

---

# **toulbar2 Documentation**

***Release 1.0.0***

**INRAE**

**Sep 26, 2025**

# CONTENTS

<b>1</b>	<b>Authors</b>	<b>2</b>
<b>2</b>	<b>Citations</b>	<b>3</b>
<b>3</b>	<b>Acknowledgments</b>	<b>4</b>
<b>4</b>	<b>License</b>	<b>5</b>
<b>5</b>	<b>Downloads</b>	<b>6</b>
5.1	Open-source code . . . . .	6
5.2	Packages . . . . .	6
5.3	Binaries . . . . .	6
5.4	Python package . . . . .	6
5.5	Docker images . . . . .	6
<b>6</b>	<b>Benchmark libraries</b>	<b>7</b>
<b>7</b>	<b>Tutorials</b>	<b>8</b>
7.1	Step-by-step tutorials . . . . .	8
7.2	Examples . . . . .	8
<b>8</b>	<b>Use cases</b>	<b>10</b>
<b>9</b>	<b>List of all examples</b>	<b>11</b>
9.1	Sudoku puzzle in Pytoulbar2 . . . . .	11
9.1.1	Getting started . . . . .	11
9.1.2	Representing the grid in ToulBar2 . . . . .	12
9.1.3	Solving first the grid . . . . .	12
9.1.4	Adding initial values . . . . .	13
9.1.5	Adding constraints and solving the grid . . . . .	14
9.1.6	Conclusion . . . . .	15
9.2	Sudoku puzzle with libtb2 in C++ . . . . .	17
9.2.1	Getting started . . . . .	17
9.2.2	Representing the grid in ToulBar2 . . . . .	18
9.2.3	Solving first the grid . . . . .	18
9.2.4	Assignment to the input values . . . . .	20
9.2.5	Adding constraints and solving the grid . . . . .	21
9.2.6	Conclusion . . . . .	23
9.3	Weighted n-queen problem . . . . .	26
9.3.1	Brief description . . . . .	26
9.3.2	CFN model . . . . .	26

9.3.3	Example for N=4 in JSON .cfn format . . . . .	26
9.3.4	Python model . . . . .	27
9.4	Weighted latin square problem . . . . .	29
9.4.1	Brief description . . . . .	29
9.4.2	CFN model . . . . .	29
9.4.3	Example for N=4 in JSON .cfn format . . . . .	29
9.4.4	Python model . . . . .	30
9.4.5	C++ model . . . . .	31
9.5	Bicriteria weighted latin square problem . . . . .	34
9.5.1	Brief description . . . . .	34
9.5.2	CFN model . . . . .	34
9.5.3	Python model . . . . .	35
9.5.4	C++ model . . . . .	38
9.6	Radio link frequency assignment problem . . . . .	42
9.6.1	Brief description . . . . .	42
9.6.2	CFN model . . . . .	42
9.6.3	Data . . . . .	42
9.6.4	Python model . . . . .	42
9.7	Frequency assignment problem with polarization . . . . .	44
9.7.1	Brief description . . . . .	44
9.7.2	CFN model . . . . .	45
9.7.3	Data . . . . .	45
9.7.4	Python model . . . . .	46
9.8	Mendelian error detection problem . . . . .	49
9.8.1	Brief description . . . . .	49
9.8.2	CFN model . . . . .	49
9.8.3	Data . . . . .	49
9.8.4	Python model . . . . .	49
9.9	Block modeling problem . . . . .	51
9.9.1	Brief description . . . . .	51
9.9.2	CFN model . . . . .	52
9.9.3	Data . . . . .	52
9.9.4	Python model . . . . .	53
9.10	Airplane landing problem . . . . .	55
9.10.1	Brief description . . . . .	55
9.10.2	CFN model . . . . .	55
9.10.3	Data . . . . .	56
9.10.4	Python model solver . . . . .	56
9.11	Warehouse location problem . . . . .	57
9.11.1	Brief description . . . . .	57
9.11.2	CFN model . . . . .	57
9.11.3	Data . . . . .	57
9.11.4	Python model solver . . . . .	58
9.12	Square packing problem . . . . .	59
9.12.1	Brief description . . . . .	59
9.12.2	CFN model . . . . .	60
9.12.3	Python model . . . . .	60
9.12.4	C++ program using libtb2.so . . . . .	62
9.13	Square soft packing problem . . . . .	64
9.13.1	Brief description . . . . .	64
9.13.2	CFN model . . . . .	64
9.13.3	Python model . . . . .	64
9.13.4	C++ program using libtb2.so . . . . .	66
9.14	Golomb ruler problem . . . . .	67

9.14.1	Brief description	67
9.14.2	CFN model	68
9.14.3	Python model	68
9.15	Board coloration problem	69
9.15.1	Brief description	69
9.15.2	CFN basic model	70
9.15.3	Python model	70
9.16	Learning to play the Sudoku	72
9.16.1	Available	72
9.17	Learning car configuration preferences	72
9.17.1	Brief description	72
9.17.2	Available	72
9.18	Visual Sudoku Tutorial	72
9.18.1	Brief description	72
9.18.2	Available	72
9.19	Visual Sudoku Application	73
9.19.1	Brief description	73
9.19.2	Available	73
9.20	Visual Sudoku App for Android	74
9.20.1	A visual sudoku solver based on cost function networks	74
9.20.2	Source Code	74
9.20.3	Download and Install	74
9.20.4	Description	75
9.21	A sudoku code	76
9.21.1	Brief description	76
9.21.2	Available	76
<b>10</b>	<b>User Guide</b>	<b>79</b>
10.1	What is toulbar2	79
10.2	How do I install it ?	80
10.3	How do I test it ?	80
10.4	Using it as a black box	81
10.5	Quick start	81
10.6	Potential issues	102
10.7	Command line options	102
10.7.1	General control	103
10.7.2	Preprocessing	103
10.7.3	Initial upper bounding	104
10.7.4	Tree search algorithms and tree decomposition selection	105
10.7.5	Variable neighborhood search algorithms	107
10.7.6	Node processing & bounding options	107
10.7.7	Branching, variable and value ordering	108
10.7.8	Diverse solutions	109
10.7.9	Console output	109
10.7.10	File output	110
10.7.11	Probability representation and numerical control	110
10.7.12	Random problem generation	110
10.8	Input formats	111
10.8.1	Introduction	111
10.8.2	Formats details	112
10.9	How do I use it ?	138
10.9.1	Using it as a C++ library	138
10.9.2	Using it from Python	138
10.10	References	139

<b>11</b>	<b>Reference Manual</b>	<b>140</b>
11.1	Introduction . . . . .	140
11.2	Exact optimization for cost function networks and additive graphical models . . . . .	140
11.2.1	What is toulbar2? . . . . .	140
11.2.2	Installation from binaries . . . . .	141
11.2.3	Python interface . . . . .	141
11.2.4	Download . . . . .	141
11.2.5	Installation from sources . . . . .	141
11.3	Modules . . . . .	143
11.3.1	Variable and cost function modeling . . . . .	143
11.3.2	Solving cost function networks . . . . .	144
11.3.3	Output messages, verbosity options and debugging . . . . .	149
11.3.4	Preprocessing techniques . . . . .	150
11.3.5	Variable and value search ordering heuristics . . . . .	150
11.3.6	Soft arc consistency and problem reformulation . . . . .	150
11.3.7	Virtual Arc Consistency enforcing . . . . .	151
11.3.8	NC bucket sort . . . . .	151
11.3.9	Variable elimination . . . . .	151
11.3.10	Propagation loop . . . . .	152
11.3.11	Backtrack management . . . . .	152
11.4	Libraries . . . . .	153
<b>12</b>	<b>Documentation in pdf</b>	<b>154</b>
<b>13</b>	<b>Publications</b>	<b>155</b>
13.1	Conference talks . . . . .	155
13.2	Related publications . . . . .	155
13.2.1	What are the algorithms inside toulbar2 ? . . . . .	155
13.2.2	toulbar2 for Combinatorial Optimization in Life Sciences . . . . .	157
13.2.3	Other publications mentioning toulbar2 . . . . .	158
	<b>Bibliography</b>	<b>161</b>

**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  $\infty$  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 performance. See [Publications](#).

toulbar2 won several medals in competitions on Max-CSP/COP ([CPAI08](#), [XCSP3 2022](#), [2023](#), and [2024](#)) 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](#)).

**AUTHORS**

**toulbar2** was originally developed by Toulouse (INRAE MIAT) and Barcelona (UPC, IIIA-CSIC) teams, hence the solver's name.

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 XCSP 2.1 and XCSP 3 format readers of CSP and COP instances
- Université de Toulouse I Capitole (IRIT) and Université du Littoral Côte d'Opale, France (LISIC) for PILS local search solver

## 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



## ACKNOWLEDGMENTS

**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 [Ficolof](#) from 2011 to 2014, and ANR-16-CE40-0028 [DemoGraph](#) from 2017 to 2021, PIA3 ANITI ANR-19-P3IA-0004 from 2019 to 2024) and a PHC PROCORE project number 28680VH (from 2013 to 2015). It is currently supported by ANITI2 (2024-2031) and ANR project GMLaAS (2025-2029).

**LICENSE****MIT License**

Copyright (c) 2025 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.

## DOWNLOADS

### 5.1 Open-source code

- [toulbar2 on GitHub](#) 

### 5.2 Packages

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

```
apt install toulbar2
```

### 5.3 Binaries

- **Latest release toulbar2 binaries**

[Linux 64bit](#) | [MacOs 64bit](#) | [Windows 64bit](#)

### 5.4 Python package

- **pytoulbar2** module for Linux and MacOS on [PyPI](#)

### 5.5 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
```

## BENCHMARK LIBRARIES

- [EvalGM](#) : 3026 discrete optimization benchmarks available in various formats (wesp, 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 wesp, wcnf, lp, qpbo, uai, opb, and more).

## 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.

### 7.1 Step-by-step tutorials


- *Sudoku puzzle in Pytoulbar2*
- *Sudoku puzzle with libtb2 in C++*

### 7.2 Examples

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*

## 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** : PPositive Multistate Protein Design, [Vucini et al Bioinformatics 2020]

- *Visual Sudoku Application* 

### Misc :

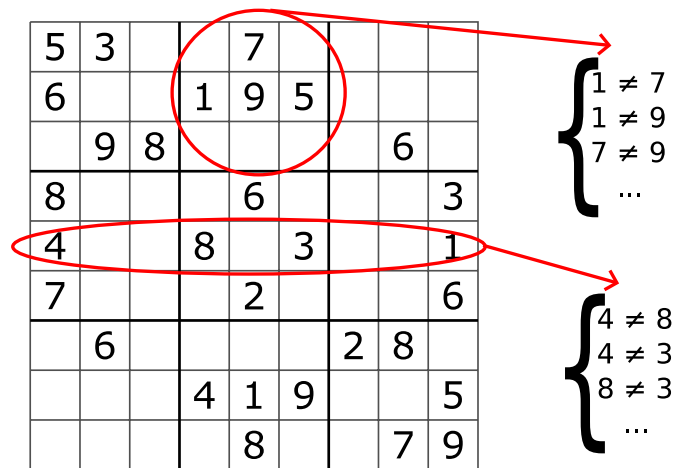
- *A sudoku code*

## LIST OF ALL EXAMPLES

### 9.1 Sudoku puzzle in Pytoulbar2

The Sudoku is a widely known puzzle game that consists in filling a grid with numbers. The typical grid has 9x9 cells in total and each cell must contain an integer between 1 and 9.

Additional constraints need to be enforced to solve the puzzle. In each line, the same number cannot appear twice. The same type of constraint occurs for each column. Finally, each sub square of size 3x3 located every 3 cells must also not contain duplicates. The image bellow shows an example of a Sudoku grid given with its initial values. The goal is to deduce the other values while verifying the different constraints.



#### 9.1.1 Getting started

Before starting, make sure pytoulbar2 is installed in your environment. It can be installed via the command:

```
pip install pytoulbar2
```

We first create a [CFN](#) object. CFN, which stands for Cost Function Network, is the main object that is manipulated in pytoulbar2. It represents a problem to solve expressed as a network of cost functions, i.e a set of variables that are connected to each other through discrete cost functions (or constraints).

As ToulBar2 is an optimization framework, an optional upper bound can be provided to the CFN object in order to exclude any solution whose value exceeds this bound. In the case of a Sudoku puzzle, since the problem does not contain a numerical objective, an upper bound of 1 can be chosen.



