toulbar2 Documentation

Release 1.0.0

INRAE

Dec 05, 2023

CONTENTS

CHAPTER

ONE

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 medals in competitions on Max-CSP/COP (CPAI08, XCSP3 2022 and 2023) and probabilistic graphical models (UAI 2008, 2010, 2014, 2022 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).

CHAPTER

TWO

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
- Université de Toulouse I Capitole (IRIT) and Université du Littoral Côte d'Opale, France (LISIC) for PILS local search solver

CHAPTER

THREE

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

CHAPTER

FOUR

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-).

CHAPTER

FIVE

LICENSE

```
MIT License

Copyright (c) 2022 toulbar2 team

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER

SIX

DOWNLOADS

6.1 Open-source code

- toulbar2 on GitHub

6.2 Packages

- to install **toulbar2** using the package manager in Debian and Debian derived Linux distributions (Ubuntu, Mint,...):

```
apt install toulbar2
```

6.3 Binaries

- **Latest release toulbar2 binaries**

  Linux 64bit | MacOs 64bit | Windows 64bit

6.4 Python package

- **pytoulbar2** module for Linux and MacOS on PyPI

6.5 Docker images

- In Toulbar2 Packages :
  - **toulbar2** : Docker image containing toulbar2 and its pytoulbar2 Python API *(installed from sources with cmake options -DPYTB2=ON and -DXML=ON)*. Install from the command line:

    ```
    docker pull ghcr.io/toulbar2/toulbar2/toulbar2:master
    ```

  - **pytoulbar2** : Docker image containing pytoulbar2 the Python API of toulbar2 *(installed with python3 -m pip install pytoulbar2)*. Install from the command line:

```
docker pull ghcr.io/toulbar2/toulbar2/pytoulbar2:master
```

CHAPTER

SEVEN

BENCHMARK LIBRARIES

- EvalGM : 3026 discrete optimization benchmarks available in various formats (wcsp, wcnf, uai, lp, mzn). [Hurley et al CPAIOR 2016]
- Cost Function Library : an on-going collection of benchmarks from various domains of Artificial Intelligence, Constraint Programming, and Operations Research (formats in wcsp, wcnf, lp, qpbo, uai, opd, and more).

CHAPTER

EIGHT

TUTORIALS

- tutorial materials on cost function networks at ACP/ANITI/GDR-IA/RO Autumn School 2020.
- 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.

Here are several examples that can be followed as tutorials. They use toulbar2 in order to resolve different problems. According to cases, they may contain source code, tutorials explaining the examples, possibility to run yourself…

You will find the mentioned examples, among the following exhaustive list of examples.

- *Weighted n-queen problem*
- *Weighted latin square problem*
- *Bicriteria weighted latin square problem*
- *Radio link frequency assignment problem*
- *Frequency assignment problem with polarization*
- *Mendelian error detection problem*
- *Block modeling problem*
- *Airplane landing problem*
- *Warehouse location problem*
- tuto_rcpsp
- *Golomb ruler problem*
- *Square packing problem*
- *Square soft packing problem*
- *Board coloration problem*
- *Learning to play the Sudoku*
- *Learning car configuration preferences*
- *Visual Sudoku Tutorial*
- *Verbose version of a sudoku code*

CHAPTER

NINE

USE CASES

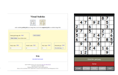
Here are several toulbar2 use cases, where toulbar2 has been used in order to resolve different problems. According to cases, they can be used to overview, learn, use toulbar2… They may contain source code, explanations, possibility to run yourself…

You will find the mentioned examples, among the following exhaustive list of examples.

toulbar2 and Deep Learning :

- *Visual Sudoku Tutorial*
- *Visual Sudoku Application* 

Some applications based on toulbar2 :

- **Mendelsoft** : Mendelsoft detects Mendelian errors in complex pedigree [Sanchez et al, Constraints 2008].
- **Pompd** : POsitive Multistate Protein Design, [Vucini et al Bioinformatics 2020]
- *Visual Sudoku Application* 

Misc :

- *A sudoku code*

CHAPTER

TEN

LIST OF ALL EXAMPLES

10.1 Weighted n-queen problem

10.1.1 Brief description

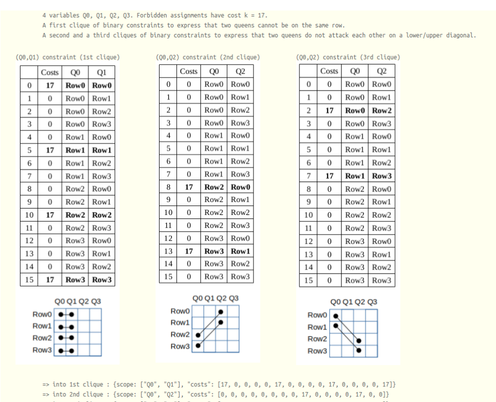
The problem consists in assigning N queens on a NxN chessboard with random costs in (1..N) associated to every cell such that each queen does not attack another queen and the sum of the costs of queen's selected cells is minimized.

10.1.2 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. We add N unary cost functions to create the objective function with random costs on every cell.

10.1.3 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]}

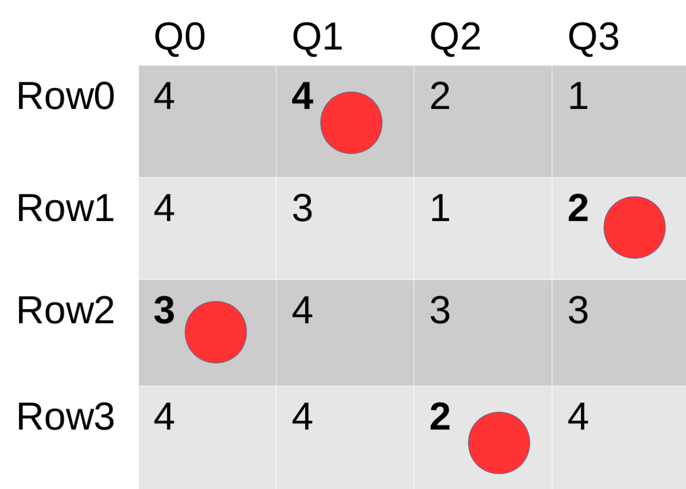
```
{
  problem: { "name": "4-queen", "mustbe": "<17" },
  variables: {"Q0":["Row0", "Row1", "Row2", "Row3"], "Q1":["Row0", "Row1", "Row2", "Row3
↪"],
               "Q2":["Row0", "Row1", "Row2", "Row3"], "Q3":["Row0", "Row1", "Row2", "Row3
```

(continues on next page)

(continued from previous page)

```
↪"]},
  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 :



10.1.4 Python model

The following code using the pytoulbar2 library solves the weighted N-queen problem with the first argument being the number of queens N (e.g. “python3 weightedqueens.py 8”).

weightedqueens.py

```
import sys
from random import seed, randint
seed(123456789)
import pytoulbar2

N = int(sys.argv[1])

top = N**2 +1

Problem = pytoulbar2.CFN(top)

for i in range(N):
    Problem.AddVariable('Q' + str(i+1), ['row' + str(a+1) for a in range(N)])

for i in range(N):
        for j in range(i+1,N):

                #Two queens cannot be on the same row constraints
                ListConstraintsRow = []
                for a in range(N):
                        for b in range(N):
                                if a != b :
                                        ListConstraintsRow.append(0)
                                else:
                                         ListConstraintsRow.append(top)
                Problem.AddFunction([i, j], ListConstraintsRow)

                #Two queens cannot be on the same upper diagonal constraints
                ListConstraintsUpperD = []
                for a in range(N):
                        for b in range(N):
                                if a + i != b + j :
                                        ListConstraintsUpperD.append(0)
                                else:
                                         ListConstraintsUpperD.append(top)
                Problem.AddFunction([i, j], ListConstraintsUpperD)

                #Two queens cannot be on the same lower diagonal constraints
                ListConstraintsLowerD = []
                for a in range(N):
                        for b in range(N):
                                if a - i != b - j :
                                        ListConstraintsLowerD.append(0)
                                else:
                                         ListConstraintsLowerD.append(top)
                Problem.AddFunction([i, j], ListConstraintsLowerD)

```

(continues on next page)

(continued from previous page)

```
#Random unary costs
for i in range(N):
        ListConstraintsUnaryC = []
        for j in range(N):
                ListConstraintsUnaryC.append(randint(1,N))
        Problem.AddFunction([i], ListConstraintsUnaryC)


#Problem.Dump('WeightQueen.cfn')
Problem.CFN.timer(300)
res = Problem.Solve(showSolutions = 3)
if res:
        for i in range(N):
                row = ['X' if res[0][j]==i else ' ' for j in range(N)]
                print(row)
        # and its cost
        print("Cost:", int(res[1]))
```

10.2 Weighted latin square problem

10.2.1 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 cost in (1…N) is associated to every domain value. The objective is to find a complete assignment where the sum of the costs associated to the selected values for the cells is minimized.

10.2.2 CFN model

We create NxN variables, one for every cell, with domain size N. An AllDifferent hard global constraint is used to model a permutation for every row and every column. Its encoding uses knapsack constraints. Unary cost functions containing random costs associated to domain values are generated for every cell. The worst possible solution is when every cell is associated with a cost of N, so the maximum cost of a solution is N**3, so forbidden assignments have cost k=N**3+1.

10.2.3 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
```

(continues on next page)

(continued from previous page)

```
↪", "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]},
   {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, costs associated to the selected values) :

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

10.2.4 Python model

The following code using the pytoulbar2 library solves the weighted latin square problem with the first argument being the dimension N of the chessboard (e.g. "python3 latinsquare.py 6").

latinsquare.py

```
import sys
from random import seed, randint
seed(123456789)
import pytoulbar2

N = int(sys.argv[1])

top = N**3 +1

Problem = pytoulbar2.CFN(top)

for i in range(N):
    for j in range(N):
        #Create a variable for each square
        Problem.AddVariable('Cell(' + str(i) + ',' + str(j) + ')', range(N))

for i in range(N):
    #Create a constraint all different with variables on the same row
    Problem.AddAllDifferent(['Cell(' + str(i) + ',' + str(j) + ')' for j in range(N)],␣
↪encoding = 'salldiffkp')

    #Create a constraint all different with variables on the same column
    Problem.AddAllDifferent(['Cell(' + str(j) + ',' + str(i) + ')'for j in range(N)],␣
↪encoding = 'salldiffkp')

#Random unary costs
for i in range(N):
    for j in range(N):
        ListConstraintsUnaryC = []
        for l in range(N):
            ListConstraintsUnaryC.append(randint(1,N))
        Problem.AddFunction(['Cell(' + str(i) + ',' + str(j) + ')'],␣
↪ListConstraintsUnaryC)

#Problem.Dump('WeightLatinSquare.cfn')
Problem.CFN.timer(300)
res = Problem.Solve(showSolutions = 3)
if res and len(res[0]) == N*N:
    # pretty print solution
    for i in range(N):
        print([res[0][i * N + j] for j in range(N)])
    # and its cost
    print("Cost:", int(res[1]))
```

10.3 Bicriteria weighted latin square problem

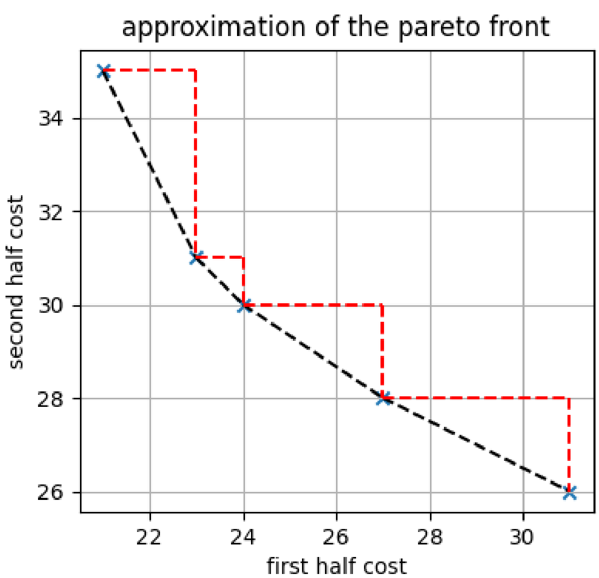
10.3.1 Brief description

In this variant of the *Weighted latin square problem*, the objective (sum of the costs of the cells) is decomposed into two criteria: the sum of the cells in the first half of the chessboard and the sum of the cells in the second half. A subset of the pareto solutions can be obtained by solving linear combinations of the two criteria with various weights on the objectives. This can be achieved in ToulBar2 via a MultiCFN object.

10.3.2 CFN model

Similarly to the *Weighted latin square problem*, NxN variables are created with a domain size N. In this model, the permutation of every row and every column is ensured through infinite costs in binary cost functions. Two different CFN are created to represent the two objectives: a first CFN where unary costs are added only for the first half of the chessboard, and a second one with unary costs for the remaining cells.

Toulbar2 allows to either solve for a chosen weighted sum of the two cost function networks as input, or approximate the pareto front by enumerating a complete set of non-redundant weights. As it is shown below, the method allows to compute solutions which costs lie in the convex hull of the pareto front. However, potential solutions belonging to the triangles will be missed with this approach.



10.3.3 Python model

The following code using the pytoulbar2 library solves the bicriteria weighted latin square problem with two different pairs of weights for the two objectives.

bicriteria_latinsquare.py

```
import sys
from random import seed, randint
seed(123456789)
import pytoulbar2
```

(continues on next page)

(continued from previous page)

```
from matplotlib import pyplot as plt

N = int(sys.argv[1])

top = N**3 +1

# printing a solution as a grid
def print_solution(sol, N):

  grid = [0 for _ in range(N*N)]
  for k,v in sol.items():
    grid[ int(k[5])*N+int(k[7]) ] = int(v[1:])

  output = ''
  for var_ind in range(len(sol)):
    output += str(grid[var_ind]) + ' '
    if var_ind % N == N-1:
      output += '\n'
  print(output, end='')


# creation of the base problem: variables and hard constraints (alldiff must be␣
↪decomposed into binary constraints)
def create_base_cfn(cfn, N, top):

  # variable creation
  var_indexes = []

  # create N^2 variables, with N values in their domains
  for row in range(N):
    for col in range(N):
      index = cfn.AddVariable('Cell_' + str(row) + '_' + str(col), ['v' + str(val) for␣
↪val in range(N)])
      var_indexes.append(index)

  # all permutation constraints: pairwise all different

  # forbidden values are enforced by infinite costs
  alldiff_costs = [ top if row == col else 0 for row in range(N) for col in range(N) ]

  for index in range(N):
    for var_ind1 in range(N):
      for var_ind2 in range(var_ind1+1, N):

        # permutations in the rows
        cfn.AddFunction([var_indexes[N*index+var_ind1], var_indexes[N*index+var_ind2]],␣
↪alldiff_costs)

        # permutations in the columns
        cfn.AddFunction([var_indexes[index+var_ind1*N], var_indexes[index+var_ind2*N]],␣
↪alldiff_costs)
```

(continues on next page)

(continued from previous page)

```
split_index = (N*N)//2

# generation of random costs
cell_costs = [[randint(1,N) for _ in range(N)] for _ in range(N*N)]

# multicfn is the main object for combining multiple cost function networks
multicfn = pytoulbar2.MultiCFN()


# first cfn: first half of the grid
cfn = pytoulbar2.CFN(ubinit = top, resolution=6)
cfn.SetName('first half')
create_base_cfn(cfn, N, top)
for variable_index in range(split_index):
  cfn.AddFunction([variable_index], cell_costs[variable_index])
multicfn.PushCFN(cfn)


# second cfn: second half of the grid
cfn = pytoulbar2.CFN(ubinit = top, resolution=6)
cfn.SetName('second half')
create_base_cfn(cfn, N, top)
for variable_index in range(split_index+1, N*N):
  cfn.AddFunction([variable_index], cell_costs[variable_index])
multicfn.PushCFN(cfn)


# solve with a first pair of weights
weights = (1., 2.)

multicfn.SetWeight(0, weights[0])
multicfn.SetWeight(1, weights[1])

cfn = pytoulbar2.CFN()
cfn.InitFromMultiCFN(multicfn) # the final cfn is initialized from the combined cfn

# cfn.Dump('python_latin_square_bicriteria.cfn')

result = cfn.Solve()

if result:
  print('Solution found with weights', weights, ':')
  sol_costs = multicfn.GetSolutionCosts()
  solution = multicfn.GetSolution()
  print_solution(solution, N)
  print('with costs:', sol_costs, '(sum=', result[1], ')')

print('\n')

# solve a second time with other weights
weights = (2.5, 1.)
```

(continues on next page)

(continued from previous page)

```
multicfn.SetWeight(0, weights[0])
multicfn.SetWeight(1, weights[1])

cfn = pytoulbar2.CFN()
cfn.InitFromMultiCFN(multicfn) # the final cfn is initialized from the combined cfn

# cfn.Dump('python_latin_square_bicriteria.cfn')

result = cfn.Solve()

if result:
  print('Solution found with weights', weights, ':')
  sol_costs = multicfn.GetSolutionCosts()
  solution = multicfn.GetSolution()
  print_solution(solution, N)
  print('with costs:', sol_costs, '(sum=', result[1], ')')


# approximate the pareto front
(solutions, costs) = multicfn.ApproximateParetoFront(0, 'min', 1, 'min')

fig, ax = plt.subplots()
ax.scatter([c[0] for c in costs], [c[1] for c in costs], marker='x')
for index in range(len(costs)-1):
  ax.plot([costs[index][0], costs[index+1][0]], [costs[index][1],costs[index+1][1]], '--
↪', c='k')
  ax.plot([costs[index][0], costs[index+1][0]], [costs[index][1],costs[index][1]], '--',␣
↪c='red')
  ax.plot([costs[index+1][0], costs[index+1][0]], [costs[index][1],costs[index+1][1]], '-
↪-', c='red')

ax.set_xlabel('first half cost')
ax.set_ylabel('second half cost')
ax.set_title('approximation of the pareto front')
ax.set_aspect('equal')

plt.grid()
plt.show()
```

10.4 Radio link frequency assignment problem

10.4.1 Brief description

The problem consists in assigning frequencies to radio communication links in such a way that no interferences occur. Domains are set of integers (non-necessarily consecutive).

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)
- (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.

So the goal is to minimize the sum:
a_1*nc1+…+a_4*nc4+b_1*nv1+…+b_4*nv4

where nci is the number of violated constraints with cost a_i and nvi is the number of modified variables with mobility cost b_i.

Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.P. Constraints (1999) 4: 79.

10.4.2 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 with cost equal to a_i. Unary cost functions are used to model mobility costs with cost equal to b_i. The initial upper bound is defined as 1 plus the total cost where all the soft constraints are maximally violated (costs a_4/b_4).

10.4.3 Data

Original data files can be downloaded 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.

10.4.4 Python model

The following code solves the corresponding cost function network using the pytoulbar2 library and needs 4 arguments: the variable file, the domain file, the constraints file and the cost file (e.g. “python3 rlfap.py var.txt dom.txt ctr.txt cst.txt”).

rlfap.py

```
import sys
import pytoulbar2

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 = 1
		self.Domain = {}

		stream = open(var)
		for line in stream:
			if len(line.split())>=4:
				(varnum, vardom, value, mobility) = line.split()[:4]
				self.Domain[int(varnum)] = int(vardom)
				self.var.append((int(varnum), int(vardom), int(value),
```

(continues on next page)

(continued from previous page)

```
↪int(mobility)))
                                        self.nbb["b" + str(mobility)] = self.nbb.get("b" +␣
↪str(mobility), 0) + 1
                                else:
                                        (varnum, vardom) = line.split()[:2]
                                        self.Domain[int(varnum)] = int(vardom)
                                        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))


#collect data
data = Data(sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4])

top = data.top
Problem = pytoulbar2.CFN(top)

#create a variable for each link
for e in data.var:
        domain = []
        for f in data.dom[e[1]]:
                domain.append('f' + str(f))
        Problem.AddVariable('link' + str(e[0]), domain)

#binary hard and soft constraints
for (var1, var2, operand, deviation, weight) in data.ctr:
        ListConstraints = []
        for a in data.dom[data.Domain[var1]]:
                for b in data.dom[data.Domain[var2]]:
                        if ((operand==">" and abs(a - b) > deviation) or (operand=="="␣
↪and abs(a - b) == deviation)):
                                ListConstraints.append(0)
```

(continues on next page)

(continued from previous page)

```
                        else:
                                ListConstraints.append(data.cost.get('a' + str(weight),
↪top))
        Problem.AddFunction(['link' + str(var1), 'link' + str(var2)], ListConstraints)

#unary hard and soft constraints
for e in data.var:
        if len(e) >= 3:
                ListConstraints = []
                for a in data.dom[e[1]]:
                        if a == e[2]:
                                ListConstraints.append(0)
                        else:
                                ListConstraints.append(data.cost.get('b' + str(e[3]),
↪top))
                Problem.AddFunction(['link' + str(e[0])], ListConstraints)

#Problem.Dump('Rflap.cfn')
Problem.CFN.timer(300)
res = Problem.Solve(showSolutions=3)
if res:
        print("Best solution found with cost:",int(res[1]),"in", Problem.GetNbNodes(),
↪"search nodes.")
else:
        print('Sorry, no solution found!')
```

10.5 Frequency assignment problem with polarization

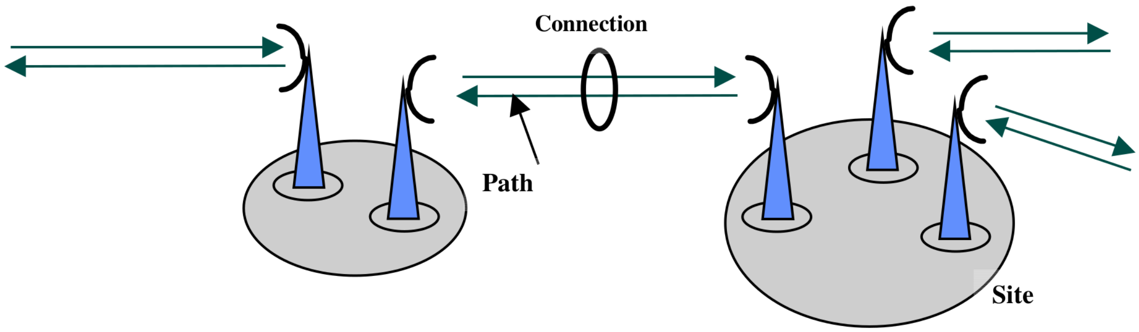
10.5.1 Brief description

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 (weighted) number of violations of (IV) at lower levels. See `ROADEF_Challenge_2001` .

