

The following code uses the C++ toulbar2 library libtb2.so. Compile toulbar2 with “cmake -DLIBTB2=ON -DPYTB2=ON . ; make” and copy the library in your current directory “cp lib/Linux/libtb2.so .” before compiling “g++ -o square square.cpp -Isrc -Llib/Linux -std=c++11 -O3 -DNDEBUG -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -DWCSPFORMATONLY libtb2.so” and running the example (e.g. “./square 15 36”).

square.cpp

```

/**
 * Square Packing Problem
 */

// Compile with cmake option -DLIBTB2=ON -DPYTB2=ON to get C++ toulbar2␣
↪library lib/Linux/libtb2.so
```

(continues on next page)

(continued from previous page)

```
// Then,
// g++ -o square square.cpp -Isrc -Llib/Linux -std=c++11 -O3 -DNDEBUG -DBOOST -
↪DLONGDOUBLE_PROB -DLONGLONG_COST -DWCSPFORMATONLY libtb2.so

#include "toulbar2lib.hpp"

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

int main(int argc, char* argv[])
{
    int N = atoi(argv[1]);
    int S = atoi(argv[2]);

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

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

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

    Cost top = UNIT_COST;
    WeightedCSPSolver* solver = WeightedCSPSolver::makeWeightedCSPSolver(top);

    for (int i=0; i<N; i++) {
        solver->getWCSP()->makeEnumeratedVariable("sq" + to_string(i+1), 0, (S-
↪i)*(S-i) - 1);
    }

    for (int i=0; i<N; i++) {
        for (int j=i+1; j<N; j++) {
            vector<Cost> costs((S-i)*(S-i)*(S-j)*(S-j), MIN_COST);
                for (int a=0; a<(S-i)*(S-i); a++) {
                    for (int b=0; b<(S-j)*(S-j); b++) {
                    costs[a*(S-j)*(S-j)+b] = ((((a%(S-i)) + i + 1 <= (b%(S-
↪j))) || ((b%(S-j)) + j + 1 <= (a%(S-i))) || ((a/(S-i)) + i + 1 <= (b/(S-j)))␣
↪|| ((b/(S-j)) + j + 1 <= (a/(S-i))))?MIN_COST:top);
                }
            }
            solver->getWCSP()->postBinaryConstraint(i, j, costs);
        }
    }

    solver->getWCSP()->sortConstraints(); // must be done at the end of the␣
↪modeling

    tb2checkOptions();
    if (solver->solve()) {
```

(continues on next page)

(continued from previous page)

```
            vector<Value> sol;
            solver->getSolution(sol);
                for (int y=0; y<S; y++) {
                for (int x=0; x<S; x++) {
                    char c = ' ';
                    for (int i=0; i<N; i++) {
                        if (x >= (sol[i]%(S-i)) && x < (sol[i]%(S-i) ) + i + 1␣
↪&& y >= (sol[i]/(S-i)) && y < (sol[i]/(S-i)) + i + 1) {
                            c = 65+i;
                            break;
                        }
                     }
                     cout << c;
                }
                cout << endl;
            }
    } else {
            cout << "No solution found!" << endl;
    }

    delete solver;
    return 0;
}
```

5.10 Square soft packing problem

5.10.1 Brief description

Find a packing of squares of size 1×1, 2×2,. . . , NxN into a given container square SxS minimizing total sum of overlaps.

5.10.2 CFN model

We reuse the *Square packing problem* model except that binary constraints are replaced by cost functions returning the overlapping size or zero if no overlaps. The initial upper bound is a worst-case upper estimation of total sum of overlaps.

5.10.3 Python model generator

The following code using python3 interpreter will generate the corresponding cost function network (e.g. "python3 squaresoft 10 20").

squaresoft.py

```
import sys
from random import seed
```

(continues on next page)