```
import pytoulbar2

top = 1 # upper bound value of the problem
cfn = pytb2.CFN(top)
```

### 9.1.2 Representing the grid in ToulBar2

To represent our problem in pytoulbar2, it is necessary to define discrete decision variables. The variables will represent the various choices that can be made to build a solution to the problem. In the Sudoku puzzle, decision variables are typically the different cells of the grid. Their values would be the possible integers they can be assigned to, from 1 to 9. We use the function `AddVariable` of our `cfn` object to create the variables.

```
# variable creation
for row in range(9):
    for col in range(9):
        cfn.AddVariable('cell_'+str(row)+'_'+str(col), range(9))
```

### 9.1.3 Solving first the grid

It is already possible to “solve” the puzzle with ToulBar2, as the `cfn` object contains the variables of the problem. The function `Solve` is used to run the solving algorithm.

```
result = cfn.Solve(showSolutions = 3)
```

If ToulBar2 finds a solution, it will return an array containing the chosen values for each variable:

```
print(result[0][0])
```

Would return 0, meaning the chosen value of the first variable (upper left cell) is 1 (first value of the list). We then define a function to print the solutions as a grid:

```
# print a solution as a Sudoku grid
def print_grid(solution):

    print('-----')

    var_index = 0

    for row in range(9):
        line = ''
        for col in range(9):
            if col % 3 == 0:
                line += '|'
            line += str(solution[var_index]+1)
            if col % 3 == 2:
                line += ' '
            var_index += 1
        line += '|'
        print(line)
        if row % 3 == 2:
            print('-----')
```

(continues on next page)

(continued from previous page)

```
# print the first solution
print_grid(result)
```

Which helps to display the first solution:

```
-----
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
-----
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
-----
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
-----
```

### 9.1.4 Adding initial values

The next step consists in initializing the variables that correspond to cells for which the value is known. We will use the values in the grid example above, defined as a double array (where 0 means the value is unspecified) :

```
# define known values
initial_values = [[5,3,0,0,7,0,0,0,0],
                  [6,0,0,1,9,5,0,0,0],
                  [0,9,8,0,0,0,0,6,0],
                  [8,0,0,0,6,0,0,0,3],
                  [4,0,0,8,0,3,0,0,1],
                  [7,0,0,0,2,0,0,0,6],
                  [0,6,0,0,0,0,2,8,0],
                  [0,0,0,4,1,9,0,0,5],
                  [0,0,0,0,8,0,0,7,9]]
```

Variables can be assigned with the function `Assign`. The variable and its value can be specified as integer indexes or as strings.

```
var_index = 0
for row in range(9):
    for col in range(9):
        if initial_values[row][col] != 0:
            cfn.Assign(var_index, initial_values[row][col]-1)
            var_index += 1

result = cfn.Solve(showSolutions = 3)
print_grid(result[0])
```

**⚠ Caution**

Although we have already solved the problem once, a CFN object cannot execute its `Solve` function twice in a row. The object must be recreated or the function must be called only once.

The solution returned by the algorithm this time looks like this :

```

-----
| 5 3 1 | 1 7 1 | 1 1 1 |
| 6 1 1 | 1 9 5 | 1 1 1 |
| 1 9 8 | 1 1 1 | 1 6 1 |
-----
| 8 1 1 | 1 6 1 | 1 1 3 |
| 4 1 1 | 8 1 3 | 1 1 1 |
| 7 1 1 | 1 2 1 | 1 1 6 |
-----
| 1 6 1 | 1 1 1 | 2 8 1 |
| 1 1 1 | 4 1 9 | 1 1 5 |
| 1 1 1 | 1 8 1 | 1 7 9 |
-----

```

The initial values are now correctly specified in the variables.

### 9.1.5 Adding constraints and solving the grid

The last missing part before being able to compute a solution is the constraints. Starting with the row constraints, we must ensure that none of the variables in the same row will be assigned to the same values. This constraint is usually called *all different* and can be added with the function `AddAllDifferent`. The function takes as an argument a list of indices of the variables that must differ. The constraint on the first row is obtained via:

```
cfn.AddAllDifferent([var_ind for var_ind in range(9)])
```

Which generates the following first row in the solution:

```

-----
| 5 3 1 | 2 7 4 | 9 8 6 |

```

Constraints for each row can be added by varying the column index for each row:

```
# row constraints
for row_ind in range(9):
    cfn.AddAllDifferent([row_ind*9+col_ind for col_ind in range(9)])
```

Constraints for each column are obtained similarly:

```
# column constraints
for col_ind in range(9):
    cfn.AddAllDifferent([row_ind*9+col_ind for row_ind in range(9)])
```

At this point, the solution is not correct yet since sub-grids of size 3x3 may contain duplicates, such as the values 9 and 3 in the example below:

```

-----
| 5 3 9 |

```

(continues on next page)

(continued from previous page)

```
| 6 2 3 |
| 1 9 8 |
-----
```

Additional constraints are added for each of the 9 sub-grids:

```
# sub grids constraints
for sub_ind1 in range(3): # row offset
    for sub_ind2 in range(3): # column offset
        cfn.AddAllDifferent([(sub_ind1*3+row_ind)*9+ sub_ind2*3+col_ind for col_ind in
↪range(3) for row_ind in range(3)])
```

These last constraints allow to obtain a consistent solution to the Sudoku puzzle finally:

```
-----
| 5 3 4 | 6 7 8 | 9 1 2 |
| 6 7 2 | 1 9 5 | 3 4 8 |
| 1 9 8 | 3 4 2 | 5 6 7 |
-----
| 8 5 9 | 7 6 1 | 4 2 3 |
| 4 2 6 | 8 5 3 | 7 9 1 |
| 7 1 3 | 9 2 4 | 8 5 6 |
-----
| 9 6 1 | 5 3 7 | 2 8 4 |
| 2 8 7 | 4 1 9 | 6 3 5 |
| 3 4 5 | 2 8 6 | 1 7 9 |
-----
```

## 9.1.6 Conclusion

This short introduction shows how to represent a problem in ToulBar2 via its python interface *Pytoulbar2* by defining the problem variables and constraints and how to obtain a solution to this problem. Below is the complete python script from this tutorial.

sudoku\_tutorial.py

```
import pytoulbar2 as pytb2

# print a solution as a sudoku grid
def print_grid(solution):

    print('-----')

    var_index = 0

    for row in range(9):
        line = ''
        for col in range(9):
            if col % 3 == 0:
                line += '|'
            line += ' '
            line += str(solution[var_index]+1)
            if col % 3 == 2:
```

(continues on next page)

(continued from previous page)

```

        line += ' '
        var_index += 1
    line += '|'
    print(line)
    if row % 3 == 2:
        print('-----')

cfn = pytb2.CFN()

# variables
for row in range(9):
    for col in range(9):
        cfn.AddVariable('cell_'+str(row)+'_'+str(col), range(9))

# define known values
initial_values = [[5,3,0,0,7,0,0,0,0],
                  [6,0,0,1,9,5,0,0,0],
                  [0,9,8,0,0,0,0,6,0],
                  [8,0,0,0,6,0,0,0,3],
                  [4,0,0,8,0,3,0,0,1],
                  [7,0,0,0,2,0,0,0,6],
                  [0,6,0,0,0,0,2,8,0],
                  [0,0,0,4,1,9,0,0,5],
                  [0,0,0,0,8,0,0,7,9]]

var_index = 0
for row in range(9):
    for col in range(9):
        if initial_values[row][col] != 0:
            cfn.Assign(var_index, initial_values[row][col]-1)
            var_index += 1

# row constraints
for row_ind in range(9):
    cfn.AddAllDifferent([row_ind*9+col_ind for col_ind in range(9)])

# column constraints
for col_ind in range(9):
    cfn.AddAllDifferent([row_ind*9+col_ind for row_ind in range(9)])

# sub grids constraints
for sub_ind1 in range(3): # row offset
    for sub_ind2 in range(3): # column offset
        cfn.AddAllDifferent([(sub_ind1*3+row_ind)*9+ sub_ind2*3+col_ind for col_ind in
↪range(3) for row_ind in range(3)])

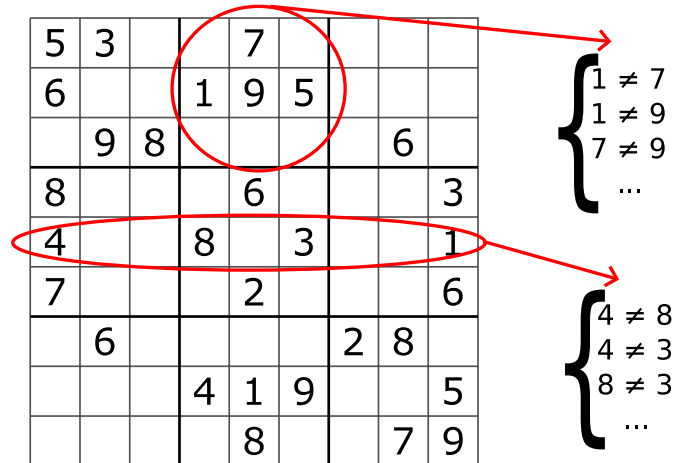
result = cfn.Solve(showSolutions = 3, allSolutions=1)
print_grid(result[0])

```

## 9.2 Sudoku puzzle with libtb2 in C++

The Sudoku is a widely known puzzle game that consists in filling a grid with numbers. The typical grid has 9x9 cells in total and each cell must contain an integer between 1 and 9.

Additional constraints need to be enforced to solve the puzzle. In each line, the same number cannot appear twice. The same type of constraint occurs for each column. Finally, each sub-square of size 3x3 located every 3 cells must also not contain duplicates. The image below shows an example of a Sudoku grid given with its initial values. The goal is to deduce the other values while verifying the different constraints.



### 9.2.1 Getting started

Before starting, make sure the ToulBar2 C++ library binaries are installed in your system (libtb2.so, see installation section from [sources](#) or [binaries](#) for more instructions).

We first create a `WeightedCSPSolver` object. This object is in charge of executing the algorithm to solve the Sudoku grid and internally creates a `WeightedCSP` object to store the problem.

`WeightedCSP` objects are used in ToulBar2 to define the optimization or decision problems. The problem is expressed as a set of discrete variables that are connected to each other through cost functions (or constraints).

As ToulBar2 is an optimization framework, an optional upper bound can be provided to the solver object in order to exclude any solution whose value exceeds this bound. In the case of a Sudoku puzzle, since the problem does not contain a numerical objective, an upper bound of 1 can be chosen.

In order to compile the following code, assuming we are in the main ToulBar2 source repository, the same compilation flags as used to compile libtb2.so must be used: `g++ -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -I./src -o sudoku sudoku_tutorial.cpp libtb2.so`

```
#include <iostream>
#include <toulbar2lib.hpp>

using namespace std;

int main() {

    // initialization
    tb2init();
```

(continues on next page)

(continued from previous page)

```

initCosts();

Cost top = 1;

// creation of the solver object
WeightedCSPSolver* solver = WeightedCSPSolver::makeWeightedCSPSolver(top);

// access to the wcsp object created by the solver
WeightedCSP* wcsp = solver->getWCSP();

delete solver;

return 0;
}

```

## 9.2.2 Representing the grid in ToulBar2

To represent our problem in pytoulbar2, it is necessary to define discrete decision variables. The variables will represent the various choices that can be made to build a solution to the problem. In the Sudoku puzzle, decision variables are typically the different cells of the grid. Their values would be the possible integers they can be assigned to, from 1 to 9. We use the `makeEnumeratedVariable` function of the `wcsp` object to make the variables.