The cost of a given solution will be calculated by the following formula: 10*k*nbsoft**2 + 10*nbsoft*V(k-1) + V(k-2) + V(k-3) + … + V0 where nbsoft is the number of soft constraints in the problem and k the feasible relaxation level and V(i) the number of violated contraints of level i.



10.5.2 CFN model

We create a single variable to represent a pair (frequency,polarization) for every radio link, but be aware, toulbar2 can only read str or int values, so in order to give a tuple to toulbar2 we need to first transform them into string. We use hard binary constraints to modelize (I) to (III) type constraints.

We assume the relaxation level k is given as input. In order to modelize (IV) type constraints we first take in argument the level of relaxation i, and we create 11 constraints, one for each relaxation level from 0 to 10. The first k-2 constraints are soft and with a violation cost of 1. The soft constraint at level k-1 has a violation cost 10*nbsoft (the number of soft constraints) in order to maximize first the number of satisfied constraints at level k-1 and then the other soft constraints. The constraints at levels k to 10 are hard constraints.

The initial upper bound of the problem will be 10*(k+1)*nbsoft**2 +1.

10.5.3 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.

10.5.4 Python model

The following code solves the corresponding cost function network using the pytoulbar2 library and needs 4 arguments: the data file and the relaxation level (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 | awk -f ./sol2fapp.awk - exemple1”).

fapp.py

```
import sys
import pytoulbar2

class Data:
        def __init__(self, filename, k):
                self.var = {}
                self.dom = {}
                self.ctr = list()
```

(continues on next page)

(continued from previous page)

```
                self.softeq = list()
                self.softne = list()
                self.nbsoft = 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) == 0:
                                        self.var[int(route)] = [(f,-1) for f in self.
↪dom[int(dom)]] + [(f,1) for f in self.dom[int(dom)]]
                                if int(polarisation) == -1:
                                        self.var[int(route)] = [(f,-1) for f in self.
↪dom[int(dom)]]
                                if int(polarisation) == 1:
                                        self.var[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]]))
                                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.top = 10*(k+1)*self.nbsoft**2 + 1

if len(sys.argv) < 2:
        exit('Command line argument is composed of the problem data filename and the␣
↪relaxation level')
k = int(sys.argv[2])

#collect data
```

(continues on next page)

(continued from previous page)

```
data = Data(sys.argv[1], k)

Problem = pytoulbar2.CFN(data.top)

#create a variable for each link
for e in list(data.var.keys()):
        domain = []
        for i in data.var[e]:
                domain.append(str(i))
        Problem.AddVariable("X" + str(e), domain)

#hard binary constraints
for (route1, route2, vartype, operand, deviation) in data.ctr:
        Constraint = []
        for (f1,p1) in data.var[route1]:
                 for (f2,p2) in data.var[route2]:
                        if vartype == 'F':
                                if operand == 'E':
                                        if abs(f2 - f1) == deviation:
                                                Constraint.append(0)
                                        else:
                                                Constraint.append(data.top)
                                else:
                                        if abs(f2 - f1) != deviation:
                                                Constraint.append(0)
                                        else:
                                                Constraint.append(data.top)
                        else:
                                if operand == 'E':
                                        if p2 == p1:
                                                Constraint.append(0)
                                        else:
                                                Constraint.append(data.top)
                                else:
                                        if p2 != p1:
                                                Constraint.append(0)
                                        else:
                                                Constraint.append(data.top)
        Problem.AddFunction(["X" + str(route1), "X" + str(route2)], Constraint)


#soft binary constraints for equal polarization
for (route1, route2, deviations) in data.softeq:
        for i in range(11):
                ListConstraints = []
                for (f1,p1) in data.var[route1]:
                        for (f2,p2) in data.var[route2]:
                                if p1!=p2 or abs(f1 - f2) >= deviations[i]:
                                        ListConstraints.append(0)
                                elif i >= k:
                                        ListConstraints.append(data.top)
                                elif i == k-1:
```

(continues on next page)

(continued from previous page)

```
                                        ListConstraints.append(10*data.nbsoft)
                                else:
                                        ListConstraints.append(1)
                Problem.AddFunction(["X" + str(route1), "X" + str(route2)],␣
↪ListConstraints)


#soft binary constraints for not equal polarization
for (route1, route2, deviations) in data.softne:
        for i in range(11):
                ListConstraints = []
                for (f1,p1) in data.var[route1]:
                        for (f2,p2) in data.var[route2]:
                                if p1==p2 or abs(f1 - f2) >= deviations[i]:
                                        ListConstraints.append(0)
                                elif i >= k:
                                        ListConstraints.append(data.top)
                                elif i == k-1:
                                        ListConstraints.append(10*data.nbsoft)
                                else:
                                        ListConstraints.append(1)
                Problem.AddFunction(["X" + str(route1), "X" + str(route2)],␣
↪ListConstraints)

#zero-arity cost function representing a constant cost corresponding to the relaxation␣
↪at level k
Problem.AddFunction([], 10*k*data.nbsoft**2)

#Problem.Dump('Fapp.cfn')
Problem.CFN.timer(900)
Problem.Solve(showSolutions=3)
```

10.6 Mendelian error detection problem

10.6.1 Brief description

The problem is to detect marker genotyping incompatibilities (Mendelian errors) in complex pedigrees. The input is a pedigree data with partial observed genotyping data at a single locus, we assume the pedigree to be exact, but not the genotyping data. The problem is to assign genotypes (unordered pairs of alleles) to all individuals such that they are compatible with the Mendelian law of heredity (one allele is the same as their father's and one as their mother's). The goal is to maximize the number of matching alleles between the genotyping data and the solution. Each difference from the genotyping data has a cost of 1.

Sanchez, M., de Givry, S. and Schiex, T. Constraints (2008) 13:130.

10.6.2 CFN model

We create N variables, one for each individual genotype with domain being all possible unordered pairs of existing alleles. Hard ternary cost functions express mendelian law of heredity (one allele is the same as their father's and one as their mother's, with mother and father defined in the pedigree data). For each genotyping data, we create one unary soft constraint with violation cost equal to 1 to represent the matching between the genotyping data and the solution.

10.6.3 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.

10.6.4 Python model

The following code solves the corresponding cost function network using the pytoulbar2 library (e.g. "python3 mendel.py simple.pre").

mendel.py

```
import sys
import pytoulbar2

class Data:
        def __init__(self, ped):
                self.id = list()
                self.father = {}
                self.mother = {}
                self.allelesId = {}
                self.ListAlle = list()
                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.allelesId[int(id)] = (int(allele1), int(allele2)) if␣
↪int(allele1) < int(allele2) else (int(allele2), int(allele1))
                        if not(int(allele1) in self.ListAlle) and int(allele1) != 0:
                                self.ListAlle.append(int(allele1))
                        if int(allele2) != 0 and not(int(allele2) in self.ListAlle):
                                self.ListAlle.append(int(allele2))
                        if int(allele1) != 0 or int(allele2) != 0:
                                self.obs += 1

#collect data
data = Data(sys.argv[1])
top = int(data.obs+1)

Problem = pytoulbar2.CFN(top)
```

(continues on next page)

(continued from previous page)

```
#create a variable for each individual
for i in data.id:
        domains = []
        for a1 in data.ListAlle:
                for a2 in data.ListAlle:
                        if a1 <= a2:
                                domains.append('a'+str(a1)+'a'+str(a2))
        Problem.AddVariable('g' + str(i) , domains)

#create the constraints that represent the mendel's laws
ListConstraintsMendelLaw = []
for p1 in data.ListAlle:
        for p2 in data.ListAlle:
                if p1 <= p2:        # father alleles
                        for m1 in data.ListAlle:
                                for m2 in data.ListAlle:
                                        if m1 <= m2:        # mother alleles
                                                for a1 in data.ListAlle:
                                                        for a2 in data.ListAlle:
                                                                if a1 <= a2:        #␣
↪child alleles
                                                                        if (a1 in (p1,
↪p2) and a2 in (m1,m2)) or (a2 in (p1,p2) and a1 in (m1,m2)) :
                                                                                ListConstraintsMendelLaw
↪append(0)
                                                                        else :
                                                                        ␣
↪        ListConstraintsMendelLaw.append(top)

for i in data.id:
        #ternary constraints representing mendel's laws
        if data.father.get(i, 0) != 0 and data.mother.get(i, 0) != 0:
                Problem.AddFunction(['g' + str(data.father[i]),'g' + str( data.
↪mother[i]), 'g' + str(i)], ListConstraintsMendelLaw)

        #unary constraints linked to the observations
        if data.allelesId[i][0] != 0 and data.allelesId[i][1] != 0:
                ListConstraintsObservation = []
                for a1 in data.ListAlle:
                        for a2 in data.ListAlle:
                                if a1 <= a2:
                                        if (a1,a2) == data.allelesId[i]:
                                                ListConstraintsObservation.append(0)
                                        else :
                                                 ListConstraintsObservation.append(1)
                Problem.AddFunction(['g' + str(i)], ListConstraintsObservation)

#Problem.Dump('Mendel.cfn')
Problem.CFN.timer(300)
res = Problem.Solve(showSolutions=3)
if res:
```

(continues on next page)

(continued from previous page)

```
        print('There are',int(res[1]),'difference(s) between the solution and the observation.')
else:
        print('No solution found')
```

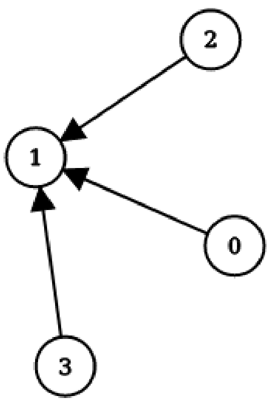
10.7 Block modeling problem

10.7.1 Brief description

This is a clustering problem, occurring in social network analysis.

The problem is to divide a given directed 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 in G between the nodes from the two clusters.

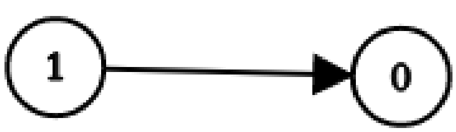
For example, the following graph G is composed of 4 nodes:



and corresponds to the following matrix:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

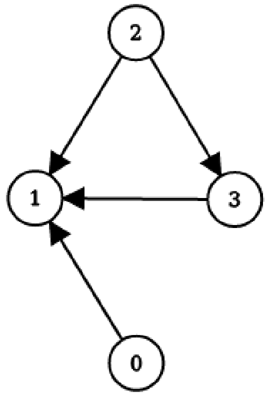
It can be perfectly clusterized into the following graph by clustering together the nodes 0, 2 and 3 in cluster 1 and the node 1 in cluster 0:



and this graph corresponds to the following M matrix:

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

On the contrary, if we decide to cluster the next graph G’ in the same way as above, the edge (2, 3) will be ‘lost’ in the process and the cost of the solution will be 1.



The goal is to find a k-clustering of a given graph and the associated matrix M that minimizes 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.

10.7.2 CFN model

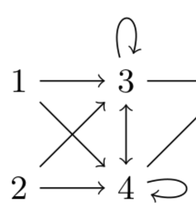
We create N variables, one for every node of the graph, with domain size k representing the clustering. We add k*k Boolean variables for representing M.

For all triplets of two nodes u, v, and one matrix cell M[i,j], we have a ternary cost function that 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) is in G. In order to break symmetries, we constrain the first k-1 node variables to be assigned to a cluster number less than or equal to their index

10.7.3 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)

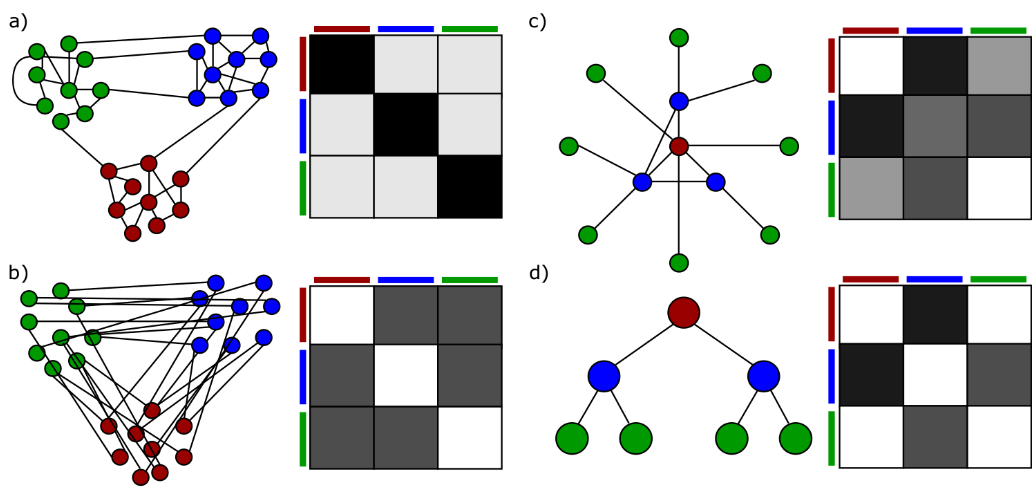


$$G = \left(\begin{array}{cc|cc|c} \mathbf{0} & \mathbf{0} & 1 & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 1 & 1 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & 1 & 1 & 1 \\ \mathbf{0} & \mathbf{0} & 1 & 1 & 1 \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right)$$

$$\{1,2\} \to \{3,4\} \longrightarrow \{5\}$$

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

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`.

10.7.4 Python model

The following code using pytoulbar2 library solves the corresponding cost function network (e.g. "python3 blockmodel.py simple.mat 3").

`blockmodel.py`

```
import sys
import pytoulbar2

#read adjency matrix of graph G
Lines = open(sys.argv[1], 'r').readlines()
GMatrix = [[int(e) for e in l.split(' ')] for l in Lines]

N = len(Lines)
Top = N*N + 1

K = int(sys.argv[2])

#give names to node variables
Var = [(chr(65 + i) if N < 28 else "x" + str(i)) for i in range(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
#    Var = ["Sherrif","CivilDef","Coroner","Attorney","HighwayP","ParksRes","GameFish","KansasDOT","ArmyCorps","ArmyReserve","CrableAmb","FrankCoAmb","LeeRescue","Shawney","BurlPolice","LyndPolice","RedCross","TopekaFD","CarbFD","TopekaRBW"] # Kansas

Problem = pytoulbar2.CFN(Top)

#create a Boolean variable for each coefficient of the M GMatrix
for u in range(K):
```

(continues on next page)

(continued from previous page)

```
    for v in range(K):
        Problem.AddVariable("M_" + str(u) + "_" + str(v), range(2))

#create a domain variable for each node in graph G
for i in range(N):
    Problem.AddVariable(Var[i], range(K))

#general case for each edge in G
for u in range(K):
    for v in range(K):
        for i in range(N):
            for j in range(N):
                if i != j:
                    ListCost = []
                    for m in range(2):
                        for k in range(K):
                            for l in range(K):
                                if (u == k and v == l and GMatrix[i][j] != m):
                                    ListCost.append(1)
                                else:
                                    ListCost.append(0)
                    Problem.AddFunction(["M_" + str(u) + "_" + str(v), Var[i], Var[j]],
↪ListCost)



# self-loops must be treated separately as they involves only two variables
for u in range(K):
    for i in range(N):
        ListCost = []
        for m in range(2):
            for k in range(K):
                if (u == k and GMatrix[i][i] != m):
                    ListCost.append(1)
                else:
                    ListCost.append(0)
        Problem.AddFunction(["M_" + str(u) + "_" + str(u), Var[i]], ListCost)

# breaking partial symmetries by fixing first (K-1) domain variables to be assigned to a␣
↪cluster number less than or equal to their index
for l in range(K-1):
    Constraint = []
    for k in range(K):
        if k > l:
            Constraint.append(Top)
        else:
            Constraint.append(0)
    Problem.AddFunction([Var[l]], Constraint)

Problem.Dump(sys.argv[1].replace('.mat','.cfn'))
Problem.CFN.timer(300)
res = Problem.Solve(showSolutions = 3)
```

(continues on next page)

(continued from previous page)

```
if res:
    print("M matrix:")
    for u in range(K):
        Line = []
        for v in range(K):
            Line.append(res[0][u*K+v])
        print(Line)
    for k in range(K):
        for i in range(N):
            if res[0][K**2+i] == k:
                print("Node",Var[i],"with index",str(i),"is in cluster",
↪str(res[0][K**2+i]))
```

10.8 Airplane landing problem

10.8.1 Brief description

We consider a single plane's landing runway. Given a set of planes with given target landing time, the objective is to minimize the total weighted deviation from the target landing time for each plane.

There are costs associated with landing either earlier or later than the target landing time for each plane.

Each plane has to land within its predetermined time window. For each pair of planes, there is an additional constraint to enforce that the separation time between those planes is larger than a given number.

J.E. Beasley, M. Krishnamoorthy, Y.M. Sharaiha and D. Abramson. Scheduling aircraft landings - the static case. Transportation Science, vol.34, 2000.

10.8.2 CFN model

We create N variables, one for each plane, with domain value equal to all their possible landing time.

Binary hard cost functions express separation times between pairs of planes. Unary soft cost functions represent the weighted deviation for each plane.

10.8.3 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.

10.8.4 Python model solver

The following code uses the pytoulbar2 module to generate the cost function network and solve it (e.g. "python3 airland.py airland1.txt").

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
    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):
    ListCost = []
    for a in range(LT[i][0], LT[i][2]+1):
```

(continues on next page)

(continued from previous page)

```
        if a < LT[i][1]:
            ListCost.append(PC[i][0]*(LT[i][1] - a))
        else:
            ListCost.append(PC[i][1]*(a - LT[i][1]))
    Problem.AddFunction([i], ListCost)

for i in range(N):
    for j in range(i+1,N):
        Constraint = []
        for a in range(LT[i][0], LT[i][2]+1):
            for b in range(LT[j][0], LT[j][2]+1):
                if a+ST[i][j]>b and b+ST[j][i]>a:
                    Constraint.append(top)
                else:
                    Constraint.append(0)
        Problem.AddFunction([i, j],Constraint)

#Problem.Dump('airplane.cfn')
Problem.NoPreprocessing()
Problem.Solve(showSolutions = 3)
```

10.9 Warehouse location problem

10.9.1 Brief description

A company considers opening warehouses at some candidate locations with each of them having a maintenance cost if it is open.

The company controls a set of given stores and each of them needs to take supplies to one of the warehouses, but depending on the warehouse chosen, there will be an additional supply cost.

The objective is to choose which warehouse to open and to divide the stores among the open warehouses in order to minimize the total cost of supply and maintenance costs.

10.9.2 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 to represent to which warehouse the store will take supplies.

Hard binary constraints represent that a store cannot take supplies from a closed warehouse. Soft unary constraints represent the maintenance cost of the warehouses. Soft unary constraints represent the store's cost regarding which warehouse to take supplies from.

10.9.3 Data

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

10.9.4 Python model solver

The following code uses the pytoulbar2 module to generate the cost function network and solve it (e.g. "python3 warehouse.py cap44.txt 1" found an optimum value equal to 10349757). Other instances are available here in cfn format.

warehouse.py

```
import sys
import pytoulbar2

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

precision = int(sys.argv[2])  # in [0,9], used to convert cost values from float to␣
↪integer (by 10**precision)

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


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):
```

(continues on next page)

(continued from previous page)

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

# 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):
        Constraint = []
        for a in range(2):
            for b in range(N):
                if a == 0 and b == i:
                    Constraint.append(top)
                else:
                    Constraint.append(0)
        Problem.AddFunction([i, N+j], Constraint)

#Problem.Dump('warehouse.cfn')
Problem.Solve(showSolutions=3)
```

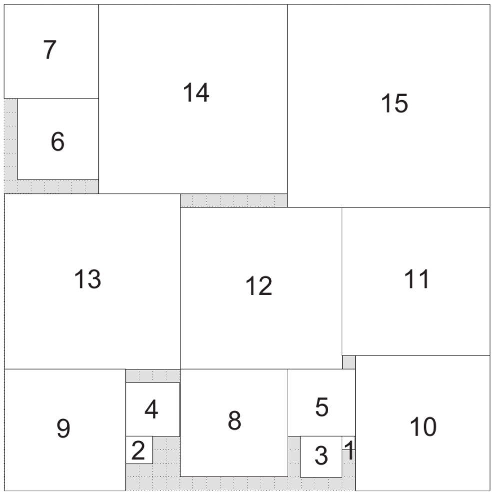
10.10 Square packing problem

10.10.1 Brief description

We have N squares of respective size 1×1, 2×2,. . . , NxN. We have to fit them without overlaps into a square of size SxS.