(continued from previous page)

```
seed(123456789)

def flatten(x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, str) and not␣
↪isinstance(el, tuple) and not isinstance(el, dict):
            result.extend(flatten(el))
        else:
            result.append(el)
    return result

def cfn(problem, isMinimization, initPrimalBound, floatPrecision=0):
    globals_key_order = ["rhs", "capacity", "weights", "weightedvalues",
↪"metric", "cost", "bounds", "vars1", "vars2", "nb_states", "starts", "ends",
↪"transitions", "nb_symbols", "nb_values", "start", "terminals", "non_
↪terminals", "min", "max", "values", "defaultcost", "tuples", "comparator",
↪"to"]
    print('{')
    print('\tproblem: { "name": "%s", "mustbe": "%s%.*f" },' % (problem["name
↪"], "<" if (isMinimization) else ">", floatPrecision, initPrimalBound))
    print('\tvariables: {', end='')
    for i,e in enumerate(flatten(problem["variables"])):
        if i > 0: print(', ', end='')
        print('"%s":' % e[0], end='')
        if isinstance(e[1], int):
            print(' %s' % e[1], end='')
        else:
            print('[', end='')
            for j,a in enumerate(e[1]):
                if j > 0: print(', ', end='')
                print('"%s"' % a, end='')
            print(']', end='')
    print('},')
    print( '\tfunctions: {')
    for i,e in enumerate(flatten(problem["functions"])):
        if i > 0: print(',')
        if e.get("name") is not None: print('\t\t"%s": {scope: [' % e.get("name
↪"), end='')
        else: print('\t\t{scope: [', end='')
        scope = {}
        for j,x in enumerate(e.get("scope")):
            if (x in scope): sys.exit(str(e) + '\nError: scope of function ' +␣
↪str(i) + ' with the same variable twice is forbidden!')
            if j > 0: print(', ', end='')
            print('"%s"' % x, end='')
            scope[x]=j
        print('], ', end='')
        if e.get("type") is not None:
            print('"type": %s, ' % e.get("type"), end='')
        if e.get("params") is not None:
            if isinstance(e.get("params"), dict):
```

(continues on next page)

(continued from previous page)

```
                print('"params": {', end='')
                first = True
                for key in globals_key_order:
                    if key in e.get("params"):
                        if not first: print(', ', end='')
                        if isinstance(e.get("params")[key], str): print('"%s":
↪"%s"' % (str(key),str(e.get("params")[key]).replace("'", '"')), end='')
                        else: print('"%s": %s' % (str(key),str(e.get("params
↪")[key]).replace("'", '"')), end='')
                        first = False
                print ('}', end='')
            else: print('"params": %s, ' % str(e.get("params")).replace("'",'"
↪'), end='')
        if e.get("defaultcost") is not None:
            print('"defaultcost:" %s, ' % e.get("defaultcost"), end='')
        if e.get("costs") is not None:
            print('"costs": ', end='')
            if isinstance(e.get("costs"), str):
                print('"%s"' % e.get("costs"), end='') # reuse future cost␣
↪function by giving its name here
            else:
                print('[', end='')
                for j,c in enumerate(e.get("costs")):
                    if j > 0: print(', ', end='')
                    if isinstance(c, str) and not c.isdigit():
                        print('"%s"' % c, end='')
                    else:
                        print('%s' % c, end='')
                print(']', end='')
        print('}', end='')
    print('}\n}')

def model(N, S, top):
    Var = ["sq" + str(i+1) for i in range(N)]
    Model = {
        "name": "SquarePacking" + str(N) + "_" + str(S),
        "variables": [(Var[i], (S-i)*(S-i)) for i in range(N)],
        "functions":
            [
             # no overlapping constraint
             [{"scope": [Var[i], Var[j]], "costs": [(0 if ((a%(S-i)) + i + 1
↪<= (b%(S-j))) or ((b%(S-j)) + j + 1 <= (a%(S-i))) or (int(a/(S-i)) + i + 1
↪<= int(b/(S-j))) or (int(b/(S-j)) + j + 1 <= int(a/(S-i))) else min((a%(S-
↪i)) + i + 1 - (b%(S-j)), (b%(S-j)) + j + 1 - (a%(S-i))) * min(int(a/(S-i)) +␣
↪i + 1 - int(b/(S-j)), int(b/(S-j)) + j + 1 - int(a/(S-i)))) for a in␣
↪range((S-i)*(S-i)) for b in range((S-j)*(S-j))]} for i in range(N) for j in␣
↪range(N) if (i < j)]
            ]
         }
    return Model

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    # read parameters
    N = int(sys.argv[1])
    S = int(sys.argv[2])
    # infinite cost
    top = int((N*N*(N-1)*(2*N-1))/6 + 1)
    # dump problem into JSON .cfn format for minimization
    cfn(model(N, S, top), True, top)
```

5.10.4 C++ program using libtb2.so

The following code uses the C++ toulbar2 library libtb2.so. Compile toulbar2 with "cmake -DLIBTB2=ON -DPYTB2=ON . ; make" and copy the library in your current directory "cp lib/Linux/libtb2.so ." before compiling "g++ -o squaresoft squaresoft.cpp -Isrc -Llib/Linux -std=c++11 -O3 -DNDEBUG -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -DWCSPFORMATONLY libtb2.so" and running the example (e.g. "./squaresoft 10 20").