```

// variable creation
for(size_t row = 0; row < 9; row++) {
    for(size_t col = 0; col < 9; col++) {
        wcsp->makeEnumeratedVariable("Cell_" + to_string(row) + "," + to_string(col), 0, 8);
    }
}

```

## 9.2.3 Solving first the grid

It is already possible to solve the puzzle with ToulBar2, as the `cfn` object contains the variables of the problem. The `WeightedCSPSolver::solve` function is used to run the solving algorithm.

```

// close the model definition
wcsp->sortConstraints();

// solve the problem
bool hasSolution = solver->solve();

```

### Warning

It is important to systematically call the function `WeightedCSP::sortConstraints` before solving the problem to close the model definition.

Problems may sometimes be especially hard to solve. In such cases, a timeout can be set on the solver to stop the search before the optimality proof is complete, or before a solution is found at all. When doing so, ToulBar2 throws an exception that must be caught to properly clean the problem data structures:

```
try {
    bool hasSolution = solver->solve();
} catch(const exception& ex) {
    cout << "no solution found: " << ex.what() << endl;
}
```

When ToulBar2 returns a solution, the solution can be accessed as a `std::vector` of `Value`, specifying the value that is assigned to each variable of the problem (in the same order they were defined):

```
if(hasSolution) {
    std::vector<Value> solution = solver->getSolution();
    cout << "the first value is " << solution[0] << endl;
}
```

The above code would display `0`, meaning the chosen value of the first variable (upper left cell) is 1 (first value of the list). We then define a function to print the solution as a grid :

```
// print a solution as a grid
void printSolution(const std::vector<Value>& solution) {

    cout << "-----" << endl;

    size_t var_index = 0;

    for(size_t row = 0; row < 9; row ++) {

        for(size_t col = 0; col < 9; col ++) {

            if(col % 3 == 0) {
                cout << "|";
            }
            cout << " " << to_string(solution[var_index]+1);
            if(col % 3 == 2) {
                cout << " ";
            }

            var_index += 1;
        }
        cout << "|" << endl;

        if(row % 3 == 2) {
            cout << "-----" << endl;
        }

    }

}
```

This function helps to visualize the variables' values as a real Sudoku grid, as follows :

```
// print the first solution
if(hasSolution) {
    std::vector<Value> solution = solver->getSolution();
```

(continues on next page)



(continued from previous page)

```
    printSolution(solution)
}
```

Which helps to display the first solution:

```
-----
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
-----
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
-----
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | 1 1 1 |
-----
```

## 9.2.4 Assignment to the input values

The next step consists in initializing the variables that correspond to cells for which the value is known. We will use the values in the grid example above, defined as a two-dimensional vector (where 0 means the value is unspecified):

```
// grid data
std::vector<std::vector<Value> > input_grid {
    {5, 3, 0, 0, 7, 0, 0, 0, 0},
    {6, 0, 0, 1, 9, 5, 0, 0, 0},
    {0, 9, 8, 0, 0, 0, 0, 6, 0},
    {8, 0, 0, 0, 6, 0, 0, 0, 3},
    {4, 0, 0, 8, 0, 3, 0, 0, 1},
    {7, 0, 0, 0, 2, 0, 0, 0, 6},
    {0, 6, 0, 0, 0, 0, 2, 8, 0},
    {0, 0, 0, 4, 1, 9, 0, 0, 5},
    {0, 0, 0, 0, 8, 0, 0, 7, 9} };
```

Variables can be assigned with the `WeightedCSP::assign` function. The variable and its value can be specified as integer indexes or as strings.

```
// input values initialization
size_t var_ind = 0;
for(size_t row = 0; row < 9; row++) {
    for(size_t col = 0; col < 9; col++) {

        if(input_grid[row][col] != 0) {
            wcsp->assign(var_ind, input_grid[row][col]-1);
        }

        var_ind++;
    }
}
```

**Warning**

Although we already solved the problem once, a `WeightedCSP` object cannot execute its `WeightedCSPSolver::solve` function twice in a row. The object must be recreated or the function must be called only once.

The solution returned by the algorithm this time looks like this:

```
-----
| 5 3 1 | 1 7 1 | 1 1 1 |
| 6 1 1 | 1 9 5 | 1 1 1 |
| 1 9 8 | 1 1 1 | 1 6 1 |
-----
| 8 1 1 | 1 6 1 | 1 1 3 |
| 4 1 1 | 8 1 3 | 1 1 1 |
| 7 1 1 | 1 2 1 | 1 1 6 |
-----
| 1 6 1 | 1 1 1 | 2 8 1 |
| 1 1 1 | 4 1 9 | 1 1 5 |
| 1 1 1 | 1 8 1 | 1 7 9 |
-----
```

The initial values are now correctly specified in the variables.

### 9.2.5 Adding constraints and solving the grid

The missing part to be able to generate a solution is the constraints. Starting with the row constraints, we must ensure that none of the variables in the same row will be assigned to the same values. This constraint is usually called *all different* and can be added with the `WeightedCSPSolver::postWAllDiff` function. The function takes as arguments a list of indices of the variables that must differ (the scope of the constraint) as well as two parameters specifying how the constraint is encoded, which we do not further describe in this tutorial. We start by adding a constraint for the first row:

```
std::vector<int> scope = {0,1,2,3,4,5,6,7,8};
std::string semantics = "hard";
std::string prop = "knapsack";

wcsp->postWAllDiff(scope, semantics, prop, top);
```

Which generates the following first row in the solution :

```
-----
| 5 3 1 | 2 7 4 | 6 8 9 |
-----
```

Constraints for each row can be added by varying the column index for each row :

```
// add one "all different" constraint for each row
for(int row = 0; row < 9; row++) {
    std::vector<int> row_scope;
    for(int col = 0; col < 9; col++) {
        row_scope.emplace_back(row*9+col);
    }
}
```

(continues on next page)

(continued from previous page)

```
wcsp->postWallDiff(row_scope, semantics, prop, top);
}
```

Constraints for each column are obtained similarly:

```
// add one "all different" constraint for each column
for(int col = 0; col < 9; col ++) {
    std::vector<int> col_scope;
    for(int row = 0; row < 9; row ++) {
        col_scope.emplace_back(row*9+col);
    }
    wcsp->postWallDiff(col_scope, semantics, prop, top);
}
```

At this point, the solution is not correct yet since sub-grids of size 3x3 may contain duplicates, such as the values 9 and 3 in the example below:

```
-----
| 9 7 6 |
| 1 9 5 |
| 5 4 2 |
-----
```

A set of 9 additional allDifferent constraints can be defined to finalize our Sudoku model definition:

```
// add one "all different" constraint for each 3x3 sub-grid
for(int sub_ind1 = 0; sub_ind1 < 3; sub_ind1 ++) {
    for(int sub_ind2 = 0; sub_ind2 < 3; sub_ind2 ++) {
        std::vector<int> sub_scope;
        for(int row_ind = 0; row_ind < 3; row_ind ++) { // iterate inside the 3x3 sub-grid
            for(int col_ind = 0; col_ind < 3; col_ind ++) {
                sub_scope.emplace_back((sub_ind1*3+row_ind)*9+sub_ind2*3+col_ind);
            }
        }
        wcsp->postWallDiff(sub_scope, semantics, prop, top);
    }
}
```

These last constraints allow to finally obtain a consistent solution to the Sudoku puzzle :

```
-----
| 5 3 4 | 6 7 8 | 9 1 2 |
| 6 7 2 | 1 9 5 | 3 4 8 |
| 1 9 8 | 3 4 2 | 5 6 7 |
-----
| 8 5 9 | 7 6 1 | 4 2 3 |
| 4 2 6 | 8 5 3 | 7 9 1 |
| 7 1 3 | 9 2 4 | 8 5 6 |
-----
| 9 6 1 | 5 3 7 | 2 8 4 |
| 2 8 7 | 4 1 9 | 6 3 5 |
| 3 4 5 | 2 8 6 | 1 7 9 |
-----
```

## 9.2.6 Conclusion

This short introduction shows how to represent a problem in ToulBar2 via its C++ library *libtb2* by defining the problem variables and constraints and how to obtain a solution to this problem. Below is the complete C++ source code from this tutorial.

sudoku\_tutorial.cpp

```
#include <iostream>
#include <toulbar2lib.hpp>

using namespace std;

// print a solution as a grid
void printSolution(const std::vector<Value>& solution) {

    cout << "-----" << endl;

    size_t var_index = 0;

    for(size_t row = 0; row < 9; row++) {
        for(size_t col = 0; col < 9; col++) {

            if(col % 3 == 0) {
                cout << "|";
            }
            cout << " " << to_string(solution[var_index]+1);
            if(col % 3 == 2) {
                cout << " ";
            }
            var_index += 1;

        }

        cout << "|" << endl;
        if(row % 3 == 2) {
            cout << "-----" << endl;
        }

    }
}

int main() {

    // grid data
    std::vector<std::vector<Value>> > input_grid { {5, 3, 0, 0, 7, 0, 0, 0, 0},
                                                    {6, 0, 0, 1, 9, 5, 0, 0, 0},
                                                    {0, 9, 8, 0, 0, 0, 0, 6, 0},
                                                    {8, 0, 0, 0, 6, 0, 0, 0, 3},
                                                    {4, 0, 0, 8, 0, 3, 0, 0, 1},
                                                    {7, 0, 0, 0, 2, 0, 0, 0, 6},
                                                    {0, 6, 0, 0, 0, 0, 2, 8, 0},
                                                    {0, 0, 0, 4, 1, 9, 0, 0, 5},
                                                    {0, 0, 0, 0, 8, 0, 0, 7, 9} };
}
```

(continues on next page)

(continued from previous page)

```

// initialisation
tb2init();
initCosts();

Cost top = 1;

// creation of the solver object
WeightedCSPSolver* solver = WeightedCSPSolver::makeWeightedCSPSolver(top);

// access to the wcsp object created by the solver
WeightedCSP* wcsp = solver->getWCSP();

//-----
// problem variables
//-----

// variable creation
for(size_t row = 0; row < 9; row++) {
    for(size_t col = 0; col < 9; col++) {
        wcsp->makeEnumeratedVariable("Cell_" + to_string(row) + "," + to_string(col), 0,
↪ 8);
    }
}