Results up to N=56 are given here.

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



10.10.2 CFN model

We create an integer variable of domain size (S-i)x(S-i) for each square. The variable represents the position of the top left corner of the square.

The value of a given variable modulo (S-i) gives the x-coordinate, whereas its value divided by (S-i) gives the y-coordinate.

We have hard binary constraints to forbid any overlapping pair of squares.

We make the problem a pure satisfaction problem by fixing the initial upper bound to 1.

10.10.3 Python model

The following code uses the pytoulbar2 library to generate the cost function network and solve it (e.g. "python3 square.py 3 5"). `square.py`

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

import pytoulbar2
try:
        N = int(sys.argv[1])
        S = int(sys.argv[2])
        assert N <= S
except:
        print('Two integers need to be given as arguments: N and S')
        exit()

#pure constraint satisfaction problem
Problem = pytoulbar2.CFN(1)

#create a variable for each square
for i in range(N):
        Problem.AddVariable('sq' + str(i+1), ['(' + str(l) + ',' + str(j) + ')' for l in␣
```

(continues on next page)

(continued from previous page)

```
↪range(S-i) for j in range(S-i)])

#binary hard constraints for overlapping squares
for i in range(N):
        for j in range(i+1,N):
                ListConstraintsOverlaps = []
                for a in [S*k+l for k in range(S-i) for l in range(S-i)]:
                        for b in [S*m+n for m in range(S-j) for n in range(S-j)]:
                                #calculating the coordinates of the squares
                                X_i = a%S
                                X_j = b%S
                                Y_i = a//S
                                Y_j = b//S
                                #calculating if squares are overlapping
                                if X_i >= X_j :
                                        if X_i - X_j < j+1:
                                                if Y_i >= Y_j:
                                                        if Y_i - Y_j < j+1:
                                                                ListConstraintsOverlaps.
↪append(1)
                                                        else:
                                                                ListConstraintsOverlaps.
↪append(0)
                                                else:
                                                        if Y_j - Y_i < i+1:
                                                                ListConstraintsOverlaps.
↪append(1)
                                                        else:
                                                                ListConstraintsOverlaps.
↪append(0)
                                        else:
                                                ListConstraintsOverlaps.append(0)
                                else :
                                        if X_j - X_i < i+1:
                                                if Y_i >= Y_j:
                                                        if Y_i - Y_j < j+1:
                                                                ListConstraintsOverlaps.
↪append(1)
                                                        else:
                                                                ListConstraintsOverlaps.
↪append(0)
                                                else:
                                                        if Y_j - Y_i < i+1:
                                                                ListConstraintsOverlaps.
↪append(1)
                                                        else:
                                                                ListConstraintsOverlaps.
↪append(0)
                                        else:
                                                ListConstraintsOverlaps.append(0)
                Problem.AddFunction(['sq' + str(i+1), 'sq' + str(j+1)],␣
↪ListConstraintsOverlaps)
```

(continues on next page)

(continued from previous page)

```
#Problem.Dump('Square.cfn')
Problem.CFN.timer(300)
res = Problem.Solve(showSolutions=3)
if res:
        for i in range(S):
                row = ''
                for j in range(S):
                        row += ' '
                        for k in range(N-1, -1, -1):
                                if (res[0][k]%(S-k) <= j and j - res[0][k]%(S-k) <= k)␣
↪and (res[0][k]//(S-k) <= i and i - res[0][k]//(S-k) <= k):
                                        row = row[:-1] + chr(65 + k)
                print(row)
else:
        print('No solution found!')
```

10.10.4 C++ program using libtb2.so

The following code uses the C++ toulbar2 library. 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
// 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
```

(continues on next page)

(continued from previous page)

```
    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()) {
            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;
    }
```

(continues on next page)

(continued from previous page)

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

10.11 Square soft packing problem

10.11.1 Brief description

The problem is almost identical to the square packing problem with the difference that we now allow overlaps but we want to minimize them.

10.11.2 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.

To calculate an initial upper bound we simply compute the worst case scenario where N squares of size N*N are all stacked together. The cost of this is N**4, so we will take N**4+1 as the initial upper bound.

10.11.3 Python model

The following code using pytoulbar2 library solves the corresponding cost function network (e.g. "python3 squaresoft.py 10 20").

squaresoft.py

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

import pytoulbar2
try:
        N = int(sys.argv[1])
        S = int(sys.argv[2])
        assert N <= S
except:
        print('Two integers need to be given as arguments: N and S')
        exit()

Problem = pytoulbar2.CFN(N**4 + 1)

#create a variable for each square
for i in range(N):
        Problem.AddVariable('sq' + str(i+1), ['(' + str(l) + ',' + str(j) + ')' for l in␣
↪range(S-i) for j in range(S-i)])


#binary soft constraints for overlapping squares
```

(continues on next page)

(continued from previous page)

```
for i in range(N):
        for j in range(i+1,N):
                ListConstraintsOverlaps = []
                for a in [S*k+l for k in range(S-i) for l in range(S-i)]:
                        for b in [S*m+n for m in range(S-j) for n in range(S-j)]:
                                #calculating the coordinates of the squares
                                X_i = a%S
                                X_j = b%S
                                Y_i = a//S
                                Y_j = b//S
                                #calculating if squares are overlapping
                                if X_i >= X_j :
                                        if X_i - X_j < j+1:
                                                if Y_i >= Y_j:
                                                        if Y_i - Y_j < j+1:
                                                                ListConstraintsOverlaps.
↪append(min(j+1-(X_i - X_j),i+1)*min(j+1-(Y_i - Y_j),i+1))
                                                        else:
                                                                ListConstraintsOverlaps.
↪append(0)
                                                else:
                                                        if Y_j - Y_i < i+1:
                                                                ListConstraintsOverlaps.
↪append(min(j+1-(X_i - X_j),i+1)*min(i+1-(Y_j - Y_i),j+1))
                                                        else:
                                                                ListConstraintsOverlaps.
↪append(0)
                                        else:
                                                ListConstraintsOverlaps.append(0)
                                else :
                                        if X_j - X_i < i+1:
                                                if Y_i >= Y_j:
                                                        if Y_i - Y_j < j+1:
                                                                ListConstraintsOverlaps.
↪append(min(i+1-(X_j - X_i),j+1)*min(j+1-(Y_i - Y_j),i+1))
                                                        else:
                                                                ListConstraintsOverlaps.
↪append(0)
                                                else:
                                                        if Y_j - Y_i < i+1:
                                                                ListConstraintsOverlaps.
↪append(min(i+1-(X_j - X_i),j+1)*min(i+1-(Y_j - Y_i),j+1))
                                                        else:
                                                                ListConstraintsOverlaps.
↪append(0)
                                        else:
                                                ListConstraintsOverlaps.append(0)
                Problem.AddFunction(['sq' + str(i+1), 'sq' + str(j+1)],␣
↪ListConstraintsOverlaps)

#Problem.Dump('SquareSoft.cfn')
Problem.CFN.timer(300)
```

(continues on next page)

(continued from previous page)

```
res = Problem.Solve(showSolutions=3)
if res:
        for i in range(S):
                row = ''
                for j in range(S):
                        row += ' '
                        for k in range(N-1, -1, -1):
                                if (res[0][k]%(S-k) <= j and j - res[0][k]%(S-k) <= k)␣
↪and (res[0][k]//(S-k) <= i and i - res[0][k]//(S-k) <= k):
                                        row = row[:-1] + chr(65 + k)
                print(row)
else:
        print('No solution found!')
```

10.11.4 C++ program using libtb2.so

The following code uses the C++ toulbar2 library. 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 -I./src -L./lib/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
```

(continues on next page)

(continued from previous page)

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

    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)

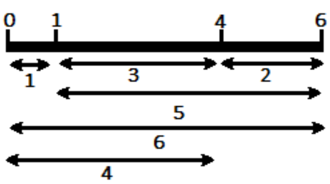
```
    delete solver;
    return 0;
}
```

10.12 Golomb ruler problem

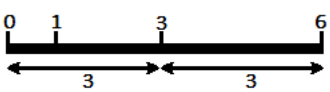
10.12.1 Brief description

A golomb ruler of order N is a set of integer marks 0=a1<a2<a3<a4<....<aN such that each difference between two ak's is unique.

For example, this is a golomb ruler:



We can see that all differences are unique, rather than in this other ruler where 0-3 and 3-6 are both equal to 3.



The size of a golomb ruler is equal to aN, the greatest number of the ruler. The goal is to find the smallest golomb ruler given N.

10.12.2 CFN model

We create N variables, one for each integer mark ak. Because we can not create an AllDifferent constraint with differences of variables directly, we also create a variable for each difference and create hard ternary constraints in order to force them be equal to the difference. Because we do not use an absolute value when creating the hard constraints, it forces the assignment of ak's variables to follow an increasing order.

Then we create an AllDifferent constraint on all the difference variables and one unary cost function on the last aN variable in order to minimize the size of the ruler. In order to break symmetries, we set the first mark to be zero.

10.12.3 Python model

The following code using pytoulbar2 library solves the golomb ruler problem with the first argument being the number of marks N (e.g. "python3 golomb.py 8").

golomb.py

```
import sys
import pytoulbar2
```

(continues on next page)

(continued from previous page)

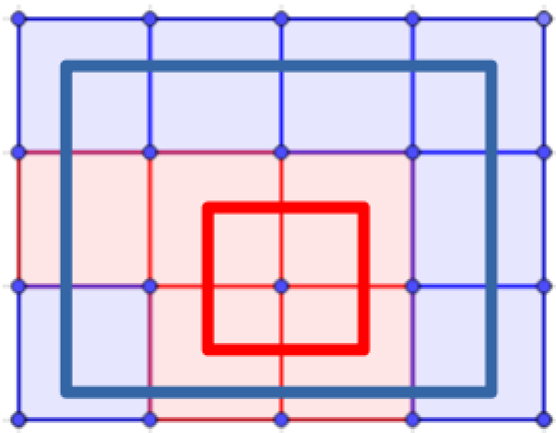
```
N = int(sys.argv[1])

top = N**2 + 1

Problem = pytoulbar2.CFN(top)

#create a variable for each mark
for i in range(N):
    Problem.AddVariable('X' + str(i), range(N**2))

#ternary constraints to link new variables of difference with the original variables
for i in range(N):
    for j in range(i+1, N):
        Problem.AddVariable('X' + str(j) + '-X' + str(i), range(N**2))
        Constraint = []
        for k in range(N**2):
            for l in range(N**2):
                for m in range(N**2):
                    if l-k == m:
                        Constraint.append(0)
                    else:
                        Constraint.append(top)
        Problem.AddFunction(['X' + str(i), 'X' + str(j), 'X' + str(j) + '-X' + str(i)],
↪Constraint)

Problem.AddAllDifferent(['X' + str(j) + '-X' + str(i) for i in range(N) for j in
↪range(i+1,N)])

Problem.AddFunction(['X' + str(N-1)], range(N**2))

#fix the first mark to be zero
Problem.AddFunction(['X0'], [0] + [top] * (N**2 - 1))

#Problem.Dump('golomb.cfn')
Problem.CFN.timer(300)
res = Problem.Solve(showSolutions=3)
if res:
    ruler = '0'
    for i in range(1,N):
        ruler += ' '*(res[0][i]-res[0][i-1]-1) + str(res[0][i])
    print('Golomb ruler of size:',int(res[1]))
    print(ruler)
```

10.13 Board coloration problem

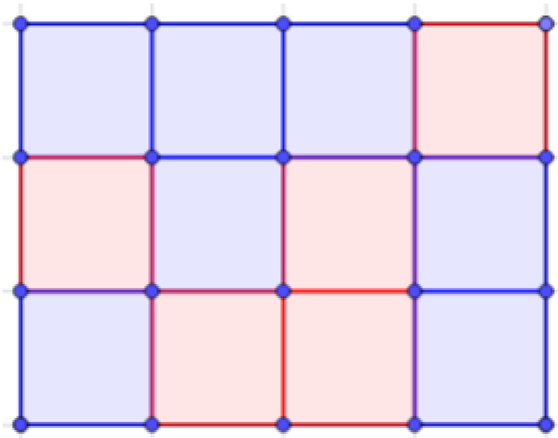
10.13.1 Brief description

Given a rectangular board with dimension n*m, the goal is to color the cells such that any inner rectangle included inside the board doesn't have all its corners colored with the same color. The goal is to minimize the number of colors used.

For example, this is not a valid solution of the 3*4 problem, because the red and blue rectangles have both their 4 corners having the same color:



On the contrary, the following coloration is a valid solution of the 3*4 problem because every inner rectangle inside the board does not have a unique color for its corners:



10.13.2 CFN basic model

We create n*m variables, one for each square of the board, with domain size equal to n*m representing all the possible colors. We also create one variable for the number of colors.

We create hard quaternary constraints for every rectangle inside the board with a cost equal to 0 if the 4 variables have different values and a forbidden cost if not.

We then create hard binary constraints between the variable of the number of colors for each cell to fix the variable for the number of colors as an upper bound.

Then we create a soft constraint on the number of colors to minimize it.

10.13.3 Python model

The following code using pytoulbar2 library solves the board coloration problem with the first two arguments being the dimensions n and m of the board (e.g. "python3 boardcoloration.py 3 4").

boardcoloration.py

```
import sys
from random import randint, seed
seed(123456789)
import pytoulbar2

try:
    n = int(sys.argv[1])
    m = int(sys.argv[2])
except:
    print('Two integers need to be in arguments: number of rows n, number of columns m')
    exit()

top = n*m + 1

Problem = pytoulbar2.CFN(top)

#create a variable for each cell
for i in range(n):
    for j in range(m):
        Problem.AddVariable('sq_' + str(i) + '_' + str(j), range(n*m))

#create a variable for the maximum of colors
Problem.AddVariable('max', range(n*m))

#quaterny hard constraints for rectangle with same color angles (encoding with forbidden␣
↪tuples)
ConstraintTuples = []
ConstraintCosts = []
for k in range(n*m):
    #if they are all the same color
    ConstraintTuples.append([k, k, k, k])
    ConstraintCosts.append(top)
#for each cell on the chessboard
for i1 in range(n):
    for i2 in range(m):
        #for every cell on the chessboard that could form a valid rectangle with the␣
↪first cell as up left corner and this cell as down right corner
        for j1 in range(i1+1, n):
            for j2 in range(i2+1, m):
                # add a compact function with zero default cost and only forbidden tuples
                Problem.AddCompactFunction(['sq_' + str(i1) + '_' + str(i2), 'sq_' +␣
↪str(i1) + '_' + str(j2), 'sq_' + str(j1) + '_' + str(i2), 'sq_' + str(j1) + '_' +␣
↪str(j2)], 0, ConstraintTuples, ConstraintCosts)
```

(continues on next page)

(continued from previous page)

```
#binary hard constraints to fix the variable max as an upper bound
Constraint = []
for k in range(n*m):
    for l in range(n*m):
        if k>l:
            #if the color of the square is more than the number of the max
            Constraint.append(top)
        else:
            Constraint.append(0)
for i in range(n):
    for j in range(m):
        Problem.AddFunction(['sq_' + str(i) + '_' + str(j), 'max'], Constraint)

#minimize the number of colors
Problem.AddFunction(['max'], range(n*m))

#symmetry breaking on colors
for i in range(n):
    for j in range(m):
        Constraint = []
        for k in range(n*m):
            if k > i*m+j:
                Constraint.append(top)
            else:
                Constraint.append(0)
        Problem.AddFunction(['sq_' + str(i) + '_' + str(j)], Constraint)

#Problem.Dump('boardcoloration.cfn')
Problem.CFN.timer(300)
res = Problem.Solve(showSolutions = 3)
if res:
    for i in range(n):
        row = []
        for j in range(m):
            row.append(res[0][m*i+j])
        print(row)
else:
    print('No solution found!')
```

10.14 Learning to play the Sudoku

10.14.1 Available

- Presentation
- GitHub code
- Data GitHub code

10.15 Learning car configuration preferences

10.15.1 Brief description

Renault car configuration system: learning user preferences.

10.15.2 Available

- Presentation
- GitHub code
- Data GitHub code

10.16 Visual Sudoku Tutorial

10.16.1 Brief description

A simple case mixing **Deep Learning** and **Graphical models**.

10.16.2 Available

- You can run it directly from your browser as a Jupyter Notebook

10.17 Visual Sudoku Application

10.17.1 Brief description

An automatic Sudoku puzzle **solver** using **OpenCV**, **Deep Learning**, and **Optical Character Recognition** (**OCR**).

10.17.2 Available

Software

Software adapted by Simon de Givry (@ INRAE, 2022) in order to use **toulbar2** solver, from a tutorial by Adrian Rosebrock (@ PyImageSearch, 2022) : GitHub code

As an APK

Based on this software, a 'Visual Sudoku' application for Android has been developed to be used from a smartphone.

See the *detailed presentation* (description, source, download…).

The application allows to capture a grid from its own camera (*'CAMERA' menu*) or to select a grid among the smartphone existing files (*'FILE' menu*), for example files coming from 'DCIM', in .jpg or .png formats. The grid image must have been captured in portrait orientation. Once the grid has been chosen, the *'Solve'* button allows to get the solution.

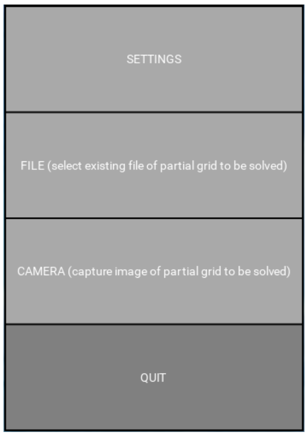
Fig.1 

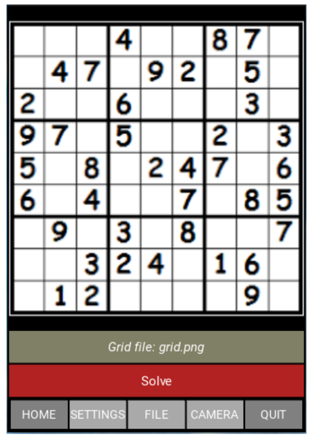
Fig.2 

Fig.3 

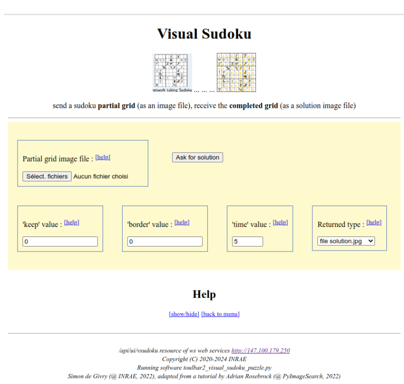
- Fig.1 : Screen of main menu
- Fig.2 : Screen of the grid to be solved
- Fig.3 : Screen of the solution (in yellow) found by the solver

`Examples of some input grids and their solved grids`

As a Web service

The software is available as a web service. The **visual sudoku web service**, hosted by the ws web services (based on HTTP protocol), can be called by many ways : from a **browser** (like above), from any softwares written in a language supporting HTTP protocol (**Python**, **R**, **C++**, **Java**, **Php**…), from command line tools (**cURL**…)…

- Calling the visual sudoku web service from a **browser** :



api/ui/vsudoku

- Example of calling the visual sudoku web service from a **terminal** by **cURL** :

  Commands (*replace mygridfilename.jpg by your own image file name*) :

```
curl --output mysolutionfilename.jpg -F 'file=@mygridfilename.jpg' -F 'keep=40' -F 'border=15' http://147.100.179.250/api/tool/vsudoku
```

- The ‘Visual Sudoku’ APK calls the visual sudoku web service.

10.18 Visual Sudoku App for Android

10.18.1 A visual sudoku solver based on cost function networks

This application solves the sudoku problem from a smartphone by reading the grid using its camera. The cost function network solver toulbar2 is used to deal with the uncertainty on the digit recognition produced by the neural network. This uncertainty, combined with the sudoku logical rules, makes it possible to correct perceptual errors. It is particularly useful in the case of hand-written digits or poor image quality. It is also possible to solve a partially filled-in grid with printed and hand-written digits. The solver will always suggest a valid solution that best adapts to the retrieved digit information. It will naturally detect (a small number of) errors in a partially filled-in grid and could be used later as a diagnosis tool (future work). This software demonstration emphasizes the tight relation between constraint programming, computer vision, and deep learning.

We used the open-source C++ solver toulbar2 in order to find the maximum a posteriori solution of a constrained probabilistic graphical model. With its dedicated numerical (soft) local consistency bounds, toulbar2 outperforms traditional CP solvers on this problem. Grid perception and cell extraction are performed by the computer vision library OpenCV. Digit recognition is done by **Keras** and **TensorFlow**. The current android application is written in Python using the Kivy framework. It is inspired from a tutorial by Adrian Rosebrock. It uses the ws RESTful web services in order to run the solver.