squaresoft.cpp

```

/**
 * Square Soft Packing Problem
 */

// Compile with cmake option -DLIBTB2=ON -DPYTB2=ON to get C++ toulbar2␣
↪library lib/Linux/libtb2.so
// Then,
// g++ -o squaresoft squaresoft.cpp -Isrc -Llib/Linux -std=c++11 -O3 -DNDEBUG -
↪DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -DWCSPFORMATONLY libtb2.so

#include "toulbar2lib.hpp"

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

int main(int argc, char* argv[])
{
    int N = atoi(argv[1]);
    int S = atoi(argv[2]);

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

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

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

(continues on next page)

(continued from previous page)

```
    Cost top = N*(N*(N-1)*(2*N-1))/6 + 1;
    WeightedCSPSolver* solver = WeightedCSPSolver::makeWeightedCSPSolver(top);

    for (int i=0; i < N; i++) {
        solver->getWCSP()->makeEnumeratedVariable("sq" + to_string(i+1), 0, (S-
↪i)*(S-i) - 1);
    }

    for (int i=0; i < N; i++) {
        for (int j=i+1; j < N; j++) {
            vector<Cost> costs((S-i)*(S-i)*(S-j)*(S-j), MIN_COST);
                for (int a=0; a < (S-i)*(S-i); a++) {
                    for (int b=0; b < (S-j)*(S-j); b++) {
                    costs[a*(S-j)*(S-j)+b] = ((((a%(S-i)) + i + 1 <= (b%(S-
↪j))) || ((b%(S-j)) + j + 1 <= (a%(S-i))) || ((a/(S-i)) + i + 1 <= (b/(S-j)))␣
↪|| ((b/(S-j)) + j + 1 <= (a/(S-i))))?MIN_COST:(min((a%(S-i)) + i + 1 - (b%(S-
↪j)), (b%(S-j)) + j + 1 - (a%(S-i))) * min((a/(S-i)) + i + 1 - (b/(S-j)), (b/
↪(S-j)) + j + 1 - (a/(S-i)))));
                }
            }
            solver->getWCSP()->postBinaryConstraint(i, j, costs);
        }
    }

    solver->getWCSP()->sortConstraints(); // must be done at the end of the␣
↪modeling

    tb2checkOptions();
    if (solver->solve()) {
            vector<Value> sol;
            solver->getSolution(sol);
                for (int y=0; y < S; y++) {
                for (int x=0; x < S; x++) {
                    char c = ' ';
                    for (int i=N-1; i >= 0; i--) {
                        if (x >= (sol[i]%(S-i)) && x < (sol[i]%(S-i) ) + i + 1␣
↪&& y >= (sol[i]/(S-i)) && y < (sol[i]/(S-i)) + i + 1) {
                            if (c != ' ') {
                                c = 97+i;
                            } else {
                                c = 65+i;
                            }
                        }
                     }
                     cout << c;
                }
                cout << endl;
            }
    } else {
            cout << "No solution found!" << endl;
    }
```

(continues on next page)

(continued from previous page)