// input values initialisation
size_t var_ind = 0;
for(size_t row = 0; row < 9; row++) {
    for(size_t col = 0; col < 9; col++) {

        if(input_grid[row][col] != 0) {
            wcsp->assign(var_ind, input_grid[row][col]-1);
        }

        var_ind++;
    }
}

//-----
// constraints definition
//-----

// all different constraint parameters
std::string semantics = "hard";
std::string prop = "knapsack";

// add one "all different" constraint for each row
for(int row = 0; row < 9; row++) {
    std::vector<int> row_scope;
    for(int col = 0; col < 9; col++) {
        row_scope.emplace_back(row*9+col);
    }
}

```

(continues on next page)

(continued from previous page)

```

    wcsp->postWallDiff(row_scope, semantics, prop, top);
}

// add one "all different" constraint for each column
for(int col = 0; col < 9; col ++) {
    std::vector<int> col_scope;
    for(int row = 0; row < 9; row ++) {
        col_scope.emplace_back(row*9+col);
    }
    wcsp->postWallDiff(col_scope, semantics, prop, top);
}

// add one "all different" constraint for each 3x3 sub square
for(int sub_ind1 = 0; sub_ind1 < 3; sub_ind1 ++) {
    for(int sub_ind2 = 0; sub_ind2 < 3; sub_ind2 ++) {
        std::vector<int> sub_scope;
        for(int row_ind = 0; row_ind < 3; row_ind ++) { // iterate inside the 3x3 sub_
↪grid
            for(int col_ind = 0; col_ind < 3; col_ind ++) {
                sub_scope.emplace_back((sub_ind1*3+row_ind)*9+sub_ind2*3+col_ind);
            }
        }
        wcsp->postWallDiff(sub_scope, semantics, prop, top);
    }
}

//-----
// solution
//-----

// close the model definition
wcsp->sortConstraints();

// solve the problem
bool hasSolution = solver->solve();

if(hasSolution) {
    std::vector<Value> solution = solver->getSolution();
    printSolution(solution);
}

// clean the solver object
delete solver;

return 0;
}

```

## 9.3 Weighted n-queen problem

### 9.3.1 Brief description

The problem consists in assigning  $N$  queens on a  $N \times N$  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.

### 9.3.2 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.

### 9.3.3 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)				(Q0,Q2) constraint (2nd clique)				(Q0,Q3) constraint (3rd clique)			
Costs	Q0	Q1		Costs	Q0	Q2		Costs	Q0	Q3	
0	17	Row0	Row0	0	0	Row0	Row0	0	0	Row0	Row0
1	0	Row0	Row1	1	0	Row0	Row1	1	0	Row0	Row1
2	0	Row0	Row2	2	0	Row0	Row2	2	17	Row0	Row2
3	0	Row0	Row3	3	0	Row0	Row3	3	0	Row0	Row3
4	0	Row1	Row0	4	0	Row1	Row0	4	0	Row1	Row0
5	17	Row1	Row1	5	0	Row1	Row1	5	0	Row1	Row1
6	0	Row1	Row2	6	0	Row1	Row2	6	0	Row1	Row2
7	0	Row1	Row3	7	0	Row1	Row3	7	17	Row1	Row3
8	0	Row2	Row0	8	17	Row2	Row0	8	0	Row2	Row0
9	0	Row2	Row1	9	0	Row2	Row1	9	0	Row2	Row1
10	17	Row2	Row2	10	0	Row2	Row2	10	0	Row2	Row2
11	0	Row2	Row3	11	0	Row2	Row3	11	0	Row2	Row3
12	0	Row3	Row0	12	0	Row3	Row0	12	0	Row3	Row0
13	0	Row3	Row1	13	17	Row3	Row1	13	0	Row3	Row1
14	0	Row3	Row2	14	0	Row3	Row2	14	0	Row3	Row2
15	17	Row3	Row3	15	0	Row3	Row3	15	0	Row3	Row3

Q0 Q1 Q2 Q3

Q0 Q1 Q2 Q3

Q0 Q1 Q2 Q3

-- 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, 17, 0, 0, 0, 0, 17, 0, 0, 0]}  
 -- into 3rd clique : {scope: ["Q0", "Q3"], "costs": [0, 0, 0, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 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"],
  ↪    },
  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, 17, 0, 0, 0, 0, 17, 0, 0, 0]},
    {scope: ["Q0", "Q3"], "costs": [0, 0, 0, 0, 0, 0, 0, 17, 0, 0, 0, 0, 17, 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, 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]},
  }
}
```

(continues on next page)

(continued from previous page)





```

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

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

```

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

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

### 9.3.4 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)])

```

(continues on next page)



(continued from previous page)

```

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)

#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]))

```

## 9.4 Weighted latin square problem

### 9.4.1 Brief description

The problem consists in assigning a value from 0 to  $N-1$  to every cell of a  $N \times N$  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.

### 9.4.2 CFN model

We create  $N \times N$  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  $N^2$ , so forbidden assignments have cost  $k=N^2+1$ .

### 9.4.3 Example for $N=4$ in JSON .cfn format

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

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

    {scope: ["X0_0"], "costs": [4, 4, 3, 4]},
    {scope: ["X0_1"], "costs": [4, 3, 4, 4]},
    {scope: ["X0_2"], "costs": [2, 1, 3, 2]},
    {scope: ["X0_3"], "costs": [1, 2, 3, 4]},
    {scope: ["X1_0"], "costs": [3, 1, 3, 3]},
    {scope: ["X1_1"], "costs": [4, 1, 1, 1]},
    {scope: ["X1_2"], "costs": [4, 1, 1, 3]},
    {scope: ["X1_3"], "costs": [4, 4, 1, 4]},
    {scope: ["X2_0"], "costs": [1, 3, 3, 2]},
    {scope: ["X2_1"], "costs": [2, 1, 3, 1]},
```

(continues on next page)

(continued from previous page)

```
{
  {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, <b>4</b> 3	4, 3, <b>4</b> , 4 2	<b>2</b> , 1, 3, 2 0	1, <b>2</b> , 3, 4 1
3, <b>1</b> , 3, 3 1	4, 1, 1, <b>1</b> 3	4, 1, <b>1</b> , 3 2	<b>4</b> , 4, 1, 4 0
<b>1</b> , 3, 3, 2 0	2, <b>1</b> , 3, 1 1	3, 4, 2, <b>2</b> 3	2, 3, <b>1</b> , 3 2
3, 4, <b>4</b> , 2 2	<b>3</b> , 2, 4, 4 0	4, <b>1</b> , 3, 4 1	4, 4, 4, <b>3</b> 3

#### 9.4.4 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)],
```

(continues on next page)

(continued from previous page)

```

↪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]))

```

### 9.4.5 C++ model

The following code using the C++ toulbar2 library API solves the weighted latin square problem.

latinsquare.cpp

```

#include <iostream>
#include <vector>

#include "core/tb2wcsp.hpp"

using namespace std;

// an alias for storing the variable costs
// first dim is the grid rows and second is the columns
typedef std::vector<std::vector<std::vector<Cost>>> LatinCostArray;

/*!
 \brief generate random costs for each variable (cell)
 */
void initLatinCosts(size_t N, LatinCostArray& costs) {

    // N*N*N values, costs for each cell
    costs.resize(N);
    for(auto& col: costs) {
        col.resize(N);
    }
}

```

(continues on next page)

(continued from previous page)

```

    for(auto& cell: col) {
        cell.resize(N);
        for(size_t val_ind = 0; val_ind < N; val_ind += 1) {
            cell[val_ind] = (rand()%N)+1;
        }
    }
}

/*!
 \brief print the costs for each unary variable (cell)
 */
void printCosts(LatinCostArray& costs) {

    for(size_t row_ind = 0; row_ind < costs.size(); row_ind ++) {
        for(size_t col_ind = 0; col_ind < costs[row_ind].size(); col_ind ++) {
            cout << "cell " << row_ind << "_" << col_ind;
            cout << " : ";
            for(auto& cost: costs[row_ind][col_ind]) {
                cout << cost << ", ";
            }
            cout << endl;
        }
    }
}

/*!
 \brief fill in a WCSP object with a latin square problem
 */
void buildWCSP(WeightedCSP& wcsp, LatinCostArray& costs, size_t N, Cost top) {

    // variables
    for(size_t row = 0; row < N; row ++) {
        for(unsigned int col = 0; col < N; col ++) {
            wcsp.makeEnumeratedVariable("Cell_" + to_string(row) + "_" + to_string(col), 0, N-1);
        }
    }

    cout << "number of variables: " << wcsp.numberOfVariables() << endl;

    /* costs for all different constraints (top on diagonal) */
    vector<Cost> alldiff_costs;
    for(unsigned int i = 0; i < N; i ++) {
        for(unsigned int j = 0; j < N; j ++) {
            if(i == j) {
                alldiff_costs.push_back(top);
            } else {
                alldiff_costs.push_back(0);
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    /* all different constraints */
    for(unsigned int index = 0; index < N; index ++) {
        for(unsigned int var_ind1 = 0; var_ind1 < N; var_ind1 ++) {
            for(unsigned int var_ind2 = var_ind1+1; var_ind2 < N; var_ind2 ++) {
                /* row constraints */
                wcsp.postBinaryConstraint(N*index+var_ind1, N*index+var_ind2, alldiff_
↪ costs);
                /* col constraints */
                wcsp.postBinaryConstraint(index+var_ind1*N, index+var_ind2*N, alldiff_
↪ costs);
            }
        }
    }

    /* unary costs */
    size_t var_ind = 0;
    for(size_t row = 0; row < N; row ++) {
        for(size_t col = 0; col < N; col ++) {
            wcsp.postUnaryConstraint(var_ind, costs[row][col]);
            var_ind += 1;
        }
    }
}

int main() {

    srand(123456789);

    size_t N = 5;
    Cost top = N*N*N + 1;

    // N*N*N values, costs for each cell
    LatinCostArray objective_costs;

    // init the costs for each cell
    initLatinCosts(N, objective_costs);

    cout << "Randomly genereated costs : " << endl;
    printCosts(objective_costs);
    cout << endl;

    tb2init();

    ToulBar2::verbose = 0;

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

    // fill in the WeightedCSP object
    WeightedCSP* wcsp = solver->getWCSP();

```

(continues on next page)

(continued from previous page)

```

buildWCSP(*wcsp, objective_costs, N, top);

bool result = solver->solve();

if(result) {

    Cost bestCost = solver->getSolutionValue();
    Cost bestLowerBound = solver->getDDualBound();

    if(!ToulBar2::limited) {
        cout << "Optimal solution found with cost " << bestCost << endl;
    } else {
        cout << "Best solution found with cost " << bestCost << " and best lower
↪bound of " << bestLowerBound << endl;
    }

    // retrieve the solution
    std::vector<Value> solution = solver->getSolution();

    cout << endl << "Best solution : " << endl;
    for(size_t var_ind = 0; var_ind < solution.size(); var_ind++) {
        cout << solution[var_ind] << " ";
        if((var_ind+1) % N == 0) {
            cout << endl;
        }
    }

} else {
    cout << "No solution has been found !" << endl;
}

delete solver;

return 0;
}

```

## 9.5 Bicriteria weighted latin square problem

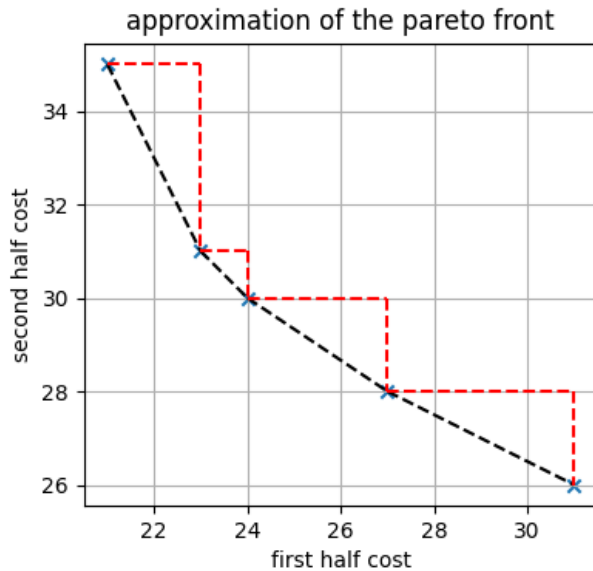
### 9.5.1 Brief description

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.