See also : *Visual Sudoku Application*.

10.18.2 Source Code

GitHub code

10.18.3 Download and Install

To install the ‘Visual Sudoku’ application on smartphone :

1) **Download** the **visualsudoku-release.apk** APK file from Github repository :

   https://github.com/toulbar2/visualsudoku/releases/latest

2) Click on the downloaded **visualsudoku-release.apk** APK file to ask for **installation** (*you have to accept to ‘install anyway’ from unknown developer*).

3) In your parameter settings for the app, give permissions to the ‘Visual Sudoku’ application (smartphone menu ‘Parameters’ > ‘Applications’ > ‘Visual Sudoku’) : allow camera (required to capture grids), files and multimedia contents (required to save images as files). Re-run the app.

Warnings :

- The application may fail at first start and you may have to launch it twice.
- While setting up successfully, the application should have created itself the required 'VisualSudoku' folder (under the smartphone 'Internal storage' folder) but if not, you will have to create it by yourself manually.
- Since the application calls a web service, an internet connection is required.

10.18.4 Description

The 'SETTINGS' menu allows to save grids or solutions as image files ('savinginputfile', 'savingoutputfile' parameters) and to access to some 'expert' parameters in order to enhance the resolution process ('keep', 'border', 'time' parameters).

The application allows to capture a grid from its own camera (*'CAMERA' menu*) or to select a grid among the smartphone existing files (*'FILE' menu*), for example files coming from 'DCIM', in .jpg or .png formats. The grid image must have been captured in portrait orientation. Once the grid has been chosen, the *'Solve'* button allows to get the solution.

Fig.1

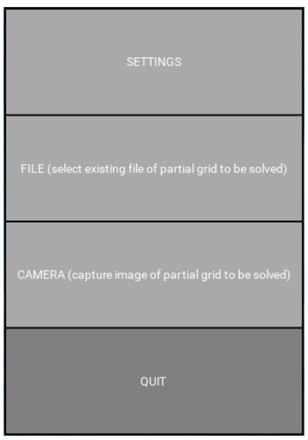


Fig.2

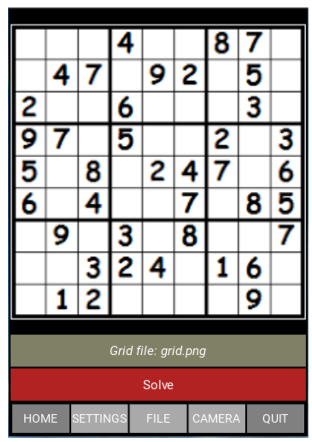
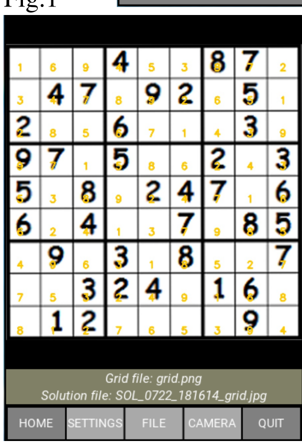


Fig.3



- Fig.1 : Screen of main menu
- Fig.2 : Screen of the grid to be solved
- Fig.3 : Screen of the solution (in yellow) found by the solver

`Examples of some input grids and their solved grids`

10.19 A sudoku code

10.19.1 Brief description

A Sudoku code returning a sudoku partial grid (sudoku problem) and the corresponding completed grid (sudoku solution), such as `partial and completed grids`.

The verbose version, that further gives a detailed description of what the program does, could be useful as tutorial example. Example : `partial and completed grids with explanations`.

10.19.2 Available

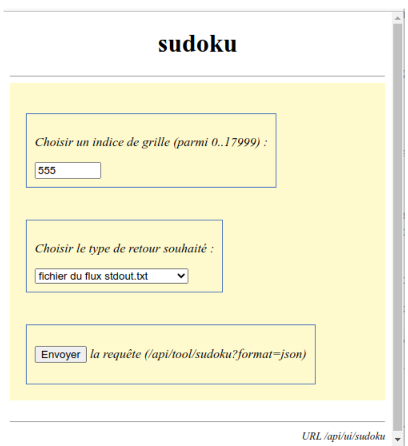
Available as a web service.

You can run the software directly from your **browser** as a web service :

Grids information is returned into the output stream. The **returned_type** parameter of the web service allows to choose how to receive it :

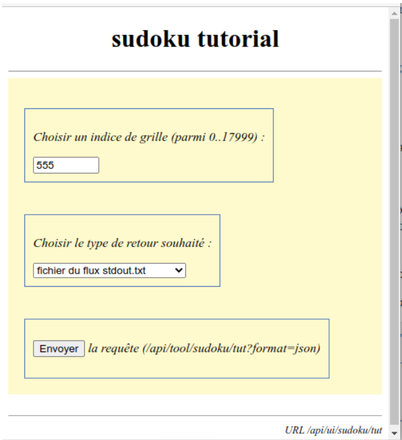
- *returned_type=stdout.txt* : to get the output stream as a .txt file.
- *returned_type=run.zip* : to get the .zip run folder containing the output stream __WS__stdout.txt (+ *the error stream __WS__stderr.txt that may be useful to investigate*).

Web service to get one sudoku grids (both partial and completed) :



api/ui/sudoku

Web service to further get a detailed description of what the program does (verbose version) :



api/ui/sudoku/tut (verbose version)

Note: The **sudoku web services**, hosted by the ws web services (based on HTTP protocol), can be called by many other ways : from a **browser** (like above), from any softwares written in a language supporting HTTP protocol (**Python**, **R**, **C++**, **Java**, **Php**. . .), from command line tools (**cURL**. . .). . .

Example of calling the sudoku web services from a terminal by cURL :

- Commands (*replace indice value by any value in 1. . . 17999*) :

```
curl --output mygrids.txt -F 'indice=778' -F 'returned_type=stdout.txt' http://147.100.179.250/api/tool/sudoku

curl --output myrun.zip -F 'indice=778' -F 'returned_type=run.zip' http://147.100.179.250/api/tool/sudoku

# verbose version

curl --output mygrids_details.txt -F 'indice=778' -F 'returned_type=stdout.txt' http://147.100.179.250/api/tool/sudoku/tut

curl --output myrun_details.zip -F 'indice=778' -F 'returned_type=run.zip' http://147.100.179.250/api/tool/sudoku/tut
```

- Responses corresponding with the requests above :
  - `mygrids.txt`
  - __WS__stdout.txt into myrun.zip has the same content as `mygrids.txt`
  - `mygrids_details.txt` (*__WS__stdout.txt into myrun_details.zip has the same content*)
  - __WS__stdout.txt into myrun_details.zip has the same content as `mygrids_details.txt`

CHAPTER

ELEVEN

USER GUIDE

11.1 What is toulbar2

toulbar2 is an exact black box discrete optimization solver targeted at solving cost function networks (CFN), thus solving the so-called “weighted Constraint Satisfaction Problem” or WCSP. Cost function networks can be simply described by a set of discrete variables each having a specific finite domain and a set of integer cost functions, each involving some of the variables. The WCSP is to find an assignment of all variables such that the sum of all cost functions is minimum and lest than a given upper bound often denoted as k or $\top$. Functions can be typically specified by sparse or full tables but also more concisely as specific functions called “global cost functions” [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. A parallel version of previous algorithm also exists [Beldjilali2022].

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.

11.2 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 (for more information, see *Installation from sources*). 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].

11.3 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
```

11.4 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.bz2**, **.cfn.xz**, **.wcsp**, **.wcsp.gz**, **.wcsp.bz2**, **.wcsp.xz**, **.wcnf**, **.wcnf.gz**, **.wcnf.bz2**, **.wcnf.xz**, **.cnf**, **.cnf.gz**, **.cnf.bz2**, **.cnf.xz**, **.qpbo**, **.qpbo.gz**, **.qpbo.bz2**, **.qpbo.xz**, **.opb**, **.opb.gz**, **.opb.bz2**, **.opb.xz**, **.uai**, **.uai.gz**, **.uai.bz2**, **.uai.xz**, **.LG**, **.LG.gz**, **.LG.bz2**, **.LG.xz**, **.xml**, **.xml.gz**, **.xml.bz2**, **.xml.xz**, **.pre** or **.bep**) is used to determine the nature of the file (see *Input 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.

11.5 Quick start

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

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

```
Read 25 variables, with 5 values at most, and 63 cost functions, with maximum arity␣
↪2.
Cost function decomposition time : 1.6e-05 seconds.
Reverse DAC dual bound: 20 (+10.000%)
Preprocessing time: 0.001 seconds.
24 unassigned variables, 116 values in all current domains (med. size:5, max␣
↪size:5) and 62 non-unary cost functions (med. arity:2, med. degree:5)
Initial lower and upper bounds: [20, 64] 68.750%
New solution: 28 (0 backtracks, 6 nodes, depth 8)
New solution: 27 (5 backtracks, 15 nodes, depth 5)
Optimality gap: [21, 27] 22.222 % (8 backtracks, 18 nodes)
Optimality gap: [22, 27] 18.519 % (21 backtracks, 55 nodes)
Optimality gap: [23, 27] 14.815 % (49 backtracks, 122 nodes)
Optimality gap: [24, 27] 11.111 % (63 backtracks, 153 nodes)
Optimality gap: [25, 27] 7.407 % (81 backtracks, 217 nodes)
Optimality gap: [27, 27] 0.000 % (89 backtracks, 240 nodes)
Node redundancy during HBFS: 25.417 %
Optimum: 27 in 89 backtracks and 240 nodes ( 460 removals by DEE) and 0.006 seconds.
end.
```

- 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
```

```
Read 25 variables, with 5 values at most, and 63 cost functions, with maximum arity␣
↪2.
Cost function decomposition time : 1.6e-05 seconds.
Reverse DAC dual bound: 20 (+10.000%)
```

(continues on next page)

(continued from previous page)

```
Preprocessing time: 0.001 seconds.
New solution: 27 (0 backtracks, 0 nodes, depth 1)
INCOP solving time: 0.254 seconds.
24 unassigned variables, 116 values in all current domains (med. size:5, max␣
↪size:5) and 62 non-unary cost functions (med. arity:2, med. degree:5)
Initial lower and upper bounds: [20, 27] 25.926%
Optimality gap: [21, 27] 22.222 % (4 backtracks, 8 nodes)
Optimality gap: [22, 27] 18.519 % (42 backtracks, 95 nodes)
Optimality gap: [23, 27] 14.815 % (93 backtracks, 209 nodes)
Optimality gap: [24, 27] 11.111 % (111 backtracks, 253 nodes)
Optimality gap: [25, 27] 7.407 % (121 backtracks, 280 nodes)
Optimality gap: [27, 27] 0.000 % (128 backtracks, 307 nodes)
Node redundancy during HBFS: 16.612 %
Optimum: 27 in 128 backtracks and 307 nodes ( 647 removals by DEE) and 0.263␣
↪seconds.
end.
```

- 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
```

```
Read 25 variables, with 5 values at most, and 63 cost functions, with maximum arity␣
↪2.
Cost function decomposition time : 1.6e-05 seconds.
Reverse DAC dual bound: 20 (+10.000%)
Preprocessing time: 0.001 seconds.
24 unassigned variables, 116 values in all current domains (med. size:5, max␣
↪size:5) and 62 non-unary cost functions (med. arity:2, med. degree:5)
Initial lower and upper bounds: [20, 28] 28.571%
New solution: 27 (0 backtracks, 4 nodes, depth 6)
Optimality gap: [21, 27] 22.222 % (6 backtracks, 14 nodes)
Optimality gap: [22, 27] 18.519 % (25 backtracks, 61 nodes)
Optimality gap: [23, 27] 14.815 % (56 backtracks, 133 nodes)
Optimality gap: [24, 27] 11.111 % (60 backtracks, 148 nodes)
Optimality gap: [25, 27] 7.407 % (83 backtracks, 228 nodes)
Optimality gap: [27, 27] 0.000 % (89 backtracks, 265 nodes)
Node redundancy during HBFS: 32.453 %
Optimum: 27 in 89 backtracks and 265 nodes ( 441 removals by DEE) and 0.007 seconds.
end.
```

- … and see this saved "example.sol" file:

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

```
1 0 2 2 2 2 0 4 2 0 4 1 0 0 3 0 3 1 2 4 2 1 2 4 1
```

- 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
```

```
Read 100 variables, with 44 values at most, and 1222 cost functions, with maximum␣
↪arity 2.
Cost function decomposition time : 0.000133 seconds.
Preprocessing time: 0.154752 seconds.
82 unassigned variables, 3273 values in all current domains (med. size:44, max␣
↪size:44) and 327 non-unary cost functions (med. arity:2, med. degree:6)
Initial lower and upper bounds: [0, 248338] 100.000%
--- [1] LDS 0 --- (0 nodes)
c 2097152 Bytes allocated for long long stack.
c 4194304 Bytes allocated for long long stack.
New solution: 7771 (0 backtracks, 101 nodes, depth 3)
--- [1] LDS 1 --- (101 nodes)
c 8388608 Bytes allocated for long long stack.
New solution: 5848 (1 backtracks, 282 nodes, depth 4)
New solution: 5384 (3 backtracks, 397 nodes, depth 4)
New solution: 5039 (4 backtracks, 466 nodes, depth 4)
New solution: 4740 (8 backtracks, 640 nodes, depth 4)
--- [1] LDS 2 --- (738 nodes)
New solution: 4675 (37 backtracks, 966 nodes, depth 5)
New solution: 4633 (44 backtracks, 1113 nodes, depth 5)
New solution: 4509 (45 backtracks, 1165 nodes, depth 5)
New solution: 4502 (51 backtracks, 1226 nodes, depth 5)
New solution: 4344 (54 backtracks, 1291 nodes, depth 5)
New solution: 4258 (135 backtracks, 1864 nodes, depth 5)
New solution: 4118 (136 backtracks, 1907 nodes, depth 5)
New solution: 4107 (138 backtracks, 1965 nodes, depth 4)
New solution: 4101 (147 backtracks, 2040 nodes, depth 5)
New solution: 4099 (150 backtracks, 2057 nodes, depth 4)
New solution: 4037 (152 backtracks, 2080 nodes, depth 5)
New solution: 3853 (157 backtracks, 2171 nodes, depth 5)
New solution: 3800 (209 backtracks, 2475 nodes, depth 5)
New solution: 3781 (222 backtracks, 2539 nodes, depth 5)
New solution: 3769 (226 backtracks, 2559 nodes, depth 5)
New solution: 3750 (227 backtracks, 2568 nodes, depth 5)
New solution: 3748 (229 backtracks, 2575 nodes, depth 5)
--- [1] LDS 4 --- (2586 nodes)
New solution: 3615 (663 backtracks, 5086 nodes, depth 7)
New solution: 3614 (698 backtracks, 5269 nodes, depth 6)
New solution: 3599 (704 backtracks, 5310 nodes, depth 6)
New solution: 3594 (708 backtracks, 5335 nodes, depth 7)
New solution: 3591 (709 backtracks, 5343 nodes, depth 6)
New solution: 3580 (710 backtracks, 5354 nodes, depth 7)
New solution: 3578 (716 backtracks, 5374 nodes, depth 6)
New solution: 3551 (988 backtracks, 6456 nodes, depth 7)
New solution: 3539 (996 backtracks, 6522 nodes, depth 7)
New solution: 3516 (1000 backtracks, 6554 nodes, depth 7)
New solution: 3507 (1002 backtracks, 6573 nodes, depth 7)
New solution: 3483 (1037 backtracks, 6718 nodes, depth 7)
New solution: 3464 (1038 backtracks, 6739 nodes, depth 7)
New solution: 3438 (1047 backtracks, 6806 nodes, depth 7)
```

(continues on next page)

(continued from previous page)

```
New solution: 3412 (1049 backtracks, 6824 nodes, depth 7)
--- [1] Search with no discrepancy limit --- (9443 nodes)
New solution: 3404 (4415 backtracks, 14613 nodes, depth 27)
New solution: 3402 (4416 backtracks, 14615 nodes, depth 25)
New solution: 3400 (4417 backtracks, 14619 nodes, depth 24)
New solution: 3391 (4419 backtracks, 14630 nodes, depth 28)
New solution: 3389 (4420 backtracks, 14632 nodes, depth 26)
Optimality gap: [100, 3389] 97.049 % (21663 backtracks, 49099 nodes)
Optimality gap: [300, 3389] 91.148 % (24321 backtracks, 54415 nodes)
Optimality gap: [957, 3389] 71.762 % (37965 backtracks, 81703 nodes)
Optimality gap: [1780, 3389] 47.477 % (39060 backtracks, 83893 nodes)
Optimality gap: [1999, 3389] 41.015 % (39252 backtracks, 84277 nodes)
Optimum: 3389 in 39276 backtracks and 84325 nodes ( 444857 removals by DEE) and 36.
↪293 seconds.
end.
```

- 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
```

```
Read 100 variables, with 44 values at most, and 1222 cost functions, with maximum␣
↪arity 2.
Cost function decomposition time : 9.2e-05 seconds.
Preprocessing time: 0.152035 seconds.
82 unassigned variables, 3273 values in all current domains (med. size:44, max␣
↪size:44) and 327 non-unary cost functions (med. arity:2, med. degree:6)
Initial lower and upper bounds: [0, 248338] 100.000%
c 2097152 Bytes allocated for long long stack.
c 4194304 Bytes allocated for long long stack.
c 8388608 Bytes allocated for long long stack.
New solution: 7566 (0 backtracks, 109 nodes, depth 110)
Problem decomposition in 55 clusters with size distribution: min: 1 median: 5 mean:␣
↪4.782 max: 12
****** Restart 1 with 1 discrepancies and UB=7566 ****** (109 nodes)
New solution: 7555 (0 backtracks, 109 nodes, depth 1)
New solution: 7545 (0 backtracks, 111 nodes, depth 2)
New solution: 7397 (0 backtracks, 114 nodes, depth 2)
New solution: 7289 (0 backtracks, 118 nodes, depth 2)
New solution: 7287 (0 backtracks, 118 nodes, depth 1)
New solution: 7277 (0 backtracks, 118 nodes, depth 1)
New solution: 5274 (0 backtracks, 118 nodes, depth 1)
New solution: 5169 (0 backtracks, 118 nodes, depth 1)
New solution: 5159 (0 backtracks, 118 nodes, depth 1)
New solution: 5158 (0 backtracks, 118 nodes, depth 1)
New solution: 5105 (1 backtracks, 120 nodes, depth 1)
New solution: 4767 (2 backtracks, 140 nodes, depth 2)
New solution: 4667 (2 backtracks, 140 nodes, depth 1)
New solution: 4655 (8 backtracks, 164 nodes, depth 2)
New solution: 4588 (8 backtracks, 171 nodes, depth 2)
New solution: 4543 (8 backtracks, 172 nodes, depth 2)
```

(continues on next page)

(continued from previous page)

```
New solution: 4541 (8 backtracks, 172 nodes, depth 1)
New solution: 4424 (8 backtracks, 174 nodes, depth 2)
New solution: 4423 (8 backtracks, 174 nodes, depth 1)
New solution: 4411 (8 backtracks, 174 nodes, depth 1)
New solution: 4401 (8 backtracks, 174 nodes, depth 1)
New solution: 4367 (8 backtracks, 175 nodes, depth 2)
New solution: 4175 (9 backtracks, 177 nodes, depth 1)
New solution: 4174 (9 backtracks, 177 nodes, depth 1)
New solution: 4173 (9 backtracks, 177 nodes, depth 1)
New solution: 4171 (9 backtracks, 177 nodes, depth 1)
New solution: 4152 (9 backtracks, 177 nodes, depth 1)
New solution: 4142 (12 backtracks, 187 nodes, depth 2)
New solution: 4001 (43 backtracks, 562 nodes, depth 2)
New solution: 3900 (43 backtracks, 562 nodes, depth 1)
New solution: 3891 (78 backtracks, 779 nodes, depth 1)
New solution: 3890 (80 backtracks, 788 nodes, depth 1)
New solution: 3816 (130 backtracks, 1192 nodes, depth 2)
New solution: 3768 (137 backtracks, 1217 nodes, depth 1)
New solution: 3740 (205 backtracks, 1660 nodes, depth 2)
New solution: 3738 (205 backtracks, 1660 nodes, depth 1)
New solution: 3730 (229 backtracks, 1780 nodes, depth 1)
New solution: 3723 (230 backtracks, 1786 nodes, depth 2)
New solution: 3721 (230 backtracks, 1786 nodes, depth 1)
New solution: 3711 (236 backtracks, 1819 nodes, depth 1)
New solution: 3633 (239 backtracks, 1850 nodes, depth 2)
New solution: 3628 (245 backtracks, 1941 nodes, depth 2)
New solution: 3621 (245 backtracks, 1943 nodes, depth 2)
New solution: 3609 (245 backtracks, 1943 nodes, depth 1)
New solution: 3608 (411 backtracks, 3079 nodes, depth 2)
New solution: 3600 (518 backtracks, 3775 nodes, depth 2)
New solution: 3598 (525 backtracks, 3806 nodes, depth 2)
New solution: 3597 (525 backtracks, 3806 nodes, depth 1)
New solution: 3587 (525 backtracks, 3806 nodes, depth 1)
New solution: 3565 (534 backtracks, 3846 nodes, depth 2)
New solution: 3554 (536 backtracks, 3856 nodes, depth 1)
New solution: 3534 (538 backtracks, 3860 nodes, depth 1)
New solution: 3522 (538 backtracks, 3861 nodes, depth 2)
New solution: 3507 (560 backtracks, 3987 nodes, depth 2)
New solution: 3505 (584 backtracks, 4130 nodes, depth 2)
New solution: 3500 (598 backtracks, 4255 nodes, depth 2)
New solution: 3498 (600 backtracks, 4281 nodes, depth 2)
New solution: 3493 (657 backtracks, 4648 nodes, depth 2)
****** Restart 2 with 2 discrepancies and UB=3493 ****** (6206 nodes)
New solution: 3492 (1406 backtracks, 9011 nodes, depth 3)
****** Restart 3 with 2 discrepancies and UB=3492 ****** (10128 nodes)
New solution: 3389 (1652 backtracks, 10572 nodes, depth 3)
****** Restart 4 with 2 discrepancies and UB=3389 ****** (11566 nodes)