```
    delete solver;
    return 0;
}
```

5.11 Learning to play the Sudoku

5.12 Renault car configuration system: learning user preferences

BIBLIOGRAPHY

[Schiex2020b] Céline Brouard and Simon de Givry and Thomas Schiex. Pushing Data in CP Models Using Graphical Model Learning and Solving. In *Proc. of CP-20*, Louvain-la-neuve, Belgium, 2020.

[Trosser2020a] Fulya Trösser, Simon de Givry and George Katsirelos. Relaxation-Aware Heuristics for Exact Optimization in Graphical Models. In *Proc.of CP-AI-OR'2020*, Vienna, Austria, 2020.

[Ruffini2019a] M. Ruffini, J. Vucinic, S. de Givry, G. Katsirelos, S. Barbe and T. Schiex. Guaranteed Diversity & Quality for the Weighted CSP. In *Proc. of ICTAI-19*, pages 18-25, Portland, OR, USA, 2019.

[Ouali2017] Abdelkader Ouali, David Allouche, Simon de Givry, Samir Loudni, Yahia Lebbah, Francisco Eckhardt, Lakhdar Loukil. Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization. In *Proc. of UAI-17*, pages 550-559, Sydney, Australia, 2017.

[Schiex2016a] David Allouche, Christian Bessière, Patrice Boizumault, Simon de Givry, Patricia Gutierrez, Jimmy H.M. Lee, Ka Lun Leung, Samir Loudni, Jean-Philippe Métivier, Thomas Schiex and Yi Wu. Tractability-preserving transformations of global cost functions. *Artificial Intelligence*, 238:166-189, 2016.

[Hurley2016b] B Hurley, B O'Sullivan, D Allouche, G Katsirelos, T Schiex, M Zytnicki and S de Givry. Multi-Language Evaluation of Exact Solvers in Graphical Model Discrete Optimization. *Constraints*, 21(3):413-434, 2016. Presentation at CPAIOR'16, Banff, Canada, http://www.inra.fr/mia/T/degivry/cpaior16sdg.pdf.

[Katsirelos2015a] 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.

[Schiex2014a] David Allouche, Jessica Davies, Simon de Givry, George Katsirelos, Thomas Schiex, Seydou Traoré, Isabelle André, Sophie Barbe, Steve Prestwich and Barry O'Sullivan. Computational Protein Design as an Optimization Problem. *Artificial Intelligence*, 212:59-79, 2014.

[Givry2013a] 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.

[Ficolofo2012] D Allouche, C Bessiere, P Boizumault, S de Givry, P Gutierrez, S Loudni, JP Métivier and T Schiex. Decomposing Global Cost Functions. In *Proc. of AAAI-12*, Toronto, Canada, 2012. http://www.inra.fr/mia/T/degivry/Ficolofo2012poster.pdf (poster).

[Favier2011a] A Favier, S de Givry, A Legarra and T Schiex. Pairwise decomposition for combinatorial optimization in graphical models. In *Proc. of IJCAI-11*, Barcelona, Spain, 2011. Video demonstration at http://www.inra.fr/mia/T/degivry/Favier11.mov.

[Cooper2010a] 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.

[Favier2009a] A. Favier, S. de Givry and P. Jégou. Exploiting Problem Structure for Solution Counting. I, *Proc. of CP-09*, pages 335-343, Lisbon, Portugal, 2009.

[Sanchez2009a] M Sanchez, D Allouche, S de Givry and T Schiex. Russian Doll Search with Tree Decomposition. In *Proc. of IJCAI'09*, Pasadena (CA), USA, 2009. http://www.inra.fr/mia/T/degivry/rdsbtd_ijcai09_sdg.ppt.

[Cooper2008] M. Cooper, S. de Givry, M. Sanchez, T. Schiex and M. Zytnicki. Virtual Arc Consistency for Weighted CSP. In *Proc. of AAAI-08*, Chicago, IL, 2008.

[Schiex2006a] S. de Givry, T. Schiex and G. Verfaillie. Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In *Proc. of AAAI-06*, Boston, MA, 2006. http://www.inra.fr/mia/T/degivry/VerfaillieAAAI06pres.pdf (slides).

[Heras2005] S. de Givry, M. Zytnicki, F. Heras and J. Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *Proc. of IJCAI-05*, pages 84-89, Edinburgh, Scotland, 2005.

[Larrosa2000] 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.

[koller2009] D Koller and N Friedman. Probabilistic graphical models: principles and techniques. The MIT Press, 2009.

[Ginsberg1995] W. D. Harvey and M. L. Ginsberg. Limited Discrepency Search. In *Proc. of IJCAI-95*, Montréal, Canada, 1995.

[Lecoutre2009] C. Lecoutre, L. Saïs, S. Tabary and V. Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173:1592,1614, 2009.

[boussemart2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

[idwalk:cp04] Bertrand Neveu, Gilles Trombettoni and Fred Glover. ID Walk: A Candidate List Strategy with a Simple Diversification Device. In *Proc. of CP*, pages 423-437, Toronto, Canada, 2004.

[Verfaillie1996] G. Verfaillie, M. Lemaître and T. Schiex. Russian Doll Search. In *Proc. of AAAI-96*, pages 181-187, Portland, OR, 1996.

[LL2009] J. H. M. Lee and K. L. Leung. Towards Efficient Consistency Enforcement for Global Constraints in Weighted Constraint Satisfaction. In *Proceedings of IJCAI'09*, pages 559-565, 2009.

[LL2010] J. H. M. Lee and K. L. Leung. A Stronger Consistency for Soft Global Constraints in Weighted Constraint Satisfaction. In *Proceedings of AAAI'10*, pages 121-127, 2010.

[LL2012asa] J. H. M. Lee and K. L. Leung. Consistency Techniques for Global Cost Functions in Weighted Constraint Satisfaction. *Journal of Artificial Intelligence Research*, 43:257-292, 2012.

[Larrosa2002] J. Larrosa. On Arc and Node Consistency in weighted {CSP}. In *Proc. AAAI'02*, pages 48-53, Edmondton, (CA), 2002.

[Larrosa2003] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for Weighted CSP. In *Proc. of the 18th IJCAI*, pages 239-244, Acapulco, Mexico, August 2003.

[Schiex2000b] T. Schiex. Arc consistency for soft constraints. In *Principles and Practice of Constraint Programming - CP 2000*, volume 1894 of *LNCS*, pages 411-424, Singapore, September 2000.

[CooperFCSP] M.C. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3):311-342, 2003.