### 9.5.2 CFN model

Similarly to the *Weighted latin square problem*,  $N \times N$  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.



### 9.5.3 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
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'
```

(continues on next page)



(continued from previous page)

```

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)

    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)

```

(continues on next page)

(continued from previous page)

```

# 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(timeLimit = 60)

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

print('\n')

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

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(timeLimit = 60)

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

```

(continues on next page)

(continued from previous page)

```

# approximate the pareto front
(solutions, costs) = multicfn.ApproximateParetoFront(0, 'min', 1, 'min', showSolutions = 0,
timeLimit = 300, timeLimit_per_solution = 60)

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 objective')
ax.set_ylabel('Second objective')
ax.set_title('Approximation of the Pareto front')
ax.set_aspect('equal')

plt.grid()
plt.show()

```

### 9.5.4 C++ model

The following code using the C++ toulbar2 library API solves the weighted latin square problem.

bicriteria\_latinsquare.cpp

```

#include <iostream>
#include <vector>

#include "core/tb2wcsp.hpp"
#include "mcriteria/multicfn.hpp"
#include "mcriteria/bicriteria.hpp"

using namespace std;

// an alias for storing the variable costs
// first dim is the grid rows and second is the columns
typedef std::vector<std::vector<std::vector<Cost>>> LatinCostArray;

// generate random costs for each variable (cell)
// param N grid size
// param costs the matrix costs
void createCostMatrix(size_t N, LatinCostArray& costs) {

    // N*N*N values, costs for each cell
    costs.resize(N);
    for(auto& col: costs) {
        col.resize(N);
        for(auto& cell: col) {

```

(continues on next page)

(continued from previous page)

```

        cell.resize(N);
        for(size_t val_ind = 0; val_ind < N; val_ind += 1) {
            cell[val_ind] = (rand()%N)+1;
        }
    }
}

// print the costs for each unary variable (cell)
// param costs the cost matrix
void printCosts(LatinCostArray& costs) {

    for(size_t row_ind = 0; row_ind < costs.size(); row_ind++) {
        for(size_t col_ind = 0; col_ind < costs[row_ind].size(); col_ind++) {
            cout << "cell " << row_ind << "-" << col_ind;
            cout << " : ";
            for(auto& cost: costs[row_ind][col_ind]) {
                cout << cost << ", ";
            }
            cout << endl;
        }
    }
}

// fill in a WCSP object with a latin square problem
// param wcsp the wcsp object to fill
// param LatinCostArray the cost matrix
// param N grid size
// top the top value, problem upper bound (the objective is always lower than top)
void buildLatinSquare(WeightedCSP& wcsp, LatinCostArray& costs, size_t N, Cost top) {

    // variables
    for(size_t row = 0; row < N; row++) {
        for(unsigned int col = 0; col < N; col++) {
            wcsp.makeEnumeratedVariable("Cell_" + to_string(row) + "," + to_string(col),
↪0, N-1);
        }
    }

    /* costs for all different constraints (top on diagonal) */
    vector<Cost> alldiff_costs;
    for(unsigned int i = 0; i < N; i++) {
        for(unsigned int j = 0; j < N; j++) {
            if(i == j) {
                alldiff_costs.push_back(top);
            } else {
                alldiff_costs.push_back(0);
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    /* all different constraints */
    for(unsigned int index = 0; index < N; index ++) {
        for(unsigned int var_ind1 = 0; var_ind1 < N; var_ind1 ++) {
            for(unsigned int var_ind2 = var_ind1+1; var_ind2 < N; var_ind2 ++) {
                /* row constraints */
                wcsp.postBinaryConstraint(N*index+var_ind1, N*index+var_ind2, alldiff_
↪ costs);
                /* col constraints */
                wcsp.postBinaryConstraint(index+var_ind1*N, index+var_ind2*N, alldiff_
↪ costs);
            }
        }
    }

    /* unary costs */
    size_t var_ind = 0;
    for(size_t row = 0; row < N; row ++) {
        for(size_t col = 0; col < N; col ++) {
            wcsp.postUnaryConstraint(var_ind, costs[row][col]);
            var_ind += 1;
        }
    }
}

// print a solution as a grid
// param N the size of the grid
// param solution the multicfn solution (dict)
// param point the objective costs (objective space point)
void printSolution(size_t N, MultiCFN::Solution& solution, Bicriteria::Point& point) {
    for(size_t row = 0; row < N; row ++) {
        for(size_t col = 0; col < N; col ++) {
            string var_name = "Cell_" + to_string(row) + "," + to_string(col);
            cout << solution[var_name].substr(1) << " ";
        }
        cout << endl;
    }
    cout << "obj_1 = " << point.first << " ; obj2 = " << point.second << endl;
}

// main function
int main() {

    srand(123456789);

    size_t N = 4;
    Cost top = N*N*N + 1;

    // two cost matrice
    LatinCostArray costs_obj1, costs_obj2;

```

(continues on next page)

(continued from previous page)

```

// init the objective with random costs
createCostMatrix(N, costs_obj1);
createCostMatrix(N, costs_obj2);

// cout << "Randomly generated costs : " << endl;
// printCosts(costs_obj1);
// cout << endl << endl;
// printCosts(costs_obj2);

tb2init();
initCosts();

// create the two wcsp objects
WeightedCSP* wcsp1 = WeightedCSP::makeWeightedCSP(top);
WeightedCSP* wcsp2 = WeightedCSP::makeWeightedCSP(top);

// initialize the objects as a latin square problem objectives with two different
↳ objectives
buildLatinSquare(*wcsp1, costs_obj1, N, top);
buildLatinSquare(*wcsp2, costs_obj2, N, top);

// creation of the multicfn
MultiCFN mcfn;
mcfn.push_back(dynamic_cast<WCSP*>(wcsp1));
mcfn.push_back(dynamic_cast<WCSP*>(wcsp2));

// computation of the supported points of the biobjective problem
Bicriteria::computeSupportedPoints(&mcfn, std::make_pair(Bicriteria::OptimDir::Optim_
↳ Min, Bicriteria::OptimDir::Optim_Min));

// access to the computed solutions and their objective values
std::vector<MultiCFN::Solution> solutions = Bicriteria::getSolutions();
std::vector<Bicriteria::Point> points = Bicriteria::getPoints();

// print all solutions computed
cout << "Resulting solutions: " << endl;
for(size_t sol_index = 0; sol_index < solutions.size(); sol_index++) {
    printSolution(N, solutions[sol_index], points[sol_index]);
    cout << endl;
}

// delete the wcsp objects
delete wcsp1;
delete wcsp2;

return 0;
}

```

The above code can be compiled with the following command:

```
g++ -O3 -std=c++17 -Wall -DBOOST -DLONGLONG_COST -DLONGDOUBLE_PROB -I $YOUR_TB2_INCLUDE_
```

(continues on next page)

(continued from previous page)

```
→PATH main.cpp -c -o main.o
```

Where `$YOUR_TB2_INCLUDE_PATH` is the path to the ToulBar2 src directory. And the compiled program is obtained via :

```
g++ -O3 -std=c++17 -Wall -DBOOST -DLONGLONG_COST -DLONGDOUBLE_PROB main.o -o main -L
→$YOUR_LIBTB2_PATH -ltb2 -lgmp -lboost_graph -lboost_iostreams -lz -llzma
```

Where `$YOUR_LIBTB2_PATH` is the path to the ToulBar2 compiled library. When running the program, do not forget to set the `$(LD_LIBRARY_PATH)` environment variable in Linux.

## 9.6 Radio link frequency assignment problem

### 9.6.1 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:**

$$a_1 * nc_1 + \dots + a_4 * nc_4 + b_1 * nv_1 + \dots + b_4 * nv_4$$

where  $nc_i$  is the number of violated constraints with cost  $a_i$  and  $nv_i$  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.

### 9.6.2 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  $a_4/b_4$ ).

### 9.6.3 Data

Original data files can be downloaded from the cost function library [FullIRLFAP](#). 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.

### 9.6.4 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),
↪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()[:1]
            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])

```

(continues on next page)



(continued from previous page)

```

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)
            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('rlfap.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!')

```

## 9.7 Frequency assignment problem with polarization

### 9.7.1 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

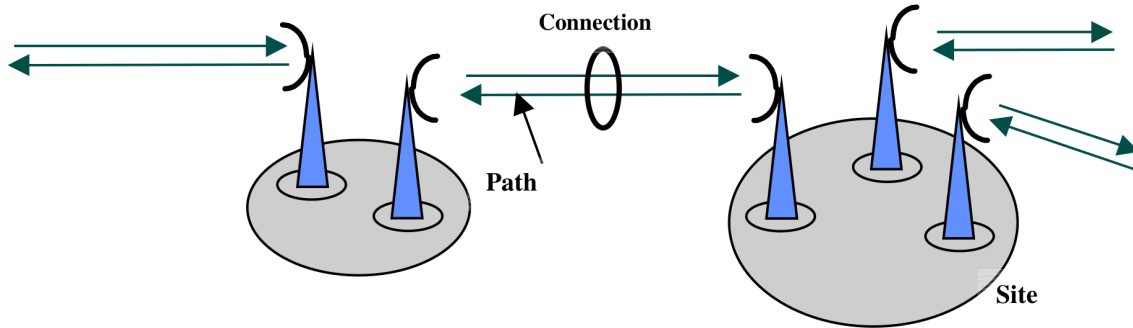
(frequency,polarization) to every radio communication link (also called a path). Frequencies are integer values taken in finite domains. Polarizations are in  $\{-1,1\}$ . Constraints are :

- (I) two paths must use equal or different frequencies ( $f_i=f_j$  or  $f_i \neq f_j$ ),
- (II) the absolute difference between two frequencies should exactly be equal or different to a given number  $e$  ( $|f_i-f_j|=e$  or  $|f_i-f_j| \neq e$ ),
- (III) two paths must use equal or different polarizations ( $p_i=p_j$  or  $p_i \neq p_j$ ),
- (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) ( $|f_i-f_j| \geq g_l$  if  $p_i=p_j$  else  $|f_i-f_j| \geq d_l$ ), 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:  $10 * k * \text{nbsoft}^2 + 10 * \text{nbsoft} * V(k-1) + V(k-2) + V(k-3) + \dots + V_0$

where nbsoft is the number of soft constraints in the problem and  $k$  the feasible relaxation level and  $V(i)$  the number of violated constraints of level  $i$ .