Time limit expired... Aborting...
```

- 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
```

```
Read 200 variables, with 44 values at most, and 2665 cost functions, with maximum␣
↪arity 2.
Cost function decomposition time : 0.000303 seconds.
Reverse DAC dual bound: 10001 (+0.010%)
Preprocessing time: 0.351 seconds.
162 unassigned variables, 6481 values in all current domains (med. size:44, max␣
↪size:44) and 764 non-unary cost functions (med. arity:2, med. degree:8)
Initial lower and upper bounds: [10001, 436543501] 99.998%
c 2097152 Bytes allocated for long long stack.
c 4194304 Bytes allocated for long long stack.
c 8388608 Bytes allocated for long long stack.
New solution: 1455221 (0 backtracks, 232 nodes, depth 233)
Tree decomposition time: 0.003 seconds.
Problem decomposition in 25 clusters with size distribution: min: 3 median: 10␣
↪mean: 10.360 max: 38
****** Restart 1 with 1 discrepancies and UB=1455221 ****** (232 nodes)
New solution: 1445522 (0 backtracks, 232 nodes, depth 1)
New solution: 1445520 (0 backtracks, 232 nodes, depth 1)
New solution: 1445320 (0 backtracks, 232 nodes, depth 1)
New solution: 1445319 (0 backtracks, 232 nodes, depth 1)
New solution: 1435218 (0 backtracks, 232 nodes, depth 1)
New solution: 1425218 (0 backtracks, 232 nodes, depth 1)
New solution: 1425217 (0 backtracks, 232 nodes, depth 1)
New solution: 1415216 (0 backtracks, 232 nodes, depth 1)
New solution: 1405218 (0 backtracks, 232 nodes, depth 1)
New solution: 1405216 (9 backtracks, 286 nodes, depth 2)
New solution: 1395016 (9 backtracks, 286 nodes, depth 1)
New solution: 1394815 (9 backtracks, 289 nodes, depth 2)
New solution: 1394716 (9 backtracks, 289 nodes, depth 1)
New solution: 394818 (13 backtracks, 300 nodes, depth 1)
New solution: 394816 (13 backtracks, 300 nodes, depth 1)
New solution: 394716 (15 backtracks, 307 nodes, depth 1)
New solution: 394715 (26 backtracks, 361 nodes, depth 1)
New solution: 394713 (26 backtracks, 361 nodes, depth 1)
New solution: 384515 (30 backtracks, 379 nodes, depth 2)
New solution: 384513 (30 backtracks, 379 nodes, depth 1)
New solution: 384313 (30 backtracks, 379 nodes, depth 1)
New solution: 384213 (33 backtracks, 390 nodes, depth 1)
New solution: 384211 (33 backtracks, 390 nodes, depth 1)
New solution: 384208 (42 backtracks, 426 nodes, depth 1)
New solution: 384207 (42 backtracks, 427 nodes, depth 2)
New solution: 364206 (42 backtracks, 427 nodes, depth 1)
New solution: 353705 (42 backtracks, 438 nodes, depth 2)
New solution: 353703 (42 backtracks, 443 nodes, depth 2)
New solution: 353702 (44 backtracks, 450 nodes, depth 1)
New solution: 353701 (52 backtracks, 482 nodes, depth 1)
New solution: 343898 (88 backtracks, 705 nodes, depth 1)
New solution: 343698 (91 backtracks, 717 nodes, depth 1)
New solution: 343593 (94 backtracks, 726 nodes, depth 1)
****** Restart 2 with 2 discrepancies and UB=343593 ****** (1906 nodes)
```

(continues on next page)

(continued from previous page)

```
New solution: 343592 (319 backtracks, 2203 nodes, depth 3)
****** Restart 3 with 2 discrepancies and UB=343592 ****** (3467 nodes)

Time limit expired... Aborting...
```

- 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
```

```
Read 100 variables, with 4 values at most, and 710 cost functions, with maximum␣
↪arity 3.
Cost function decomposition time : 6.6e-05 seconds.
Reverse DAC dual bound: 64 (+35.938%)
Reverse DAC dual bound: 66 (+3.030%)
Reverse DAC dual bound: 67 (+1.493%)
Preprocessing time: 0.008 seconds.
88 unassigned variables, 228 values in all current domains (med. size:2, max␣
↪size:4) and 591 non-unary cost functions (med. arity:2, med. degree:13)
Initial lower and upper bounds: [67, 155] 56.774%
Tree decomposition width  : 19
Tree decomposition height : 43
Number of clusters        : 47
Tree decomposition time: 0.002 seconds.
New solution: 124 (20 backtracks, 35 nodes, depth 3)
Optimality gap: [70, 124] 43.548 % (20 backtracks, 35 nodes)
New solution: 123 (34 backtracks, 64 nodes, depth 3)
Optimality gap: [77, 123] 37.398 % (34 backtracks, 64 nodes)
New solution: 119 (173 backtracks, 348 nodes, depth 3)
Optimality gap: [88, 119] 26.050 % (173 backtracks, 348 nodes)
Optimality gap: [91, 119] 23.529 % (202 backtracks, 442 nodes)
New solution: 117 (261 backtracks, 609 nodes, depth 3)
Optimality gap: [97, 117] 17.094 % (261 backtracks, 609 nodes)
New solution: 114 (342 backtracks, 858 nodes, depth 3)
Optimality gap: [98, 114] 14.035 % (342 backtracks, 858 nodes)
Optimality gap: [100, 114] 12.281 % (373 backtracks, 984 nodes)
Optimality gap: [101, 114] 11.404 % (437 backtracks, 1123 nodes)
Optimality gap: [102, 114] 10.526 % (446 backtracks, 1152 nodes)
Optimality gap: [103, 114] 9.649 % (484 backtracks, 1232 nodes)
Optimality gap: [104, 114] 8.772 % (521 backtracks, 1334 nodes)
Optimality gap: [105, 114] 7.895 % (521 backtracks, 1353 nodes)
Optimality gap: [106, 114] 7.018 % (525 backtracks, 1364 nodes)
Optimality gap: [107, 114] 6.140 % (525 backtracks, 1379 nodes)
Optimality gap: [109, 114] 4.386 % (534 backtracks, 1539 nodes)
Optimality gap: [111, 114] 2.632 % (536 backtracks, 1559 nodes)
Optimality gap: [113, 114] 0.877 % (536 backtracks, 1564 nodes)
Optimality gap: [114, 114] 0.000 % (536 backtracks, 1598 nodes)
HBFS open list restarts: 0.000 % and reuse: 11.080 % of 352
Node redundancy during HBFS: 34.355 %
Optimum: 114 in 536 backtracks and 1598 nodes ( 21 removals by DEE) and 0.031␣
↪seconds.
end.
```

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

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

```
Read 100 variables, with 4 values at most, and 710 cost functions, with maximum␣
↪arity 3.
Cost function decomposition time : 6.6e-05 seconds.
Reverse DAC dual bound: 64 (+35.938%)
Reverse DAC dual bound: 66 (+3.030%)
Reverse DAC dual bound: 67 (+1.493%)
Preprocessing time: 0.008 seconds.
88 unassigned variables, 228 values in all current domains (med. size:2, max␣
↪size:4) and 591 non-unary cost functions (med. arity:2, med. degree:13)
Initial lower and upper bounds: [67, 155] 56.774%
Tree decomposition width  : 19
Tree decomposition height : 43
Number of clusters        : 47
Tree decomposition time: 0.002 seconds.
--- Solving cluster subtree 5 ...
New solution: 0 (0 backtracks, 0 nodes, depth 2)
---  done  cost = [0,0] (0 backtracks, 0 nodes, depth 2)

--- Solving cluster subtree 6 ...
New solution: 0 (0 backtracks, 0 nodes, depth 2)
---  done  cost = [0,0] (0 backtracks, 0 nodes, depth 2)

--- Solving cluster subtree 7 ...

...

--- Solving cluster subtree 44 ...
New solution: 42 (420 backtracks, 723 nodes, depth 7)
New solution: 39 (431 backtracks, 743 nodes, depth 9)
New solution: 35 (447 backtracks, 785 nodes, depth 22)
---  done  cost = [35,35] (557 backtracks, 960 nodes, depth 2)

--- Solving cluster subtree 46 ...
New solution: 114 (557 backtracks, 960 nodes, depth 2)
---  done  cost = [114,114] (557 backtracks, 960 nodes, depth 2)

Optimum: 114 in 557 backtracks and 960 nodes ( 50 removals by DEE) and 0.026␣
↪seconds.
end.
```

- 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
```

```
Read 240 variables, with 4 values at most, and 2242 cost functions, with maximum␣
↪arity 3.
Cost function decomposition time : 0.000911 seconds.
Preprocessing time: 0.013967 seconds.
```

(continues on next page)

(continued from previous page)

```
233 unassigned variables, 666 values in all current domains (med. size:2, max␣
↪size:4) and 1966 non-unary cost functions (med. arity:2, med. degree:16)
Initial lower and upper bounds: [2, 34347] 99.994%
Tree decomposition width  : 59
Tree decomposition height : 233
Number of clusters        : 239
Tree decomposition time: 0.017 seconds.
--- Solving cluster subtree 0 ...
New solution: 0 (0 backtracks, 0 nodes, depth 2)
---  done  cost = [0,0] (0 backtracks, 0 nodes, depth 2)

--- Solving cluster subtree 1 ...
New solution: 0 (0 backtracks, 0 nodes, depth 2)
---  done  cost = [0,0] (0 backtracks, 0 nodes, depth 2)

--- Solving cluster subtree 2 ...

...

--- Solving cluster subtree 3 ...
New solution: 21253 (26963 backtracks, 48851 nodes, depth 3)
New solution: 21251 (26991 backtracks, 48883 nodes, depth 4)
---  done  cost = [21251,21251] (26992 backtracks, 48883 nodes, depth 2)

--- Solving cluster subtree 238 ...
New solution: 21253 (26992 backtracks, 48883 nodes, depth 2)
---  done  cost = [21253,21253] (26992 backtracks, 48883 nodes, depth 2)

Optimum: 21253 in 26992 backtracks and 48883 nodes ( 0 removals by DEE) and 6.180␣
↪seconds.
end.
```

- 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
```

```
Read 240 variables, with 4 values at most, and 2242 cost functions, with maximum␣
↪arity 3.
Cost function decomposition time : 0.002201 seconds.
Reverse DAC dual bound: 11120 (+81.403%)
Reverse DAC dual bound: 11128 (+0.072%)
Preprocessing time: 0.079 seconds.
233 unassigned variables, 666 values in all current domains (med. size:2, max␣
↪size:4) and 1966 non-unary cost functions (med. arity:2, med. degree:16)
Initial lower and upper bounds: [11128, 34354] 67.608%
Tree decomposition time: 0.017 seconds.
Problem decomposition in 89 clusters with size distribution: min: 4 median: 11␣
↪mean: 11.831 max: 23
New solution: 26266 (0 backtracks, 59 nodes, depth 60)
New solution: 26265 in 0.038 seconds.
New solution: 26264 in 0.046 seconds.
```

(continues on next page)

(continued from previous page)

```
New solution: 25266 in 0.047 seconds.
New solution: 25265 in 0.060 seconds.
New solution: 25260 in 0.071 seconds.
New solution: 24262 in 0.080 seconds.
New solution: 23262 in 0.090 seconds.
New solution: 23260 in 0.098 seconds.
New solution: 23259 in 0.108 seconds.
New solution: 22262 in 0.108 seconds.
New solution: 22261 in 0.110 seconds.
New solution: 22260 in 0.113 seconds.
New solution: 22259 in 0.118 seconds.
New solution: 22258 in 0.128 seconds.
New solution: 22257 in 0.138 seconds.
New solution: 22255 in 0.154 seconds.
New solution: 22254 in 0.170 seconds.
New solution: 22252 in 0.206 seconds.
New solution: 21257 in 0.227 seconds.
New solution: 21256 in 0.256 seconds.
New solution: 21254 in 0.380 seconds.
New solution: 21253 in 0.478 seconds.
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 1 in communicator MPI_COMM_WORLD
with errorcode 0.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------

Time limit expired... Aborting...
```

- 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
```

```
Read 25 variables, with 5 values at most, and 63 cost functions, with maximum arity␣
↪2.
Cost function decomposition time : 1.6e-05 seconds.
Reverse DAC dual bound: 20 (+10.000%)
Preprocessing time: 0.001 seconds.
24 unassigned variables, 116 values in all current domains (med. size:5, max␣
↪size:5) and 62 non-unary cost functions (med. arity:2, med. degree:5)
Initial lower and upper bounds: [20, 64] 68.750%
New solution: 28 (0 backtracks, 6 nodes, depth 7)
Problem decomposition in 7 clusters with size distribution: min: 11 median: 15␣
↪mean: 15.143 max: 17
****** Restart 1 with 1 discrepancies and UB=28 ****** (6 nodes)
New solution: 27 (0 backtracks, 6 nodes, depth 1)
****** Restart 2 with 2 discrepancies and UB=27 ****** (57 nodes)
****** Restart 3 with 4 discrepancies and UB=27 ****** (143 nodes)
```

(continues on next page)

(continued from previous page)

```
****** Restart 4 with 8 discrepancies and UB=27 ****** (418 nodes)
****** Restart 5 with 16 discrepancies and UB=27 ****** (846 nodes)
Optimum: 27 in 521 backtracks and 1156 nodes ( 3066 removals by DEE) and 0.039␣
↪seconds.
end.
```

- 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
```

```
Read 25 variables, with 5 values at most, and 63 cost functions, with maximum arity␣
↪2.
Cost function decomposition time : 2.7e-05 seconds.
Reverse DAC dual bound: 20 (+10.000%)
Preprocessing time: 0.002 seconds.
24 unassigned variables, 116 values in all current domains (med. size:5, max␣
↪size:5) and 62 non-unary cost functions (med. arity:2, med. degree:5)
Initial lower and upper bounds: [20, 64] 68.750%
Problem decomposition in 7 clusters with size distribution: min: 11 median: 15␣
↪mean: 15.143 max: 17
New solution: 28 (0 backtracks, 7 nodes, depth 8)
New solution: 27 in 0.001 seconds.
Optimum: 27 in 0 backtracks and 7 nodes ( 36 removals by DEE) and 0.064 seconds.
Total CPU time = 0.288 seconds
Solving real-time = 0.071 seconds (not including preprocessing time)
end.
```

- 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
```

```
Read 25 variables, with 5 values at most, and 63 cost functions, with maximum arity␣
↪2.
Cost function decomposition time : 1.5e-05 seconds.
Reverse DAC dual bound: 20 (+10.000%)
Reverse DAC dual bound: 21 (+4.762%)
Preprocessing time: 0.001 seconds.
24 unassigned variables, 116 values in all current domains (med. size:5, max␣
↪size:5) and 62 non-unary cost functions (med. arity:2, med. degree:5)
Initial lower and upper bounds: [21, 64] 67.188%
Tree decomposition width  : 8
Tree decomposition height : 16
Number of clusters        : 18
Tree decomposition time: 0.000 seconds.
New solution: 29 (19 backtracks, 30 nodes, depth 3)
New solution: 28 (37 backtracks, 62 nodes, depth 3)
Optimality gap: [22, 28] 21.429 % (37 backtracks, 62 nodes)
Optimality gap: [23, 28] 17.857 % (309 backtracks, 629 nodes)
New solution: 27 (328 backtracks, 672 nodes, depth 3)
Optimality gap: [23, 27] 14.815 % (328 backtracks, 672 nodes)
```

(continues on next page)

(continued from previous page)

```
Optimality gap: [24, 27] 11.111 % (347 backtracks, 724 nodes)
Optimality gap: [25, 27] 7.407 % (372 backtracks, 819 nodes)
Optimality gap: [26, 27] 3.704 % (372 backtracks, 829 nodes)
Optimality gap: [27, 27] 0.000 % (372 backtracks, 873 nodes)
HBFS open list restarts: 0.000 % and reuse: 10.769 % of 65
Node redundancy during HBFS: 16.724 %
Optimum: 27 in 372 backtracks and 873 nodes ( 463 removals by DEE) and 0.020␣
↪seconds.
end.
```

- 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
```

```
Read 25 variables, with 5 values at most, and 63 cost functions, with maximum arity␣
↪2.
Warning! Cannot apply variable elimination during search with a given tree␣
↪decomposition file.
Warning! Cannot apply functional variable elimination with a given tree␣
↪decomposition file.
Cost function decomposition time : 1.6e-05 seconds.
Reverse DAC dual bound: 20 (+15.000%)
Reverse DAC dual bound: 22 (+9.091%)
Preprocessing time: 0.001 seconds.
25 unassigned variables, 120 values in all current domains (med. size:5, max␣
↪size:5) and 63 non-unary cost functions (med. arity:2, med. degree:5)
Initial lower and upper bounds: [22, 64] 65.625%
Tree decomposition width  : 16
Tree decomposition height : 24
Number of clusters        : 9
Tree decomposition time: 0.000 seconds.
New solution: 29 (23 backtracks, 29 nodes, depth 3)
New solution: 28 (32 backtracks, 46 nodes, depth 3)
Optimality gap: [23, 28] 17.857 % (37 backtracks, 58 nodes)
New solution: 27 (61 backtracks, 122 nodes, depth 3)
Optimality gap: [23, 27] 14.815 % (61 backtracks, 122 nodes)
Optimality gap: [24, 27] 11.111 % (132 backtracks, 269 nodes)
Optimality gap: [25, 27] 7.407 % (177 backtracks, 395 nodes)
Optimality gap: [26, 27] 3.704 % (189 backtracks, 467 nodes)
Optimality gap: [27, 27] 0.000 % (189 backtracks, 482 nodes)
HBFS open list restarts: 0.000 % and reuse: 25.926 % of 27
Node redundancy during HBFS: 25.519 %
Optimum: 27 in 189 backtracks and 482 nodes ( 95 removals by DEE) and 0.010 seconds.
end.
```

- 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
```

```
Read 1118 variables, with 7 values at most, and 1118 cost functions, with maximum␣
↪arity 4.
No evidence file specified. Trying EXAMPLES/pedigree9.uai.xz.evid
No evidence file.
Generic variable elimination of degree 4
Maximum degree of generic variable elimination: 4
Cost function decomposition time : 0.003733 seconds.
Preprocessing time: 0.073664 seconds.
232 unassigned variables, 517 values in all current domains (med. size:2, max␣
↪size:4) and 415 non-unary cost functions (med. arity:2, med. degree:6)
Initial lower and upper bounds: [553902779, 13246577453] 95.819%
Tree decomposition width  : 227
Tree decomposition height : 230
Number of clusters        : 890
Tree decomposition time: 0.047 seconds.
New solution: 865165767 energy: 298.395 prob: 2.564e-130 (72 backtracks, 140 nodes,␣
↪depth 3)
New solution: 844685630 energy: 296.347 prob: 1.987e-129 (128 backtracks, 254 nodes,
↪ depth 3)
New solution: 822713386 energy: 294.149 prob: 1.789e-128 (188 backtracks, 373 nodes,
↪ depth 3)
New solution: 809800912 energy: 292.858 prob: 6.506e-128 (327 backtracks, 665 nodes,
↪ depth 3)
New solution: 769281277 energy: 288.806 prob: 3.742e-126 (383 backtracks, 771 nodes,
↪ depth 3)
New solution: 755317979 energy: 287.410 prob: 1.512e-125 (714 backtracks, 1549␣
↪nodes, depth 3)
New solution: 755129381 energy: 287.391 prob: 1.540e-125 (927 backtracks, 2038␣
↪nodes, depth 3)
New solution: 711184893 energy: 282.997 prob: 1.248e-123 (1249 backtracks, 2685␣
↪nodes, depth 3)
HBFS open list restarts: 0.000 % and reuse: 39.620 % of 1474
Node redundancy during HBFS: 22.653 %
Optimum: 711184893 energy: 282.997 prob: 1.248e-123 in 21719 backtracks and 56124␣
↪nodes ( 72435 removals by DEE) and 4.310 seconds.
end.
```