### 9.7.2 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  $10 * \text{nbsoft}$  (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  $10 * (k+1) * \text{nbsoft}^2 + 1$ .

### 9.7.3 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](#).

### 9.7.4 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()
        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
```

(continues on next page)

(continued from previous page)

```

        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
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:

```

(continues on next page)

(continued from previous page)

```

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:
                    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)

```

## 9.8 Mendelian error detection problem

### 9.8.1 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.

### 9.8.2 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.

### 9.8.3 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.

### 9.8.4 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))
```

(continues on next page)

(continued from previous page)

```

        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)

#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:           #_
                                    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:

```

(continues on next page)

(continued from previous page)

```

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

```

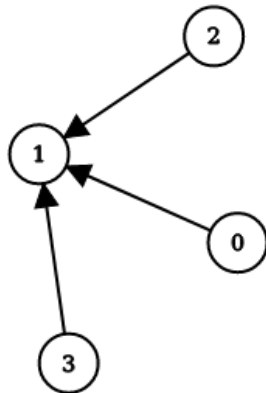
## 9.9 Block modeling problem

### 9.9.1 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  $k \times k$  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:

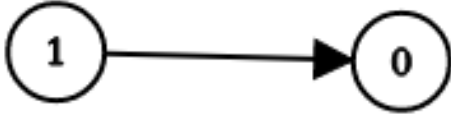


and corresponds to the following matrix:

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

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

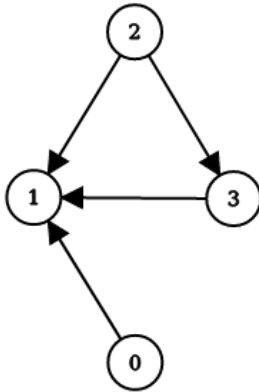




and this graph corresponds to the following M matrix:

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

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.

### 9.9.2 CFN model

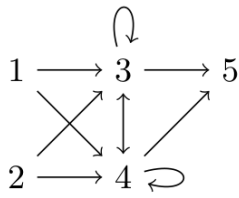
We create N variables, one for every node of the graph, with domain size k representing the clustering. We add k\*k Boolean variables for representing M.

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

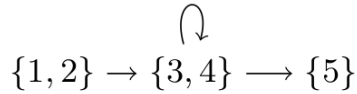
### 9.9.3 Data

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)

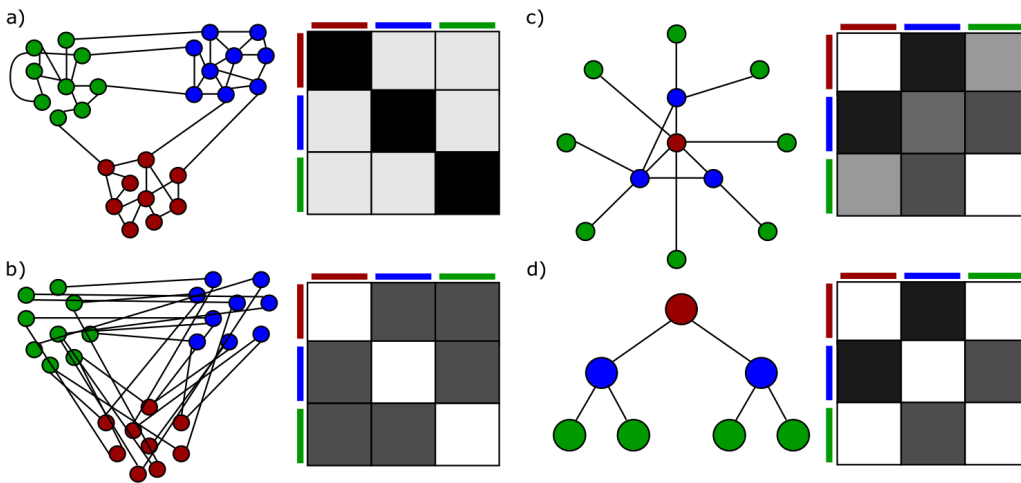


$$G = \left( \begin{array}{cc|cc|c} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 \end{array} \right)$$



$$M = \left( \begin{array}{ccc} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{array} \right)$$

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](#).

## 9.9.4 Python model

The following code using `pytoulbar2` library solves the corresponding cost function network (e.g. “python3 block-model.py simple.mat 3”).

`blockmodel.py`

```
import sys
import pytoulbar2

#read adjacency 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
```

(continues on next page)

(continued from previous page)

```

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):
    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:

```

(continues on next page)

(continued from previous page)

```

        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)
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]))

```

## 9.10 Airplane landing problem

### 9.10.1 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.

### 9.10.2 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.

### 9.10.3 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.

### 9.10.4 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))
```

(continues on next page)

(continued from previous page)

```

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

```

## 9.11 Warehouse location problem

### 9.11.1 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.

### 9.11.2 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.

### 9.11.3 Data

Original data files can be download from the cost function library [warehouses](#). Their format is described [here](#).

### 9.11.4 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

uncapacitated = True    # if True then do not enforce capacity constraints on warehouses

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

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

top += sum(CostW)

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

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

(continues on next page)

(continued from previous page)

```

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

# optional: add capacity constraint on each warehouse
if not(uncapacitated):
    for i in range(N):
        Problem.AddGeneralizedLinearConstraint([N+j, i, min(max(Capacity), Demand[j]))
↪for j in range(M)], '<=', Capacity[i])

#Problem.Dump('warehouse.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!')

```

## 9.12 Square packing problem

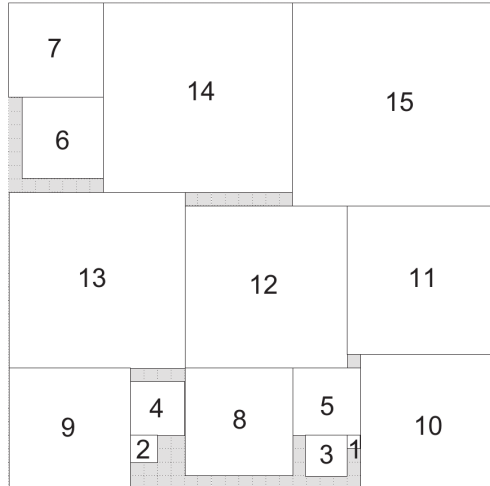
### 9.12.1 Brief description

We have  $N$  squares of respective size  $1 \times 1, 2 \times 2, \dots, N \times N$ . We have to fit them without overlaps into a square of size  $S \times S$ .

Results up to  $N=56$  are given [here](#).

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





### 9.12.2 CFN model

We create an integer variable of domain size  $(S-i) \times (S-i)$  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.

### 9.12.3 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
    range(S-i) for j in range(S-i)])
```

(continues on next page)

(continued from previous page)

```

#binary hard constraints for overlapping squares
for i in range(N):
    for j in range(i+1,N):
        ListConstraintsOverlaps = []
        for a in [S*k+1 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)
#Problem.Dump('Square.cfn')

```

(continues on next page)

(continued from previous page)

```

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
↪ 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!')

```

### 9.12.4 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 -DWCSFORMATONLY 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 -DWCSFORMATONLY libtb2.so

#include "toulbar2lib.hpp"

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

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

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

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

    initCosts(); // last check for compatibility issues between ToulBar2 options and

```

(continues on next page)

(continued from previous page)

`↪ Cost data-type`

```

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

for (int i=0; i<N; i++) {
    solver->getWCSP()->makeEnumeratedVariable(to_string("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;
}

delete solver;
return 0;
}

```

## 9.13 Square soft packing problem

### 9.13.1 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.

### 9.13.2 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  $N \times N$  are all stacked together. The cost of this is  $N^4$ , so we will take  $N^4 + 1$  as the initial upper bound.

### 9.13.3 Python model

The following code using `pytoulbar2` library solves the corresponding cost function network (e.g. “python3 square-soft.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
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
```

(continues on next page)

(continued from previous page)

```

        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)
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)

```

(continues on next page)

(continued from previous page)

```
else:
    print('No solution found!')
```

### 9.13.4 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 -DWCSPPFORMATONLY 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 -DWCSPPFORMATONLY libtb2.so

#include "toulbar2lib.hpp"

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

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

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

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

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

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

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

    for (int i=0; i < N; i++) {
```

(continues on next page)

(continued from previous page)

```

    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;
}

delete solver;
return 0;
}

```

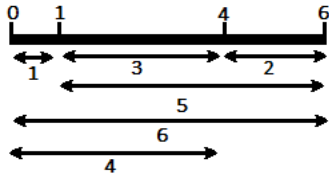
## 9.14 Golomb ruler problem

### 9.14.1 Brief description

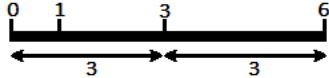
A golomb ruler of order  $N$  is a set of integer marks  $0=a_1 < a_2 < a_3 < a_4 < \dots < a_N$  such that each difference between two  $a_k$ 's is unique.



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  $N$ .

### 9.14.2 CFN model

We create  $N$  variables, one for each integer mark  $a_k$ . 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  $a_k$ '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.

### 9.14.3 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

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):
```

(continues on next page)

(continued from previous page)

```

        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)

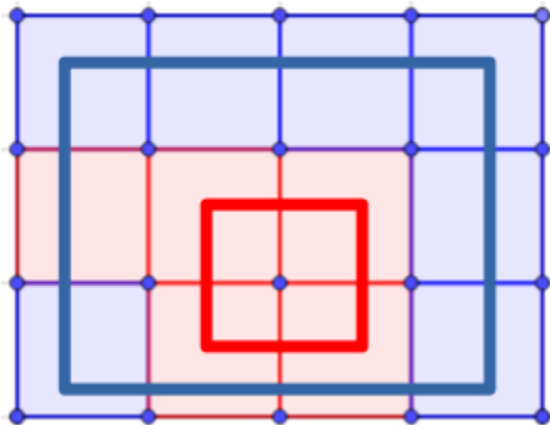
```

## 9.15 Board coloration problem

### 9.15.1 Brief description

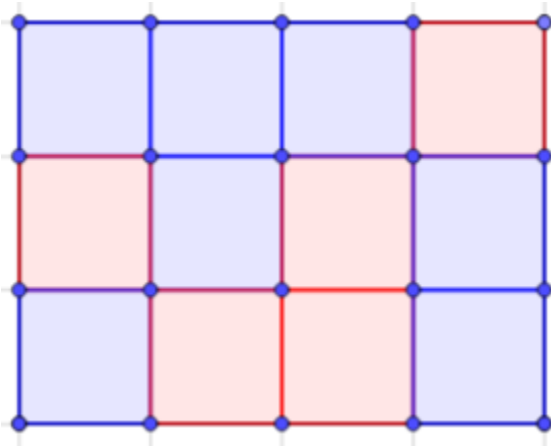
Given a rectangular board with dimension  $n*m$ , the goal is to color the cells such that any inner rectangle included inside the board doesn't have all its corners colored with the same color. The goal is to minimize the number of colors used.