- 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
```

```
Read 787 variables, with 7 values at most, and 3527 cost functions, with maximum␣
↪arity 3.
No evidence file specified. Trying EXAMPLES/GeomSurf-7-gm256.uai.xz.evid
No evidence file.
Cost function decomposition time : 0.001227 seconds.
Reverse DAC dual bound: 5879065363 energy: 1074.088 (+0.082%)
VAC dual bound: 5906374927 energy: 1076.819 (iter:486)
Number of VAC iterations: 726
Number of times is VAC: 240 Number of times isvac and itThreshold > 1: 234
Preprocessing time: 1.872 seconds.
```

(continues on next page)

(continued from previous page)

```
729 unassigned variables, 4819 values in all current domains (med. size:7, max␣
↪size:7) and 3128 non-unary cost functions (med. arity:2, med. degree:6)
Initial lower and upper bounds: [5906374927, 111615200815] 94.708%
c 2097152 Bytes allocated for long long stack.
New solution: 5968997522 energy: 1083.081 prob: 4.204e-471 (0 backtracks, 19 nodes,␣
↪depth 21)
Optimality gap: [5920086558, 5968997522] 0.819 % (17 backtracks, 36 nodes)
New solution: 5922481881 energy: 1078.430 prob: 4.404e-469 (17 backtracks, 48 nodes,
↪ depth 8)
Optimality gap: [5922481881, 5922481881] 0.000 % (21 backtracks, 52 nodes)
Number of VAC iterations: 846
Number of times is VAC: 360 Number of times isvac and itThreshold > 1: 351
Node redundancy during HBFS: 11.538 %
Optimum: 5922481881 energy: 1078.430 prob: 4.404e-469 in 21 backtracks and 52 nodes␣
↪( 2749 removals by DEE) and 1.980 seconds.
end.
```

- 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
```

```
Read 37 variables, with 350 values at most, and 703 cost functions, with maximum␣
↪arity 2.
No evidence file specified. Trying EXAMPLES/1CM1.uai.xz.evid
No evidence file.
Cost function decomposition time : 0.000679 seconds.
Reverse DAC dual bound: 103988236701 energy: -12486.138 (+0.000%)
VAC dual bound: 103988236701 energy: -12486.138 (iter:4068)
Number of VAC iterations: 4389
Number of times is VAC: 189 Number of times isvac and itThreshold > 1: 186
Threshold: 2326139858 NbAssignedVar: 0 Ratio: 0.0000000
Threshold: 2320178814 NbAssignedVar: 0 Ratio: 0.0000000
Threshold: 21288438 NbAssignedVar: 19 Ratio: 0.0000000
Threshold: 11823689 NbAssignedVar: 20 Ratio: 0.0000000
Threshold: 8187968 NbAssignedVar: 21 Ratio: 0.0000001
Threshold: 6858739 NbAssignedVar: 22 Ratio: 0.0000001
Threshold: 6058812 NbAssignedVar: 22 Ratio: 0.0000001
Threshold: 5504560 NbAssignedVar: 22 Ratio: 0.0000001
Threshold: 3972336 NbAssignedVar: 23 Ratio: 0.0000002
Threshold: 3655432 NbAssignedVar: 23 Ratio: 0.0000002
Threshold: 3067825 NbAssignedVar: 23 Ratio: 0.0000002
Threshold: 2174446 NbAssignedVar: 24 Ratio: 0.0000003
Threshold: 1641827 NbAssignedVar: 24 Ratio: 0.0000004
Threshold: 1376213 NbAssignedVar: 24 Ratio: 0.0000005
Threshold: 208082 NbAssignedVar: 24 Ratio: 0.0000031
Threshold: 104041 NbAssignedVar: 26 Ratio: 0.0000068
Threshold: 52020 NbAssignedVar: 27 Ratio: 0.0000140
Threshold: 26010 NbAssignedVar: 27 Ratio: 0.0000281
Threshold: 13005 NbAssignedVar: 27 Ratio: 0.0000561
Threshold: 6502 NbAssignedVar: 27 Ratio: 0.0001122
Threshold: 3251 NbAssignedVar: 27 Ratio: 0.0002245
```

(continues on next page)

(continued from previous page)

```
Threshold: 1625 NbAssignedVar: 27 Ratio: 0.0004491
Threshold: 812 NbAssignedVar: 27 Ratio: 0.0008987
Threshold: 406 NbAssignedVar: 27 Ratio: 0.0017974
Threshold: 203 NbAssignedVar: 27 Ratio: 0.0035947
Threshold: 101 NbAssignedVar: 27 Ratio: 0.0072250
Threshold: 50 NbAssignedVar: 27 Ratio: 0.0145946
Threshold: 25 NbAssignedVar: 27 Ratio: 0.0291892
Threshold: 12 NbAssignedVar: 27 Ratio: 0.0608108
Threshold: 6 NbAssignedVar: 27 Ratio: 0.1216216
Threshold: 3 NbAssignedVar: 27 Ratio: 0.2432432
Threshold: 1 NbAssignedVar: 27 Ratio: 0.7297297
RASPS/VAC threshold: 203
Preprocessing time: 41.340 seconds.
37 unassigned variables, 3366 values in all current domains (med. size:38, max␣
↪size:331) and 626 non-unary cost functions (med. arity:2, med. degree:35)
Initial lower and upper bounds: [103988236701, 239074057808] 56.504%
New solution: 104206588216 energy: -12464.303 prob: inf (0 backtracks, 3 nodes,␣
↪depth 6)
RASPS done in preprocessing (backtrack: 4 nodes: 8)
New solution: 104174014744 energy: -12467.560 prob: inf (4 backtracks, 12 nodes,␣
↪depth 6)
Optimality gap: [104174014744, 104174014744] 0.000 % (7 backtracks, 15 nodes)
Number of VAC iterations: 4695
Number of times is VAC: 458 Number of times isvac and itThreshold > 1: 451
Node redundancy during HBFS: 0.000 %
Optimum: 104174014744 energy: -12467.560 prob: inf in 7 backtracks and 15 nodes (␣
↪937 removals by DEE) and 41.354 seconds.
end.
```

- 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
```

```
c Read 200 variables, with 2 values at most, and 7011 clauses, with maximum arity 2.
Cost function decomposition time : 0.000485 seconds.
Reverse DAC dual bound: 91 (+86.813%)
Reverse DAC dual bound: 92 (+1.087%)
Preprocessing time: 0.040 seconds.
200 unassigned variables, 400 values in all current domains (med. size:2, max␣
↪size:2) and 6811 non-unary cost functions (med. arity:2, med. degree:68)
Initial lower and upper bounds: [92, 200] 54.000%
New solution: 189 (0 backtracks, 9 nodes, depth 11)
New solution: 188 (45 backtracks, 143 nodes, depth 37)
New solution: 187 (155 backtracks, 473 nodes, depth 47)
New solution: 186 (892 backtracks, 2247 nodes, depth 19)
New solution: 185 (3874 backtracks, 8393 nodes, depth 70)
New solution: 184 (29475 backtracks, 62393 nodes, depth 40)
New solution: 183 (221446 backtracks, 522724 nodes, depth 11)
Node redundancy during HBFS: 37.221 %
Optimum: 183 in 281307 backtracks and 896184 nodes ( 9478 removals by DEE) and 25.
↪977 seconds.
```

(continues on next page)

(continued from previous page)

```
end.
```

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

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

```
Read 16 variables, with 4 values at most, and 24 cost functions, with maximum arity␣
↪4.
Cost function decomposition time : 2e-06 seconds.
Reverse DAC dual bound: 48 (+2.083%)
Preprocessing time: 0.006 seconds.
16 unassigned variables, 64 values in all current domains (med. size:4, max size:4)␣
↪and 8 non-unary cost functions (med. arity:4, med. degree:6)
Initial lower and upper bounds: [48, 1000] 95.200%
Optimality gap: [49, 1000] 95.100 % (17 backtracks, 41 nodes)
Optimality gap: [58, 1000] 94.200 % (355 backtracks, 812 nodes)
Optimality gap: [72, 1000] 92.800 % (575 backtracks, 1309 nodes)
Optimality gap: [1000, 1000] 0.000 % (575 backtracks, 1318 nodes)
Number of solutions    : =  576
Time                   :    0.306 seconds
... in 575 backtracks and 1318 nodes
end.
```

- 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
```

```
Read 16 variables, with 4 values at most, and 24 cost functions, with maximum arity␣
↪4.
Cost function decomposition time : 3e-06 seconds.
Reverse DAC dual bound: 48 (+2.083%)
Preprocessing time: 0.009 seconds.
320 unassigned variables, 7968 values in all current domains (med. size:26, max␣
↪size:26) and 8 non-unary cost functions (med. arity:4, med. degree:0)
Initial lower and upper bounds: [48, 1000] 95.200%
+++++++++ Search for solution 1 +++++++++
New solution: 49 (0 backtracks, 7 nodes, depth 10)
New solution: 48 (2 backtracks, 11 nodes, depth 3)
Node redundancy during HBFS: 18.182 %
Optimum: 48 in 2 backtracks and 11 nodes ( 0 removals by DEE) and 0.017 seconds.
+++++++++ Search for solution 2 +++++++++
New solution: 52 (2 backtracks, 879 nodes, depth 871)
Optimality gap: [50, 49] -2.000 % (5 backtracks, 882 nodes)
New solution: 51 (5 backtracks, 1748 nodes, depth 868)
Optimality gap: [51, 49] -3.922 % (6 backtracks, 1749 nodes)
Node redundancy during HBFS: 0.172 %
Optimum: 51 in 6 backtracks and 1749 nodes ( 0 removals by DEE) and 0.046 seconds.
+++++++++ Search for solution 3 +++++++++
New solution: 74 (6 backtracks, 2569 nodes, depth 823)
New solution: 62 (14 backtracks, 3407 nodes, depth 824)
New solution: 58 (21 backtracks, 4245 nodes, depth 821)
```

(continues on next page)

(continued from previous page)

```
Optimality gap: [53, 49] -7.547 % (29 backtracks, 4270 nodes)
Optimality gap: [56, 49] -12.500 % (30 backtracks, 4276 nodes)
Optimality gap: [57, 49] -14.035 % (31 backtracks, 4292 nodes)
New solution: 57 (31 backtracks, 5114 nodes, depth 819)
Node redundancy during HBFS: 1.017 %
Optimum: 57 in 31 backtracks and 5114 nodes ( 0 removals by DEE) and 0.146 seconds.
+++++++++ Search for solution 4 +++++++++
New solution: 73 (44 backtracks, 5923 nodes, depth 773)
New solution: 72 (46 backtracks, 6702 nodes, depth 778)
New solution: 58 (53 backtracks, 7485 nodes, depth 773)
Optimality gap: [58, 49] -15.517 % (70 backtracks, 7584 nodes)
Node redundancy during HBFS: 1.846 %
Optimum: 58 in 70 backtracks and 7584 nodes ( 0 removals by DEE) and 0.256 seconds.
+++++++++ Search for solution 5 +++++++++
New solution: 80 (70 backtracks, 8307 nodes, depth 726)
New solution: 74 (100 backtracks, 9139 nodes, depth 728)
New solution: 66 (112 backtracks, 9896 nodes, depth 724)
New solution: 64 (116 backtracks, 10636 nodes, depth 725)
New solution: 58 (171 backtracks, 11654 nodes, depth 725)
Node redundancy during HBFS: 3.484 %
Optimum: 58 in 171 backtracks and 11654 nodes ( 0 removals by DEE) and 0.474␣
↪seconds.
+++++++++ Search for solution 6 +++++++++
New solution: 79 (178 backtracks, 12347 nodes, depth 677)
New solution: 76 (207 backtracks, 13102 nodes, depth 677)
New solution: 65 (212 backtracks, 13804 nodes, depth 680)
Optimality gap: [59, 49] -16.949 % (251 backtracks, 14053 nodes)
Optimality gap: [60, 49] -18.333 % (256 backtracks, 14093 nodes)
Optimality gap: [61, 49] -19.672 % (259 backtracks, 14126 nodes)
Optimality gap: [62, 49] -20.968 % (260 backtracks, 14165 nodes)
New solution: 62 (260 backtracks, 14849 nodes, depth 675)
Node redundancy during HBFS: 4.936 %
Optimum: 62 in 260 backtracks and 14849 nodes ( 0 removals by DEE) and 0.688␣
↪seconds.
+++++++++ Search for solution 7 +++++++++
c 2097152 Bytes allocated for long long stack.
New solution: 77 (267 backtracks, 15495 nodes, depth 630)
New solution: 76 (283 backtracks, 16160 nodes, depth 629)
New solution: 75 (334 backtracks, 16982 nodes, depth 628)
New solution: 68 (335 backtracks, 17615 nodes, depth 628)
Optimality gap: [64, 49] -23.438 % (383 backtracks, 17946 nodes)
New solution: 65 (383 backtracks, 18577 nodes, depth 627)
Optimality gap: [65, 49] -24.615 % (383 backtracks, 18581 nodes)
Node redundancy during HBFS: 5.915 %
Optimum: 65 in 383 backtracks and 18581 nodes ( 0 removals by DEE) and 0.963␣
↪seconds.
+++++++++ Search for solution 8 +++++++++
New solution: 81 (383 backtracks, 19161 nodes, depth 583)
New solution: 80 (425 backtracks, 19865 nodes, depth 583)
New solution: 69 (471 backtracks, 20646 nodes, depth 585)
New solution: 68 (479 backtracks, 21273 nodes, depth 581)
New solution: 65 (483 backtracks, 21881 nodes, depth 580)
```

(continues on next page)

(continued from previous page)

```
Node redundancy during HBFS: 6.014 %
Optimum: 65 in 483 backtracks and 21881 nodes ( 0 removals by DEE) and 1.175␣
↪seconds.
+++++++++ Search for solution 9 +++++++++
New solution: 68 (483 backtracks, 22413 nodes, depth 535)
Optimality gap: [66, 49] -25.758 % (581 backtracks, 22902 nodes)
New solution: 66 (581 backtracks, 23434 nodes, depth 531)
Node redundancy during HBFS: 6.900 %
Optimum: 66 in 581 backtracks and 23434 nodes ( 0 removals by DEE) and 1.379␣
↪seconds.
+++++++++ Search for solution 10 +++++++++
New solution: 68 (619 backtracks, 24035 nodes, depth 484)
Optimality gap: [67, 49] -26.866 % (686 backtracks, 24436 nodes)
Optimality gap: [68, 49] -27.941 % (686 backtracks, 24444 nodes)
Node redundancy during HBFS: 7.924 %
Optimum: 68 in 686 backtracks and 24444 nodes ( 0 removals by DEE) and 1.597␣
↪seconds.
+++++++++ Search for solution 11 +++++++++
New solution: 72 (714 backtracks, 24958 nodes, depth 436)
New solution: 68 (739 backtracks, 25534 nodes, depth 436)
Node redundancy during HBFS: 8.052 %
Optimum: 68 in 739 backtracks and 25534 nodes ( 0 removals by DEE) and 1.712␣
↪seconds.
+++++++++ Search for solution 12 +++++++++
c 4194304 Bytes allocated for long long stack.
New solution: 81 (770 backtracks, 26006 nodes, depth 389)
New solution: 78 (772 backtracks, 26399 nodes, depth 389)
New solution: 77 (779 backtracks, 26818 nodes, depth 389)
New solution: 76 (809 backtracks, 27354 nodes, depth 390)
New solution: 72 (858 backtracks, 28065 nodes, depth 389)
Optimality gap: [69, 49] -28.986 % (863 backtracks, 28122 nodes)
Optimality gap: [70, 49] -30.000 % (864 backtracks, 28130 nodes)
Optimality gap: [71, 49] -30.986 % (864 backtracks, 28140 nodes)
New solution: 71 (864 backtracks, 28532 nodes, depth 387)
Node redundancy during HBFS: 8.762 %
Optimum: 71 in 864 backtracks and 28532 nodes ( 0 removals by DEE) and 1.981␣
↪seconds.
+++++++++ Search for solution 13 +++++++++
New solution: 76 (898 backtracks, 28974 nodes, depth 343)
New solution: 72 (906 backtracks, 29334 nodes, depth 340)
Optimality gap: [72, 49] -31.944 % (979 backtracks, 29782 nodes)
Node redundancy during HBFS: 9.563 %
Optimum: 72 in 979 backtracks and 29782 nodes ( 0 removals by DEE) and 2.212␣
↪seconds.
+++++++++ Search for solution 14 +++++++++
New solution: 86 (1062 backtracks, 30429 nodes, depth 292)
New solution: 80 (1078 backtracks, 30768 nodes, depth 292)
New solution: 74 (1085 backtracks, 31080 nodes, depth 292)
Optimality gap: [74, 49] -33.784 % (1102 backtracks, 31203 nodes)
Node redundancy during HBFS: 10.124 %
Optimum: 74 in 1102 backtracks and 31203 nodes ( 0 removals by DEE) and 2.441␣
↪seconds.
```

(continues on next page)

(continued from previous page)

```
+++++++++ Search for solution 15 +++++++++
New solution: 79 (1103 backtracks, 31448 nodes, depth 246)
New solution: 78 (1122 backtracks, 31726 nodes, depth 246)
New solution: 76 (1183 backtracks, 32087 nodes, depth 245)
Optimality gap: [76, 49] -35.526 % (1231 backtracks, 32181 nodes)
Node redundancy during HBFS: 9.816 %
Optimum: 76 in 1231 backtracks and 32181 nodes ( 0 removals by DEE) and 2.603␣
↪seconds.
+++++++++ Search for solution 16 +++++++++
New solution: 80 (1253 backtracks, 32419 nodes, depth 197)
New solution: 79 (1315 backtracks, 32735 nodes, depth 197)
New solution: 78 (1336 backtracks, 32968 nodes, depth 196)
Optimality gap: [78, 49] -37.179 % (1349 backtracks, 32993 nodes)
Node redundancy during HBFS: 9.575 %
Optimum: 78 in 1349 backtracks and 32993 nodes ( 0 removals by DEE) and 2.760␣
↪seconds.
+++++++++ Search for solution 17 +++++++++
New solution: 80 (1349 backtracks, 33141 nodes, depth 151)
New solution: 79 (1374 backtracks, 33334 nodes, depth 149)
Optimality gap: [79, 49] -37.975 % (1474 backtracks, 33532 nodes)
Node redundancy during HBFS: 9.421 %
Optimum: 79 in 1474 backtracks and 33532 nodes ( 0 removals by DEE) and 2.924␣
↪seconds.
+++++++++ Search for solution 18 +++++++++
New solution: 80 (1546 backtracks, 33775 nodes, depth 102)
Optimality gap: [80, 49] -38.750 % (1592 backtracks, 33864 nodes)
Node redundancy during HBFS: 9.328 %
Optimum: 80 in 1592 backtracks and 33864 nodes ( 0 removals by DEE) and 3.085␣
↪seconds.
+++++++++ Search for solution 19 +++++++++
New solution: 80 (1687 backtracks, 34105 nodes, depth 54)
Node redundancy during HBFS: 9.263 %
Optimum: 80 in 1687 backtracks and 34105 nodes ( 0 removals by DEE) and 3.219␣
↪seconds.
+++++++++ Search for solution 20 +++++++++
Optimality gap: [1000, 49] -95.100 % (1809 backtracks, 34349 nodes)
Node redundancy during HBFS: 9.197 %
No solution in 1809 backtracks and 34349 nodes ( 0 removals by DEE) and 3.377␣
↪seconds.
end.
```

- 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:
```

```
Read 40 variables, with 6 values at most, and 78 cost functions, with maximum arity␣
↪2.
Cost function decomposition time : 1.1e-05 seconds.
Preprocessing time: 0.001019 seconds.
40 unassigned variables, 240 values in all current domains (med. size:6, max␣
```

(continues on next page)

(continued from previous page)

```
↪size:6) and 78 non-unary cost functions (med. arity:2, med. degree:4)
Initial lower and upper bounds: [0, 1] 100.000%
Tree decomposition width  : 5
Tree decomposition height : 20
Number of clusters        : 29
Tree decomposition time: 0.000 seconds.
Number of solutions    : =  411110802705928379432960
Number of #goods       :    3993
Number of used #goods  :    17190
Size of sep            :    4
Time                   :    0.055 seconds
... in 13689 backtracks and 27378 nodes
end.
```

- 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:
```

```
Read 40 variables, with 6 values at most, and 78 cost functions, with maximum arity␣
↪2.
Cost function decomposition time : 9e-06 seconds.
Preprocessing time: 0.000997 seconds.
40 unassigned variables, 240 values in all current domains (med. size:6, max␣
↪size:6) and 78 non-unary cost functions (med. arity:2, med. degree:4)
Initial lower and upper bounds: [0, 1] 100.000%

part 1 : 40 variables and 71 constraints (really added)
part 2 : 10 variables and 7 constraints (really added)
--> number of parts : 2
--> time : 0.000 seconds.

Tree decomposition width  : 5
Tree decomposition height : 17
Number of clusters        : 33
Tree decomposition time: 0.001 seconds.

Cartesian product                  :    13367494538843734031554962259968
Upper bound of number of solutions : <= 1719926784000000000000000
Number of solutions    : ~= 480000000000000000000000
Number of #goods       :    468
Number of used #goods  :    4788
Size of sep            :    3
Time                   :    0.011 seconds
... in 3738 backtracks and 7476 nodes
end.
```

11.6 Command line options

If you just execute:

```
toulbar2
```

toulbar2 will give you its (long) list of optional parameters, that you can see in part *'Available options'* of : `ToulBar2 Help Message`.