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:



### 9.15.2 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.

### 9.15.3 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):
```

(continues on next page)

(continued from previous page)

```

        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)

#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)

```

(continues on next page)

(continued from previous page)

```
#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!')
```

## 9.16 Learning to play the Sudoku

### 9.16.1 Available

- Presentation

- GitHub code 

- Data GitHub code 

## 9.17 Learning car configuration preferences

### 9.17.1 Brief description

Renault car configuration system: learning user preferences.

### 9.17.2 Available

- Presentation

- GitHub code 


- Data GitHub code 

## 9.18 Visual Sudoku Tutorial

### 9.18.1 Brief description

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

### 9.18.2 Available

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

## 9.19 Visual Sudoku Application


### 9.19.1 Brief description

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

### 9.19.2 Available

#### Software

**Software** adapted by Simon de Givry (@ INRAE, 2022) in order to use **toulbar2** solver, from a [tutorial](#) by Adrian

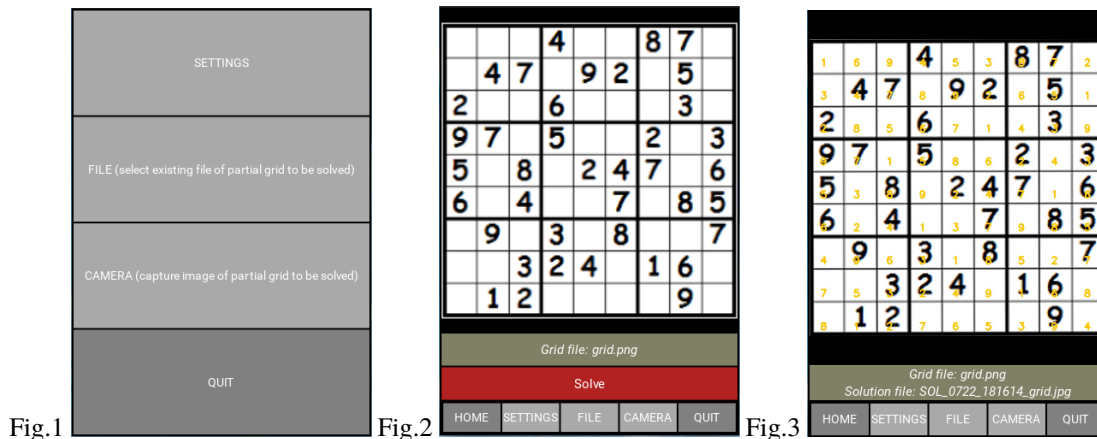
Rosebrock (@ PyImageSearch, 2022) : [GitHub code](#) 

#### 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 : 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.

## 9.20 Visual Sudoku App for Android

### 9.20.1 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](#).

### 9.20.2 Source Code

[GitHub code](#) 

### 9.20.3 Download and Install

To install the ‘Visual Sudoku’ application on smartphone :

- 1) **Download** the **visalsudoku-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.

#### 9.20.4 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.

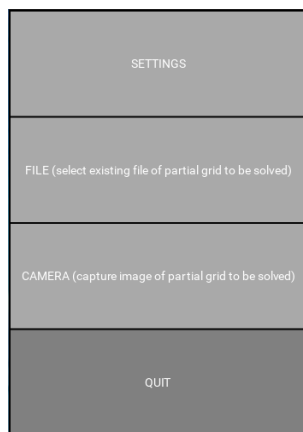


Fig.1

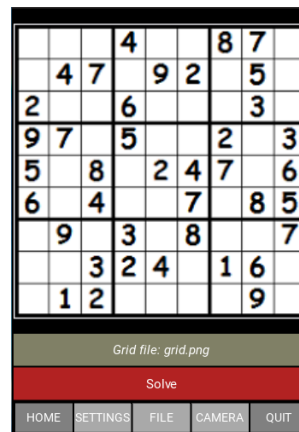
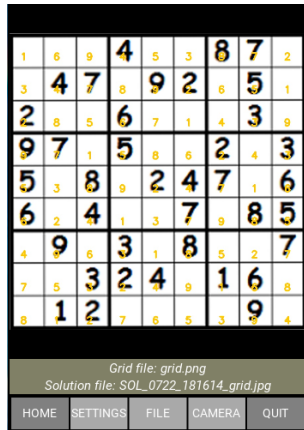


Fig.2

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

## 9.21 A sudoku code

### 9.21.1 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`.

### 9.21.2 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) :



Web service to further get a detailed description of what the program does (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' ↪
```

(continues on next page)

(continued from previous page)

```
↪ 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

## 10.1 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 less 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 Optimization problems, linear programs 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>.

## 10.2 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
# cmake ..
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].

## 10.3 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 [toulbar](#) (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
```

## 10.4 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**, **.wbo**, **.wbo.gz**, **.wbo.bz2**, **.wbo.xz**, **.lp**, **.lp.gz**, **.lp.bz2**, **.lp.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.

## 10.5 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.
Reverse original DAC dual bound: 20 (+10.000%)
Cost function decomposition time : 0.000 seconds.
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, 0.001 seconds)
Optimality gap: [21, 28] 25.000 % (9 backtracks, 20 nodes, 0.002 seconds)
New solution: 27 (9 backtracks, 27 nodes, depth 7, 0.002 seconds)
Optimality gap: [21, 27] 22.222 % (14 backtracks, 32 nodes, 0.002 seconds)
Optimality gap: [22, 27] 18.519 % (17 backtracks, 41 nodes, 0.002 seconds)
Optimality gap: [23, 27] 14.815 % (46 backtracks, 115 nodes, 0.004 seconds)
Optimality gap: [24, 27] 11.111 % (83 backtracks, 210 nodes, 0.005 seconds)
Optimality gap: [25, 27] 7.407 % (102 backtracks, 275 nodes, 0.007 seconds)
Optimality gap: [27, 27] 0.000 % (105 backtracks, 297 nodes, 0.007 seconds)
Node redundancy during HBFS: 28.956
Optimum: 27 in 105 backtracks and 297 nodes ( 544 removals by DEE) and 0.007
↪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.
```

(continues on next page)

(continued from previous page)

```
Reverse original DAC dual bound: 20 (+10.000%)
Cost function decomposition time : 0.000 seconds.
New solution: 27 (0 backtracks, 0 nodes, depth 1, 0.120 seconds)
INCOPI solving time: 0.326 seconds.
Preprocessing time: 0.328 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] 25.92%
Optimality gap: [21, 27] 22.222 % (14 backtracks, 34 nodes, 0.328 seconds)
Optimality gap: [22, 27] 18.519 % (19 backtracks, 48 nodes, 0.329 seconds)
Optimality gap: [23, 27] 14.815 % (79 backtracks, 189 nodes, 0.332 seconds)
Optimality gap: [24, 27] 11.111 % (82 backtracks, 201 nodes, 0.332 seconds)
Optimality gap: [25, 27] 7.407 % (87 backtracks, 222 nodes, 0.333 seconds)
Optimality gap: [27, 27] 0.000 % (99 backtracks, 258 nodes, 0.333 seconds)
Node redundancy during HBFS: 23.256
Optimum: 27 in 99 backtracks and 258 nodes ( 539 removals by DEE) and 0.333 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.
Reverse original DAC dual bound: 20 (+10.000%)
Cost function decomposition time : 0.000 seconds.
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, 0.002 seconds)
Optimality gap: [21, 27] 22.222 % (6 backtracks, 14 nodes, 0.002 seconds)
Optimality gap: [22, 27] 18.519 % (23 backtracks, 57 nodes, 0.003 seconds)
Optimality gap: [23, 27] 14.815 % (69 backtracks, 159 nodes, 0.005 seconds)
Optimality gap: [24, 27] 11.111 % (76 backtracks, 180 nodes, 0.006 seconds)
Optimality gap: [25, 27] 7.407 % (92 backtracks, 240 nodes, 0.007 seconds)
Optimality gap: [27, 27] 0.000 % (100 backtracks, 286 nodes, 0.007 seconds)
Node redundancy during HBFS: 29.720
Optimum: 27 in 100 backtracks and 286 nodes ( 569 removals by DEE) and 0.008_
↳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 : 6.1e-05 seconds.
Preprocessing time: 0.159001 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: 7769 (0 backtracks, 98 nodes, depth 3, 0.197 seconds)
--- [1] LDS 1 --- (98 nodes)
c 8388608 Bytes allocated for long long stack.
New solution: 5848 (1 backtracks, 284 nodes, depth 4, 0.280 seconds)
New solution: 5384 (3 backtracks, 399 nodes, depth 4, 0.347 seconds)
New solution: 5039 (4 backtracks, 468 nodes, depth 4, 0.376 seconds)
New solution: 4740 (8 backtracks, 642 nodes, depth 4, 0.510 seconds)
--- [1] LDS 2 --- (739 nodes)
New solution: 4716 (37 backtracks, 976 nodes, depth 5, 0.863 seconds)
New solution: 4693 (44 backtracks, 1020 nodes, depth 5, 0.881 seconds)
New solution: 4690 (49 backtracks, 1125 nodes, depth 5, 0.984 seconds)
New solution: 4683 (79 backtracks, 1481 nodes, depth 5, 1.361 seconds)
New solution: 4679 (97 backtracks, 1607 nodes, depth 4, 1.476 seconds)
New solution: 4527 (105 backtracks, 1773 nodes, depth 5, 1.632 seconds)
New solution: 4494 (109 backtracks, 1867 nodes, depth 5, 1.700 seconds)
New solution: 4487 (128 backtracks, 2068 nodes, depth 5, 1.869 seconds)
New solution: 4457 (139 backtracks, 2184 nodes, depth 5, 1.956 seconds)
New solution: 4442 (150 backtracks, 2272 nodes, depth 5, 2.014 seconds)
New solution: 3986 (169 backtracks, 2444 nodes, depth 5, 2.151 seconds)
New solution: 3858 (171 backtracks, 2488 nodes, depth 5, 2.176 seconds)
New solution: 3709 (180 backtracks, 2570 nodes, depth 5, 2.248 seconds)
--- [1] LDS 4 --- (2654 nodes)
New solution: 3706 (511 backtracks, 3957 nodes, depth 6, 3.490 seconds)
New solution: 3683 (513 backtracks, 4008 nodes, depth 7, 3.511 seconds)
New solution: 3662 (515 backtracks, 4034 nodes, depth 7, 3.522 seconds)
New solution: 3634 (660 backtracks, 4735 nodes, depth 7, 3.971 seconds)
New solution: 3632 (661 backtracks, 4742 nodes, depth 6, 3.973 seconds)
New solution: 3630 (663 backtracks, 4750 nodes, depth 6, 3.974 seconds)
New solution: 3619 (685 backtracks, 4842 nodes, depth 7, 4.029 seconds)
New solution: 3600 (736 backtracks, 5103 nodes, depth 7, 4.178 seconds)
New solution: 3587 (739 backtracks, 5124 nodes, depth 7, 4.184 seconds)
New solution: 3586 (741 backtracks, 5133 nodes, depth 6, 4.185 seconds)
New solution: 3585 (742 backtracks, 5137 nodes, depth 6, 4.185 seconds)
New solution: 3584 (743 backtracks, 5141 nodes, depth 6, 4.185 seconds)
New solution: 3583 (744 backtracks, 5144 nodes, depth 6, 4.185 seconds)
New solution: 3570 (757 backtracks, 5208 nodes, depth 7, 4.214 seconds)
New solution: 3563 (758 backtracks, 5220 nodes, depth 7, 4.217 seconds)
New solution: 3551 (762 backtracks, 5236 nodes, depth 7, 4.222 seconds)
New solution: 3543 (764 backtracks, 5248 nodes, depth 7, 4.225 seconds)
New solution: 3524 (765 backtracks, 5258 nodes, depth 7, 4.227 seconds)
```

(continues on next page)



(continued from previous page)