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.

We now describe in more detail toulbar2 optional parameters.

11.6.1 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)

-sigma=[float]
add a (truncated) random zero-centered gaussian noise 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)

11.6.2 Preprocessing

-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, it assumes a complete assignment – used as initial upper bound and as value heuristic – is read from default file "sol", or, without the option -x, given as input filename with ".sol" extension)

-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].

-amo automatically detects at-most-one constraints and adds them to existing knapsack/linear/pseudo-boolean constraints.

-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)

-hve=[integer]
hidden variable encoding with a given limit to the maximum domain size of hidden variables (default value is 0) A negative size limit means restoring the original encoding after preprocessing while keeping the improved dual bound. See also option -n to limit the maximum arity of dualized n-ary cost functions.

-pwc=[integer]
pairwise consistency by hidden variable encoding plus intersection constraints, each one bounded by a given maximum space limit (in MB) (default value is 0) A negative size limit means restoring the original encoding after preprocessing while keeping the improved dual bound. See also options -minqual, -hve to limit the domain size of hidden variables, and -n to limit the maximum arity of dualized n-ary cost functions.

-minqual	finds a minimal intersection constraint graph to achieve pairwise consistency (combine with option -pwc) (default option)

11.6.3 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*.

-pils=[“string”]
initial upper bound found by PILS local search solver. The string parameter is optional, using “3 0 0.333 100 500 10000 0.1 0.5 0.1 0.1” by default with the following meaning: *nbruns perturb_mode perturb_strength flatMaxIter nbEvalHC nbEvalMax strengthMin strengthMax incrFactor decrFactor*.

-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

11.6.4 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 16384)

-hbfsmin=[integer]
hybrid best-first search compromise between BFS and DFS minimum node redundancy threshold (alpha percentage, default value is 5%)

-hbfsmax=[integer]
hybrid best-first search compromise between BFS and DFS maximum node redundancy threshold (beta percentage default value is 10%)

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

-burst	in parallel HBFS, workers send their solutions and open nodes as soon as possible (by default) For using a parallel version of HBFS, after compiling with MPI option (cmake -DMPI=ON .) use "mpirun -n [NbOfProcess] toulbar2 problem.wcsp"

-eps=[integer|filename]
Embarrassingly parallel search mode. It outputs a given number of open nodes in -x format and exit (default value is 0). See ./misc/script/eps.sh to run them. Use this option twice to specify the output filename.

-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
- (-8) lexicographic ordering of variable names

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

-root=[integer]
root cluster heuristic (0:largest, 1:max. size/(height-size), 2:min. size/(height-size), 3:min. height) (default value is 0)

-minheight minimizes cluster tree height when searching for the root cluster (can be slow to perform)

-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)

-F=[integer]
merges clusters automatically to give more freedom to variable ordering heuristic in BTD-HBFS (-1: no merging, positive value: maximum iteration value for trying to solve the same subtree given its separator assignment before considering it as unmerged) (default value is -1)

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

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

11.6.5 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 DFBB and VNS-like methods if a better solution is found (default value is 0)

11.6.6 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 (any decimal cost below this threshold is considered as null by VAC thus speeding-up its convergence, 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)

-kpdp=[integer]
solves knapsack constraints using dynamic programming (-2: never, -1: only in preprocessing, 0: at every search node, >0: after a given number of nodes) (default value is -2)

11.6.7 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). A negative number will disconnect weighted degrees in embedded WeightedCSP constraints.

-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), DAC with special knapsack order (7), increasing arity (8), increasing arity then DAC (9), 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)

-bisupport=[float]
in bi-objective optimization with the second objective encapsulated by a bounding constraint (see WeightedCSPConstraint), the value heuristic chooses between both EAC supports of first (main) and second objectives by minimum weighted regret (if parameter is non-negative, it is used as the weight for the second objective) or always chooses the EAC support of the first objective (if parameter is zero) or always chooses the second objective (if parameter is negative, -1: for choosing EAC from the lower bound constraint, -2: from the upper bound constraint, -3: to favor the smallest gap, -4: to favor the largest gap) (default value is 0)

11.6.8 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, 3:Knapsack) (default value is 3)

-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)

11.6.9 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.

11.6.10 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, -2 or -4: saves after preprocessing but keeps initial domains (this option can be used in combination with -z=filename). If the problem is saved after preprocessing (except for -2 or -4), some variables may be lost (due to variable elimination, see -e or -p or -f).

11.6.11 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). It is used by CFN/UAI/QPBO/OPB/Pedigree formats. Note that in CFN format the number of significant digits is given in the problem description by default. This option allows to overwrite this default value.

-epsilon=[float]
approximation factor for computing the partition function (greater than 1, default value is infinity)

Note that in CFN format, costs are given as decimal numbers (the same for giving an initial upper bound, an absolute optimality gap or VAC threshold values) whereas in WCSP format costs are non-negative integers only.

11.6.12 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

  wcolor-{n}-{d}-0-{p2}-{seed} random weighted graph coloring problem

  - p2 is the number of edges in the graph

  vertexcover-{n}-{d}-{t1}-{p2}-{maxcost}-{seed} random vertex cover problem

  - t1 is the tightness (should be equal to 25)
  - p2 is the number of edges in the graph
  - maxcost each vertex has a weight randomly chosen between 0 and maxcost

  bivertexcover-{n}-{d}-{t1}-{p2}-{maxcost}-{ub2}-{seed} random bi-objective vertex cover problem

  - t1 is the tightness (should be equal to 25)
  - p2 is the number of edges in the graph
  - maxcost each vertex has two weights, both randomly chosen between 0 and maxcost

  - ub2 upper bound for the bounding constraint on the second objective (see epsilon-constraint method)

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*.

11.7 Input formats

11.7.1 Introduction

The available **file formats** (possibly compressed by gzip or bzip2 or xz, e.g., .cfn.gz, .wcsp.xz, .opb.bz2) are :

- Cost Function Network format (*.cfn* file extension)
- 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)
- Constraint Satisfaction and Optimization Problem (.xml file extension)

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 evalgm and Cost Function Library.

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). Warning, it uses the minimum of all initial upper bounds read from the problem files as the initial upper bound of the merged problem.

11.7.2 Formats details

CFN format (.cfn suffix)

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.

In a **cfn** file, a Cost Function Network is described as a JSON object with extra freedom and extra constraints.

Freedom:

- the double quotes around strings are not compulsory: both `"problem"` and `problem` are strings.
- double quotes can also be added around numbers: both `1.20` and `"1.20"` will be interpreted as decimal numbers.
- the commas that separate the fields inside an array or object are not compulsory. Any separator will do (comma, white space). So `[1, 2]` or `[1,2]` or `[1 2]` are all describing the same array.
- the delimiters for objects and arrays (`{}` and `[]`) can be used arbitrarily for both types of items.
- the colon (`:`) that separates the name of a field in an object from the contents of the field is not compulsory.
- It is possible to comment a line with a # the first position of a line.

Constraints:

- strings should not start with a character in `0123456789-.+` and cannot contain `/#[]{}:,` or a space character (tabs…).
- numbers can only be integers or decimals. No scientific notation.
- the order of fields inside an object is compulsory and cannot be changed.

A CFN is an object with 3 data: a definition of the main problem properties (tag `problem`), of variables and their domains (tag `variables`) and of cost functions (tag `functions`), in this order:

```
{ "problem": <problem properties>,
  "variables": <variables and domains>,
  "functions": <functions descriptions> }
```

Problem properties:

An object with two fields:

1. `"name"` : the name of the problem.
2. `"mustbe"` : specifies the direction of optimization and a global (upper/lower) bound on the objective. This is the concatenation of a comparator (> or <) immediately followed by a decimal number, described as a string. The comparator specifies the direction of optimization:

- "<": we are minimizing and the decimal indicates a global upper bound (all costs equal to or larger than this are considered as unfeasible).

- `">"`: we are maximizing and and the decimal indicates a global lower bound (all costs equal to or less than this are considered as unfeasible).

The number of significant digits in the decimal number gives the precision that will be used for all cost computations inside toulbar2.

As an example, `"mustbe":  "<10.00"` means that the CFN describes a function where all costs larger than or equal to 10.00 are considered as infinite. All costs will also be handled with 2 digits of precision after the decimal point.

The two fields must appear in this order:

```
{ "name": "test_problem", "mustbe": "<-12.100" }
```

or

```
{test.problem <12.100}
```

in a more concise non-JSON-compatible form.

Variables and domains:

An object with as many fields as variables. All fields must have different names. The contents of a variable field can be an array or an integer. An array gives the sequence of values (defined by their name) of the variable domain. An integer gives the domain cardinality, without naming values (values are represented by their position in the domain, starting at 0). If a negative domain size is given, the variable is an interval variable instead of a finite domain variable and it has domain [0,-domainsize-1].

```
{ "fdv1": ["a", "b", "c"], "fdv2" : 2, "iv1" : -100}
```

defines 3 variables, two finite domain variables and 1 interval variable. The first domain variable has 3 values, `"a"` `"b"` and `"c"`. the second has two anonymous values and the interval variable has domain [0,99].

As an extra freedom, it is possible to give no name to variables. This can be achieved using an array instead of an object. The example above can therefore be written:

```
[[a b c] 2 -100]
```

or even just

```
[3 2 -100]
```

in a dense non JSON-compatible format.

Functions:

An object with as many fields as functions. Every function is an object with different possible fields. All functions have a `scope` which is an array of variables (names or indices). The rest of the fields depends on the type of the cost function: table cost function or global (including arithmetic functions).

Table cost functions:

Sparse functions format:* useful for functions that are dominantly constant. A numerical `defaultcost` must be given after the scope. The `costs` table must be an array of tuple costs: a sequence of value names or indices followed by a numeric cost or `inf` to represent a forbidden tuple. The `defaultcost` is used to define the cost of any missing tuple.

```
{"scope": ["fdv1", "fdv2"],
 "defaultcost": 0.234,
 "costs": ["a", 0, 5,
```

(continues on next page)

(continued from previous page)

```
           "a", 1, 6.2,
           "c", 0, -7.21] }
```

is a possible sparse function definition. Here only 3 tuples are defined with their costs. All 3 remaining tuples will have cost `0.234`.

Dense function format: if the `defaultcost` tag is absent, a complete lexicographically ordered list of costs is expected instead.

```
{"scope": [ "fdv1", "fdv2" ],
 "costs": [4.2, 3.67, -12.1, 7.1, -3.1, 100.2] }
```

describes the 6 costs of the 6 tuples insides the cartesian product of the two variables `"fdv1"` and `"fdv2"`. To assign costs to tuples, all possible tuples of the cartesian product are lexicographically ordered using the declared value order in the domain of each variable. In the example above, the order over the six pairs will be `("a",0) ("a",1) ("b",0) ("b",1) ("c",0) ("c",1)` that will be associated to the costs `4.2, 3.67, -12.1, 7.1, -3.1` and `100.2` in this order. This lexicographic ordering is used for all arities.

Shared function format: If instead of an array, a string is given for the cost table, then this string must be the name of a yet undefined function. The actual function will have the same cost table as the future indicated function (on the specified scope). The domain sizes of the two functions must match.

```
{"scope": [ "v1", "v3" ],
 "costs": "f12" }
```

defines a function on variables `v1` and `v3` that will have the same cost table as the function i:code:*f12* that must be defined later in the file.

Global and arithmetic cost functions

These functions are defined by a `scope`, a `type` and `parameters`. The `type` is a string that defines the specific function to use, the `parameters` is an array of objects. The composition of the `parameters` depends on the `type` of the function.

At this point, in maximization mode, most of the global cost functions have restricted usage (with the exception of wregular).

Arithmetic functions:

These functions have all arity 2 and it is assumed here that these variables are called x and y . The values are considered as representing their index in the domain and are therefore integer. The `type` can be either:

- `">="` : with `parameters` array $[cst, \delta]$ where cst and δ are two costs, to express cost function $max(0, y + cst - x \leq \delta ? y + cst - x : upperbound)$. This is a soft inequality with hard threshold δ.
- `">"`: similar with a strict inequality and semantics $max(0, y + 1 + cst - x \leq \delta ? y + 1 + cst - x : upperbound)$
- `"<="`: similar with an inverted inequality and semantics: $max(0, x - cst - y \leq \delta ? x - cst - y : upperbound)$
- `"<"`: similar with a strict inequality and semantics $max(0, x - cst + 1 - y \leq \delta ? x - cst + 1 - y : upperbound)$
- `"="`: similar with an equality and semantics: similar with a strict inequality and semantics $|y + cst - x| \leq \delta ? |y + cst - x| : upperbound)$
- `"disj"`: takes a `parameters` array $[cstx, csty, w]$ to express soft binary disjunctive cost function with semantics $((x \geq y + csty) \vee (y \geq x + cstx))?0 : w)$
- `"sdisj"`: takes a `parameters` array $[cstx, csty, xmax, ymaxwxwy]$ to express a special disjunctive cost function with three implicit constraints $x \leq xmax, y \leq ymax$ and $(x < xmax \wedge y < ymax) \Rightarrow (x \geq y + csty \vee y \geq x + cstx)$ and an additional cost function $((x = xmax)?wx : 0) + ((y = ymax?wy : 0)$.

Example : arithmetic function with >= operator :

```
"arith0": {"scope": ["v5", "v6"],
           "type": ">=",
           "params": [1, 3]}
```

Global cost functions:

We use an informal syntactical description of each global cost function below. the "|" is used for alternative keywords and parentheses together with ?, * and + to denote optional or repeated groups of items (+ requires that at least one repetition exists). For more details on semantics and implementation, see:

1. Lee, J. H. M., & Leung, K. L. (2012). Consistency techniques for flow-based projection-safe global cost functions in weighted constraint satisfaction. *Journal of Artificial Intelligence Research*, 43, 257-292. *Artificial Intelligence*, 238, 166-189. 2. Allouche, D., Bessiere, C., Boizumault, P., De Givry, S., Gutierrez, P., Lee, J. H., … & Wu, Y. (2016). Tractability-preserving transformations of global cost functions. *Artificial Intelligence*, 238, 166-189.

Using a flow-based propagator:

- `salldiff`" with parameters array `[metric: "var"|"dec"|"decbi" cost: cost]` expresses a soft alldifferent 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 complete binary cost function network).

  – example :

  ```
  "f1": {"scope": ["v1" "v2" "v3" "v4"],
         "type": "salldiff",
         "params": {"metric": "var" "cost": 0.7}}
  ```

  generates a cost of 0.7 per variable assignment that needs to be changed for all variables to take a different value.

- `"sgcc"` with parameters array `[metric:"var"|"dec"|"wdec" cost: cost bounds: [[value lower_bound upper_bound (shortage_weight excess_weight)?]*]` expresses 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 (`value shortage` and `excess weights` penalties must be given iff `wdec` is used).

  – example :

  ```
  name: {scope: [v1 v2 v3 v4]
         type: sgcc
         params: {
            metric: wdec
            cost: 0.5
            bounds: [[0 1 2 0.2 0.2]
                     [1 3 4 0.2 0.1]]
            }
        }
  ```

- `"ssame"` with parameters array `[cost: cost vars1: [(variable)*] vars2: [(variable)*]]` to express a permutation constraint on two lists of variables of equal size with implicit variable-based cost semantic

  – example :

  ```
  name: {scope: [v1 v2 v3 v4]
         type : ssame
  ```

(continues on next page)

(continued from previous page)

```
        params : {
           cost : 6.2
           vars1 : [v1 v2]
           vars2 : [v3 v4]
           }
       }
```

- "sregular" with parameters array [metric: "var"|"edit" cost: cost starts: [(state)*] ends: [(state)*] 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 semantics with a given cost per violation followed by the definition of a deterministic finite automaton with arrays of initial and final states, and an array of state transitions where symbols are domain values indices.
  - example :

```
name: {scope: [v1 v2 v3 v4]
        type : sregular
        params : {
           metric: var
           cost: 1.0
           nb_states: 2
           starts: [0]
           ends: [0 1]
           transitions: [[0 0 0][0 1 1][1 1 1]]
           }
       }
```

Global cost functions using a dynamic programming DAG-based propagator:

- "sregulardp" with parameters array [metric: "var" cost: cost nb_states: nb_states starts: [(state)*] ends: [(state)*] transitions: [(start_state value_index 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 arrays of initial and final states, and an array of state transitions where symbols are domain value indices.
  - example: see sregular above.
- "sgrammar"|"sgrammardp" with parameters array [metric: "var"|"weight" cost: cost nb_symbols: nb_symbols nb_values: nb_values start: start_symbol terminals: [(terminal_symbol value (cost)?)*] non_terminals: [(nonterminal_in nonterminal_out_left nonterminal_out_right (cost)?)*] to express a soft/weighted grammar in Chomsky normal form. The costs inside the rules and terminals should be used only with the weight metric.
  - example:

```
name: {scope: [v1 v2 v3 v4]
        type : sgrammardp
        params: {
           metric : var
           cost : 1.012
           nb_symbols : 4
           nb_values : 2
           start : 0
           terminals : [[1 0][3 1]]
           non_terminals : [[0 0 0][0 1 2][0 1 3][2 0 3]]
```

(continues on next page)

(continued from previous page)

```
        }
    }
```

- `"samong"|"samongdp"` with parameters array `[metric: "var" cost: cost min: lower_bound max: upper_bound values: [(value)*]]` to express a soft among constraint to restrict the number of variables taking their value into a given set of value indices
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : samong
       params: {
          metric : var
          cost : 1.0
          min: 2
          max: 2
          values: [0]
          }
    }
```

- `"salldiffdp"` with parameters array `[metric: "var" cost: 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)
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: salldiffdp
       params: {
          metric: var
          cost: 0.7
          }
    }
```

- `"sgccdp"` with parameters array `[metric: "var" cost: "cost" bounds: [(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)
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: sgccdp
       params: {
          metric: var
          cost: 1.1
          bounds: [[0 0 1] [1 2 3]]
          }
    }
```

- `"max|smaxdp"` with parameters array `[defaultcost: defcost tuples: [(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)
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : smaxdp
       params: {
          defaultcost: 3
          tuples: [[0 0 4] [1 1 3][2 2 2][3 3 1]]
          }
       }
```

- "MST"|"smstdp" with empty parameters expresses a hard spanning tree constraint where each variable is assigned to its parent variable index in order to build a spanning tree (the root being assigned to itself)
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type: MST params: []}
```

Global cost functions using a cost function network-based propagator (decompose to bounded arity table cost functions):

- "wregular" with parameters `nb_states: nbstates starts: [[state cost]*] ends: [[state cost]*] transitions: [[state value_index 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 value indices
  - example :

```
name: {scope: [v1 v2 v4 v3]
       type : wregular
       params: {
          nb_states: 4
          starts : [[0 0.0][1 0.5]]
          ends : [[2 -1.0] [3 0.0]]
          transitions : [[0 0 1 0.5][0 1 2 0.0]
                         [2 0 2 1.0][1 1 3 -1.0]]
          }
       }
```

- "walldiff" with parameters array `[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.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : walldiff
       params: {
          metric: lin
          cost: 0.8
          }
      }
```

- "wgcc" with parameters metric: `hard|lin|quad cost: cost bounds: [[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
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : wgcc
       params: {
          metric: lin
          cost: 3.3
          bounds: [[0 0 1][1 2 2][2 0 1]]
          }
      }
```

- "wsame" with parameters a `metric: hard|lin|quad cost: 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
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type : wsame
        params: {
           metric: lin
           cost: 3.3
           }
      }
```

- "wsamegcc" with parameters array `metric: hard|lin|quad cost: cost bounds: [[value lower_bound upper_bound]*]` to express the combination of a soft global cardinality constraint and a permutation constraint.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : wsamegcc
       params: {
          metric: lin
          cost: 3.3
          bounds: [[0 0 1][1 0 1][2 0 1][3 0 0]]
          }
      }
```

- "wamong" with parameters `metric: hard|lin|quad cost: cost values: [(value)*] min: lower_bound max: upper_bound` to express a soft among constraint to restrict the number of variables taking their value into a given set of values.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: wamong
       params: {
          metric: lin
          cost: 1
          values: [0]
          min: 1
          max: 1
          }
      }
```

- "wvaramong" with parameters array `metric: hard cost: cost 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.
  - example:

```
name: {scope: [v1 v2 v3 v4 v5]
       type: wvaramong
       params: {
          metric: hard
          cost: 12.0
          values: [1]
          }
      }
```

- "woverlap" with parameters `metric: hard|lin|quad cost: cost comparator: comparator to: 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))
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: woverlap
       params: {
          metric: hard
          cost: 2.01comparator: >
          to: 1
          }
      }
```

- "wdiverse" with parameters `distance: integer values: [(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 (values).
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type : wdiverse
        params: {
           distance: 2
           values: [0 1 0 1]
           }
      }
```

- "whdiverse" with parameters `distance: integer values: [(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 (values).
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type : whdiverse
        params: {
           distance: 2
           values: [0 1 0 1]
```

(continues on next page)

(continued from previous page)

```
            }
       }
```

- "wtdiverse" with parameters `distance: integer values: [(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 (values).
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type : wtdiverse
        params: {
           distance: 2
           values: [0 1 0 1]
           }
      }
```

- "wsum" parameters `metric: hard|lin|quad cost: cost comparator: comparator to: 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.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: wsum
       params: {
          metric: quad
          cost: 1.0
          comparator: "<="
          to: 4
          }
      }
```

- "wvarsum" with parameters `metric: hard cost: cost comparator: comparator` to express a hard sum constraint to restrict the sum to be comparator to the value of the last variable in the scope.
  - example:

```
mywsum: {scope: [v1 v2 v3 v4]
         type : wvarsum
         params: {
            metric: hard
            cost: 3
            comparator: "=="
            }
        }
```

Comparators: let us note <> the comparator, K the right-hand-side (to:) 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`.

Global cost functions using a dedicated propagator:

- `"knapsack"` with parameters `capacity: capacity weights: [(coefficient)*]` to express a hard global reverse knapsack constraint (i.e., a linear constraint on 0/1 variables with >= operator) where capacity and coefficients (one for each variable in the scope) are positive or negative integers. Use negative numbers to express a linear constraint with <= operator. See below a simple example encoding `v1+v2+v3+v4 >= 1`.

  - example:

```
myknapsack: {scope: [v1 v2 v3 v4]
         type : knapsack
         params: {
            capacity: 1
            weights: [1 1 1 1]
            }
        }
```

- `"knapsackv"` with parameters `capacity: capacity weightedvalues: [([variable value coefficient])*]` to express a hard global reverse knapsack constraint (i.e., a generalized linear constraint on domain variables with >= operator) where capacity and coefficients are positive or negative integers. Use negative numbers to express a generalized linear constraint with <= operator. Variables can be names or indices in the whole problem. They must also belong to the scope. See below a simple example encoding `(v1=1)+(v2=1)+(v3=1)+(v4=1) >= 1`.

  - example:

```
myknapsackv: {scope: [v1 v2 v3 v4]
         type : knapsackv
         params: {
            capacity: 1
            weightedvalues: [[v1 1 1] [v2 1 1] [v3 1 1] [v4 1 1]]
            }
        }
```

- `"clique"` with parameters `rhs: 1 values: [([(value)*])*]` to express a hard global clique constraint to restrict the number of variables taking their value into a given set of values (one set per variable) to at most 1 occurrence for all the variables. A clique of binary constraints must also be added to forbid any two variables from using both the restricted values.

  - example:

```
f01: { scope: [v0 v1] defaultcost: 0 costs: [1 1 inf]}
f02: { scope: [v0 v2] defaultcost: 0 costs: [1 1 inf]}
f03: { scope: [v0 v3] defaultcost: 0 costs: [1 1 inf]}
f12: { scope: [v1 v2] defaultcost: 0 costs: [1 1 inf]}
f13: { scope: [v1 v3] defaultcost: 0 costs: [1 1 inf]}
f23: { scope: [v2 v3] defaultcost: 0 costs: [1 1 inf]}
myclique: {scope: [v0 v1 v2 v3]
         type : clique
```

(continues on next page)

(continued from previous page)

```
         params: {
            rhs: 1
            values: [[1], [1], [1], [1]]
            }
        }
```

- "cfnconstraint" with parameters `cfn: cost-function-network lb: cost ub: cost duplicatehard: value strongduality: value` to express a hard global constraint on the cost of an input weighted constraint satisfaction problem in cfn format such that its valid solutions must have a cost value in [lb,ub[.
  - "duplicatehard" (0|1): if true then it assumes any forbidden tuple in the original input problem is also forbidden by another constraint in the main model (you must duplicate any hard constraints in your input model into the main model).
  - "strongduality" (0|1): if true then it assumes the propagation is complete when all channeling variables in the scope are assigned and the semantic of the constraint enforces that the optimum and ONLY the optimum on the remaining variables is between lb and ub.
  - example :

```
name: {scope: [v1 v2 v4]
       type : cfnconstraint
       params: {
          cfn:
            {
            problem: {name: "subcfn", mustbe: "<1000.0"}
            variables: {v1:2, v2:2, v4:2}
            functions: {
               {scope: [v1], costs: [0.0, -3.0]},
               {scope: [v2], costs: [-1.0, 0.0]},
               {scope: [v4], costs: [0.0, 2.0]}}
            }
          lb : -1.0
          ub : 0.0
          duplicatehard: 0
          strongduality: 0
          }
       }
```

Warning: the same floatting-point precision and optimization sense (minimization or maximization) should be used by the encapsulated cost function network and the main model. Warning: the list of variables of the encapsulated cost function network should be exactly the same as the scope (and with the same order).

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>
```

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

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

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 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)
  - knapsackc *capacity* (*weight*)* *nb_AMO* (*nb_variables* (*variable value*)*)* to express a reverse knapsack constraint (i.e., a linear constraint on 0/1 variables with >= operator) combined with a list of non-overlapping at-most-one constraints
  - 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
  - knapsackv *capacity nb_triplets* (*variable value weight*)* to express a reverse knapsack constraint with a list of triplets variable, value, and its corresponding weight
  - wcsp *lb ub duplicatehard strongduality wcsp* to express a hard global constraint on the cost of an input weighted constraint satisfaction problem in wcsp format such that its valid solutions must have a cost value in [lb,ub[.
- 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);

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.

Note More about network-based global cost functions can be found on ./misc/doc/DecomposableGlobalCostFunctions.html

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
```

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

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

- 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
```

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

- wcsp constraint ($3 <= 2 * x1 * x2 + 3 * x1 * x4 + 4 * x2 * x4 < 5$) on three Boolean 0/1 variables:

```
3 1 2 4 -1 wcsp 3 5 0 0 name 3 2 3 1000 2 2 2 2 0 1 0 1 1 1 2 2 0 2 0 1 1 1 3 2␣
↪1 2 0 1 1 1 4
```

- 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
```

(continues on next page)

(continued from previous page)

```
3 1 5
3 3 5
2 0 3 0 6
0 0 5
0 3 5
1 1 5
2 2 5
3 0 5
3 3 5
2 1 2 0 10
0 0 5
0 1 5
1 0 5
1 1 5
1 2 5
2 1 5
2 2 5
2 3 5
3 2 5
3 3 5
2 1 3 0 8
0 0 5
0 2 5
1 1 5
1 3 5
2 0 5
2 2 5
3 1 5
3 3 5
2 2 3 0 10
0 0 5
0 1 5
1 0 5
1 1 5
1 2 5
2 1 5
2 2 5
2 3 5
3 2 5
3 3 5
1 0 0 2
1 1
3 1
1 1 0 2
1 1
2 1
1 2 0 2
1 1
2 1
1 3 0 2
0 1
2 1
```

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
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
 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
```

(continues on next page)

(continued from previous page)

```
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.

XCSP3 format (.xml)

CSP and COP format in XML format XCSP3 core can be read (still on-going work for including globals). See a description of this format here http://xcsp.org.

Warning, toulbar2 must be compiled with specific options XML and XCSP3 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.

11.8 How do I use it ?

11.8.1 Using it as a C++ library

See *toulbar2 Reference Manual* which describes the libtb2.so C++ library API.

11.8.2 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 version of toulbar2 was integrated inside Numberjack. See https://github.com/eomahony/Numberjack.

11.9 References

See 'BIBLIOGRAPHY' at the end of the document.

CHAPTER

TWELVE

REFERENCE MANUAL

12.1 Introduction

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

toulbar2 can be used as a stand-alone solver reading various problem file formats (wcsp, uai, wcnf, qpbo) or as a C++ library.

This document describes the WCSP native file format and the toulbar2 C++ library API.

Note

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

See also : `src/toulbar2test.cpp`.

12.2 Exact optimization for cost function networks and additive graphical models

12.2.1 What is toulbar2?

toulbar2 is an open-source black-box C++ optimizer for cost function networks and discrete additive graphical models. This also covers Max-SAT, Max-Cut, QUBO (and constrained variants), among others. 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)

- XCSP3 Competitions (second place on Mini COP and Parallel COP tracks in 2022, first place on Mini COP in 2023)
- UAI 2022 Inference Competition UAI 2022 (winner on all MPE and MMAP task categories)

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.

12.2.2 Installation from binaries

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

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

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

12.2.3 Python interface

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

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

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

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

```
pip3 install pybind11
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".

12.2.4 Download

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

```
git clone --branch 1.2.0 https://github.com/toulbar2/toulbar2.git
```

12.2.5 Installation from sources

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

Required library:

- libgmp-dev

Recommended libraries (default use):

- libboost-graph-dev
- libboost-iostreams-dev
- libboost-serialization-dev
- zlib1g-dev
- liblzma-dev
- libbz2-dev

Optional libraries:

- libjemalloc-dev
- pybind11-dev
- libopenmpi-dev
- libboost-mpi-dev
- libicuuc
- libicui18n
- libicudata
- libxml2-dev
- libxcsp3parser

On MacOS, run ./misc/script/MacOS-requirements-install.sh to install the recommended libraries. For Mac with ARM64, add option -DBoost=OFF to cmake.

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

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

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

```
bash
cd src
echo '#define Toulbar_VERSION "1.2.0"' > ToulbarVersion.hpp
g++ -o toulbar2 -std=c++17 -O3 -DNDEBUG -static -static-libgcc -static-libstdc++ -DBOOST␣
↪-DLONGDOUBLE_PROB -DLONGLONG_COST -DWCSPFORMATONLY \
 -I. -I./pils/src tb2*.cpp applis/*.cpp core/*.cpp globals/*.cpp incop/*.cpp mcriteria/*.
↪cpp pils/src/exe/*.cpp search/*.cpp utils/*.cpp vns/*.cpp ToulbarVersion.cpp \
 -lboost_graph -lboost_iostreams -lboost_serialization -lgmp -lz -lbz2 -llzma
```

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

```
bash
cd src
echo '#define Toulbar_VERSION "1.2.0"' > ToulbarVersion.hpp
mpicxx -o toulbar2 -std=c++17 -O3 -DNDEBUG -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -
↪DWCSPFORMATONLY -DOPENMPI \
 -I. -I./pils/src tb2*.cpp applis/*.cpp core/*.cpp globals/*.cpp incop/*.cpp mcriteria/*.
↪cpp pils/src/exe/*.cpp search/*.cpp utils/*.cpp vns/*.cpp ToulbarVersion.cpp \
 -lboost_graph -lboost_iostreams -lboost_serialization -lboost_mpi -lgmp -lz -lbz2 -llzma
```

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

Replace WCSPFORMATONLY by XMLFLAG3 and add libxcsp3parser.a from xcsp.org in your current directory for reading XCSP3 files:

```
bash
cd src
echo '#define Toulbar_VERSION "1.2.0"' > ToulbarVersion.hpp
mpicxx -o toulbar2 -std=c++17 -O3 -DNDEBUG -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -
↪DXMLFLAG3 -DOPENMPI \
 -I/usr/include/libxml2 -I. -I./pils/src -I./xmlcsp3 tb2*.cpp applis/*.cpp core/*.cpp␣
↪globals/*.cpp incop/*.cpp mcriteria/*.cpp pils/src/exe/*.cpp search/*.cpp utils/*.cpp␣
↪vns/*.cpp ToulbarVersion.cpp \
 -lboost_graph -lboost_iostreams -lboost_serialization -lboost_mpi -lxml2 -licuuc -
↪licui18n -licudata libxcsp3parser.a -lgmp -lz -lbz2 -llzma -lm -lpthread -ldl
```


12.3 Modules

12.3.1 Variable and cost function modeling

group **modeling**

Modeling a Weighted CSP consists in creating variables and cost functions.

Domains of variables can be of two different types:

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

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

Cost functions can be defined in extension (table or maps) or having a specific semantic.

Cost functions in extension depend on their arity:

- unary cost function (directly associated to an enumerated variable)

- binary and ternary cost functions (table of costs)
- n-ary cost functions (n >= 4) defined by a list of tuples with associated costs and a default cost for missing tuples (allows for a compact representation)

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

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

Note : The default semantics (using *var* keyword) of monolithic (flow and DAG-based propagators) global cost functions is to count the number of variables to change in order to restore consistency and to multiply it by the basecost. Other particular semantics may be used in conjunction with the flow-based propagator

Note : The semantics of the network-based propagator approach is either a hard constraint ("hard" keyword) or a soft constraint by multiplying the number of changes by the basecost ("lin" or "var" keyword) or by multiplying the square value of the number of changes by the basecost ("quad" keyword)

Note : A decomposable version exists for each monolithic global cost function, except grammar and MST. The decomposable ones may propagate less than their monolithic counterpart and they introduce extra variables but they can be much faster in practice

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

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

Warning : Current implementation of toulbar2 disallows global cost functions with less than or equal to three variables in their scope (use cost functions in extension instead)

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

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

12.3.2 Solving cost function networks

group **solving**

After creating a Weighted CSP, it can be solved using a local search method like INCOP or PILS (see *WeightedCSPSolver::narycsp* or *WeightedCSPSolver::pils*) 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;
    // ToulBar2::varOrder = reinterpret_cast<char*>(-3);

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

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

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

toulbar2test.cpp

toulbar2test.cpp

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

#include "toulbar2lib.hpp"

#include "core/tb2wcsp.hpp"
#include "vns/tb2vnsutils.hpp"
#include "vns/tb2dgvns.hpp"
#ifdef OPENMPI
#include "vns/tb2cpdgvns.hpp"
#include "vns/tb2rpdgvns.hpp"
#endif
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

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

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

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

    // uncomment if Virtual Arc Consistency (equivalent to Augmented DAG␣
↪algorithm) enable
    //        ToulBar2::vac = 1; // option -A
    //        ToulBar2::vacValueHeuristic = true; // option -V
    // uncomment if partial Limited Discrepancy Search enable
    //        ToulBar2::lds = 1;  // option -l=1
```

(continues on next page)

(continued from previous page)

```
    // uncomment if INCOP local search enable
    //        ToulBar2::incop_cmd = Incop_cmd; // option -i
    // uncomment the following lines if variable neighborhood search enable
    //ToulBar2::lds = 4;
    //ToulBar2::restart = 10000;
    //#ifdef OPENMPI
    //     if (world.size() > 1) {
    //             ToulBar2::searchMethod = RPDGVNS;
    //             ToulBar2::vnsParallel = true;
    //             ToulBar2::vnsNeighborVarHeur = MASTERCLUSTERRAND;
    //             ToulBar2::vnsParallelSync = false;
    //     } else {
    //             ToulBar2::searchMethod = DGVNS;
    //             ToulBar2::vnsNeighborVarHeur = CLUSTERRAND;
    //     }
    //#else
    //        ToulBar2::searchMethod = DGVNS;
    //        ToulBar2::vnsNeighborVarHeur = CLUSTERRAND;
    //**or**
    //  ToulBar2::searchMethod = VNS;
    //  ToulBar2::vnsNeighborVarHeur = RANDOMVAR;
    //#endif

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

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

    // add binary cost functions (Ising) on each pair of variables
    {
        vector<Cost> costs;
```

(continues on next page)

(continued from previous page)

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

    // add a ternary hard constraint (x+y=z)
    {
        vector<Cost> costs;
        for (unsigned int i = 0; i < 2; i++) {
            for (unsigned int j = 0; j < 2; j++) {
                for (unsigned int k = 0; k < 2; k++) {
                    costs.push_back((solver->getWCSP()->toValue(x, i) + solver-
↪>getWCSP()->toValue(y, j) == solver->getWCSP()->toValue(z, k)) ? 0 : MAX_
↪COST);
                }
            }
        }
        solver->getWCSP()->postTernaryConstraint(x, y, z, costs);
    }

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

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

    //tb2checkOptions();
    if (solver->solve()) {
#ifdef OPENMPI
        if (world.rank() == WeightedCSPSolver::MASTER) {
#endif
            // show optimal solution
            vector<Value> sol;
            Cost optimum = solver->getSolution(sol);
            cout << "Optimum=" << optimum << endl;
            cout << "Solution: x=" << sol[x] << " ,y=" << sol[y] << " ,z=" <<␣
↪sol[z] << endl;
#ifdef OPENMPI
        }
#endif
    } else {
#ifdef OPENMPI
        if (world.rank() == WeightedCSPSolver::MASTER) {
```

(continues on next page)

(continued from previous page)

```
#endif
            cout << "No solution found!" << endl;
#ifdef OPENMPI
        }
#endif
    }
    // cout << "Problem lower bound: " << solver->getWCSP()->getLb() << endl; /
↪/ initial problem lower bound possibly enhanced by value removals at the␣
↪root during search

    delete solver;
    return 0;
}

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

12.3.3 Output messages, verbosity options and debugging

group **verbosity**

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

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

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

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

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

Note : `toulbar2`, compiled in debug mode, can be more verbose and it checks a lot of assertions (pre/post conditions in the code)

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

12.3.4 Preprocessing techniques

group **`preprocessing`**

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

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

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

12.3.5 Variable and value search ordering heuristics

group **`heuristics`**

See : *Boosting Systematic Search by Weighting Constraints* . Frederic Boussemart, Fred Hemery, Christophe Lecoutre, Lakhdar Sais. Proc. of ECAI 2004, pages 146-150. Valencia, Spain, 2004.

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

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

12.3.6 Soft arc consistency and problem reformulation

group **softac**

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

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

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

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

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

12.3.7 Virtual Arc Consistency enforcing

group **VAC**

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

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

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

12.3.8 NC bucket sort

group **ncbucket**

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

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

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

12.3.9 Variable elimination

group **varelim**

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

Note : Variable elimination order used in preprocessing is either lexicographic or given by an external file *.order (see toulbar2 options)

Note : 2-bounded variable elimination during search is optimal in the sense that any elimination order should result in the same final graph

Warning : It is not possible to display/save solutions when bounded variable elimination is applied in preprocessing

Warning : toulbar2 maintains a list of current cost functions for each variable. It uses the size of these lists as an approximation of variable degrees. During the search, if variable *x* has three cost functions *xy*, *xz*, *xyz*, its true degree is two but its approximate degree is three. In toulbar2 options, it is the approximate degree which is given by the user for variable elimination during the search (thus, a value at most three). But it is the true degree which is given by the user for variable elimination in preprocessing.

12.3.10 Propagation loop

group **propagation**

Propagates soft local consistencies and bounded variable elimination until all the propagation queues are empty or a contradiction occurs.

While (queues are not empty or current objective bounds have changed):

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

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

12.3.11 Backtrack management

group **backtrack**

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

Trailing stacks are associated to each storable type:

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

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

Note : storable data are not trailed at depth 0.

Warning : Current storable data management is not multi-threading safe! (Store is a static virtual class relying on StoreBasic<T> static members)

12.4 Libraries

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

CHAPTER

THIRTEEN

DOCUMENTATION IN PDF

- Main documentation :

  `toulbar2`

- API Reference :

  `Class Diagram|C++ Library of toulbar2|Python Library of toulbar2`

- Some extracts :

  `User manual|Reference manual`

  `WCSP format|CFN format`

  `Tutorials|Use cases`

CHAPTER

FOURTEEN

PUBLICATIONS

14.1 Conference talks

- talk on toulbar2 at ROADEF 2023, Rennes, France, February 21, 2023.
- ANITI webinar on toulbar2 for industrial applications : slides in English | talk in French
- 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.

14.2 Related publications

14.2.1 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.

- **Partition crossover iterative local search (pils)**

  Iterated local search with partition crossover for computational protein design, François Beuvin, Simon de Givry, Thomas Schiex, Sébastien Verel, and David Simoncini, Proteins: Structure, Function, and Bioinformatics, 2021.

- **Knapsack/generalized linear global constraint (knapsack/knapsackp)**

Multiple-choice knapsack constraint in graphical models, P Montalbano, S de Givry, and G Katsirelos, In Proc. of CP-AI-OR'2022, Los Angeles, CA, 2022.

- **Parallel hybrid best-first search (parallel HBFS)**

  Parallel Hybrid Best-First Search, A Beldjilali, P Montalbano, D Allouche, G Katsirelos, and S de Givry, In Proc. of CP-22, volume 235, pages 7:1-7:10, Haifa, Israel, 2022.

- **Virtual Pairwise Consistency (pwc, hve)**

  Virtual Pairwise Consistency in Cost Function Networks, P Montalbano, D Allouche, S de Givry, G Katsirelos, and Tomas Werner In Proc. of CP-AI-OR'2023, Nice, France, 2023.

14.2.2 toulbar2 for Combinatorial Optimization in Life Sciences

- **Computational Protein Design**

  Colom, Mireia Solà, et al. Deep Evolutionary Forecasting identifies highly-mutated SARS-CoV-2 variants via functional sequence-landscape enumeration. Research Square, 2022.

  XENet: Using a new graph convolution to accelerate the timeline for protein design on quantum computers Jack B. Maguire, Daniele Grattarola, Vikram Khipple Mulligan, Eugene Klyshko, Hans Melo. Plos Comp. Biology, 2021.

  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.

14.2.3 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**

  Dubray, A., Derval, G., Nijssen, S., Schaus, P. Optimal Decoding of Hidden Markov Models with Consistency Constraints. In Proc. of Discovery Science (DS), LNCS 13601, 2022.

  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.

BIBLIOGRAPHY

[Beldjilali22] A Beldjilali, P Montalbano, D Allouche, G Katsirelos and S de Givry. Parallel Hybrid Best-First Search. In *Proc. of CP-22*, Haifa, Israel, 2022.

[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. In *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.