```

New solution: 3501 (795 backtracks, 5428 nodes, depth 7, 4.382 seconds)
New solution: 3499 (980 backtracks, 6144 nodes, depth 6, 5.110 seconds)
New solution: 3477 (985 backtracks, 6172 nodes, depth 7, 5.119 seconds)
New solution: 3443 (989 backtracks, 6196 nodes, depth 7, 5.125 seconds)
--- [1] Search with no discrepancy limit --- (7446 nodes)
New solution: 3442 (4423 backtracks, 13529 nodes, depth 28, 8.371 seconds)
New solution: 3441 (4424 backtracks, 13534 nodes, depth 28, 8.371 seconds)
New solution: 3424 (4462 backtracks, 13619 nodes, depth 32, 8.388 seconds)
New solution: 3422 (4463 backtracks, 13620 nodes, depth 29, 8.388 seconds)
New solution: 3420 (4464 backtracks, 13624 nodes, depth 28, 8.388 seconds)
New solution: 3412 (4474 backtracks, 13648 nodes, depth 28, 8.395 seconds)
New solution: 3410 (4475 backtracks, 13650 nodes, depth 26, 8.396 seconds)
New solution: 3404 (4582 backtracks, 13870 nodes, depth 28, 8.443 seconds)
New solution: 3402 (4583 backtracks, 13872 nodes, depth 26, 8.443 seconds)
New solution: 3400 (4584 backtracks, 13876 nodes, depth 25, 8.443 seconds)
New solution: 3391 (4586 backtracks, 13887 nodes, depth 29, 8.444 seconds)
New solution: 3389 (4587 backtracks, 13889 nodes, depth 27, 8.444 seconds)
Optimality gap: [100, 3389] 97.049 % (33368 backtracks, 71431 nodes, 31.505 seconds)
Optimality gap: [292, 3389] 91.384 % (50312 backtracks, 105319 nodes, 46.031
↪seconds)
Optimality gap: [475, 3389] 85.984 % (52721 backtracks, 110137 nodes, 48.301
↪seconds)
Optimality gap: [921, 3389] 72.824 % (62170 backtracks, 129035 nodes, 54.846
↪seconds)
Optimality gap: [1864, 3389] 44.999 % (62688 backtracks, 130071 nodes, 55.259
↪seconds)
Optimality gap: [3086, 3389] 8.941 % (62722 backtracks, 130139 nodes, 55.307
↪seconds)
Optimality gap: [3156, 3389] 6.875 % (62723 backtracks, 130141 nodes, 55.309
↪seconds)
Optimum: 3389 in 62724 backtracks and 130143 nodes ( 680102 removals by DEE) and 55.
↪310 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 : 5.5e-05 seconds.
Preprocessing time: 0.154142 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, 108 nodes, depth 109, 0.206 seconds)
Problem decomposition in 55 clusters with size distribution: min: 1 median: 5 mean:
↪

```

(continues on next page)

(continued from previous page)

```

↪4.782 max: 12
***** Restart 1 with 1 discrepancies and UB=7566 ***** (108 nodes)
New solution: 7468 (0 backtracks, 108 nodes, depth 1, 0.208 seconds)
New solution: 7360 (0 backtracks, 112 nodes, depth 2, 0.209 seconds)
New solution: 7359 (0 backtracks, 112 nodes, depth 1, 0.209 seconds)
New solution: 7357 (0 backtracks, 112 nodes, depth 1, 0.210 seconds)
New solution: 7337 (0 backtracks, 112 nodes, depth 1, 0.210 seconds)
New solution: 7267 (0 backtracks, 112 nodes, depth 1, 0.211 seconds)
New solution: 5958 (0 backtracks, 124 nodes, depth 2, 0.219 seconds)
New solution: 5900 (0 backtracks, 124 nodes, depth 1, 0.219 seconds)
New solution: 5756 (0 backtracks, 128 nodes, depth 2, 0.222 seconds)
New solution: 5709 (0 backtracks, 128 nodes, depth 1, 0.223 seconds)
New solution: 5653 (0 backtracks, 128 nodes, depth 1, 0.224 seconds)
New solution: 5619 (0 backtracks, 131 nodes, depth 2, 0.227 seconds)
New solution: 5608 (0 backtracks, 133 nodes, depth 2, 0.231 seconds)
New solution: 5171 (5 backtracks, 176 nodes, depth 2, 0.238 seconds)
New solution: 5065 (7 backtracks, 188 nodes, depth 2, 0.240 seconds)
New solution: 5062 (7 backtracks, 189 nodes, depth 2, 0.241 seconds)
New solution: 4972 (7 backtracks, 189 nodes, depth 1, 0.242 seconds)
New solution: 4962 (7 backtracks, 189 nodes, depth 1, 0.243 seconds)
New solution: 4906 (17 backtracks, 238 nodes, depth 2, 0.291 seconds)
New solution: 4905 (22 backtracks, 274 nodes, depth 1, 0.303 seconds)
New solution: 4869 (23 backtracks, 282 nodes, depth 2, 0.309 seconds)
New solution: 4807 (26 backtracks, 321 nodes, depth 2, 0.349 seconds)
New solution: 4797 (26 backtracks, 321 nodes, depth 1, 0.349 seconds)
New solution: 4625 (27 backtracks, 333 nodes, depth 2, 0.354 seconds)
New solution: 4380 (32 backtracks, 354 nodes, depth 2, 0.361 seconds)
New solution: 4372 (32 backtracks, 354 nodes, depth 1, 0.361 seconds)
New solution: 4370 (32 backtracks, 354 nodes, depth 1, 0.362 seconds)
New solution: 4368 (32 backtracks, 354 nodes, depth 1, 0.364 seconds)
New solution: 4349 (32 backtracks, 357 nodes, depth 2, 0.365 seconds)
New solution: 4153 (33 backtracks, 375 nodes, depth 2, 0.372 seconds)
New solution: 4144 (34 backtracks, 379 nodes, depth 2, 0.374 seconds)
New solution: 3714 (43 backtracks, 462 nodes, depth 2, 0.423 seconds)
New solution: 3611 (44 backtracks, 464 nodes, depth 1, 0.426 seconds)
New solution: 3510 (46 backtracks, 482 nodes, depth 2, 0.432 seconds)
***** Restart 2 with 2 discrepancies and UB=3510 ***** (1935 nodes)
New solution: 3477 (1220 backtracks, 6501 nodes, depth 3, 5.336 seconds)
New solution: 3476 (1220 backtracks, 6501 nodes, depth 1, 5.338 seconds)
New solution: 3473 (1220 backtracks, 6501 nodes, depth 1, 5.339 seconds)
New solution: 3461 (1220 backtracks, 6506 nodes, depth 2, 5.343 seconds)
New solution: 3449 (1220 backtracks, 6507 nodes, depth 2, 5.344 seconds)
New solution: 3427 (1220 backtracks, 6507 nodes, depth 1, 5.345 seconds)
New solution: 3426 (1235 backtracks, 6607 nodes, depth 2, 5.426 seconds)
New solution: 3425 (1237 backtracks, 6617 nodes, depth 2, 5.429 seconds)
New solution: 3424 (1237 backtracks, 6619 nodes, depth 2, 5.432 seconds)
New solution: 3407 (1272 backtracks, 6871 nodes, depth 2, 5.662 seconds)
***** Restart 3 with 2 discrepancies and UB=3407 ***** (8284 nodes)
New solution: 3402 (1604 backtracks, 8939 nodes, depth 3, 7.663 seconds)
***** Restart 4 with 2 discrepancies and UB=3402 ***** (10196 nodes)

Time limit expired... Aborting...

```

(continues on next page)

(continued from previous page)

```
Dual bound: 0
Primal bound: 3402 in 2174 backtracks and 11657 nodes ( 96931 removals by DEE) and
→9.975 seconds.
end.
```

- 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 : 0.000202 seconds.
Preprocessing time: 0.35719 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: [10000, 436552965] 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: 2535618 (0 backtracks, 231 nodes, depth 232, 0.459 seconds)
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=2535618 ***** (231 nodes)
New solution: 2535518 (0 backtracks, 231 nodes, depth 1, 0.467 seconds)
New solution: 2525819 (0 backtracks, 231 nodes, depth 1, 0.472 seconds)
New solution: 2515619 (2 backtracks, 237 nodes, depth 1, 0.475 seconds)
New solution: 2515519 (2 backtracks, 237 nodes, depth 1, 0.477 seconds)
New solution: 2515518 (2 backtracks, 237 nodes, depth 1, 0.477 seconds)
New solution: 2515516 (3 backtracks, 240 nodes, depth 1, 0.481 seconds)
New solution: 2515514 (3 backtracks, 240 nodes, depth 1, 0.481 seconds)
New solution: 2495513 (3 backtracks, 240 nodes, depth 1, 0.482 seconds)
New solution: 2485515 (3 backtracks, 240 nodes, depth 1, 0.483 seconds)
New solution: 2485513 (3 backtracks, 240 nodes, depth 1, 0.484 seconds)
New solution: 2475412 (3 backtracks, 240 nodes, depth 1, 0.486 seconds)
New solution: 2465411 (3 backtracks, 240 nodes, depth 1, 0.487 seconds)
New solution: 2465210 (3 backtracks, 240 nodes, depth 1, 0.488 seconds)
New solution: 2465011 (4 backtracks, 246 nodes, depth 2, 0.496 seconds)
New solution: 2455217 (8 backtracks, 272 nodes, depth 2, 0.502 seconds)
New solution: 2455016 (8 backtracks, 272 nodes, depth 1, 0.504 seconds)
New solution: 2454814 (8 backtracks, 272 nodes, depth 1, 0.505 seconds)
New solution: 2454715 (9 backtracks, 281 nodes, depth 2, 0.510 seconds)
New solution: 2434722 (13 backtracks, 303 nodes, depth 2, 0.524 seconds)
New solution: 2434622 (15 backtracks, 317 nodes, depth 2, 0.528 seconds)
New solution: 2434522 (15 backtracks, 317 nodes, depth 1, 0.530 seconds)
New solution: 2434520 (15 backtracks, 317 nodes, depth 1, 0.532 seconds)
New solution: 2434519 (15 backtracks, 317 nodes, depth 1, 0.534 seconds)
New solution: 2425017 (16 backtracks, 327 nodes, depth 2, 0.538 seconds)
New solution: 2425016 (24 backtracks, 392 nodes, depth 2, 0.558 seconds)
New solution: 1475515 (24 backtracks, 395 nodes, depth 2, 0.561 seconds)
```

(continues on next page)

(continued from previous page)

```

New solution: 1455716 (24 backtracks, 395 nodes, depth 1, 0.565 seconds)
New solution: 1455715 (24 backtracks, 395 nodes, depth 1, 0.565 seconds)
New solution: 1395508 (24 backtracks, 399 nodes, depth 2, 0.569 seconds)
New solution: 1395408 (26 backtracks, 407 nodes, depth 2, 0.573 seconds)
New solution: 1384805 (28 backtracks, 416 nodes, depth 2, 0.577 seconds)
New solution: 1384803 (28 backtracks, 417 nodes, depth 2, 0.581 seconds)
New solution: 1374803 (28 backtracks, 417 nodes, depth 1, 0.584 seconds)
New solution: 1374401 (28 backtracks, 417 nodes, depth 1, 0.585 seconds)
New solution: 1374297 (37 backtracks, 468 nodes, depth 1, 0.594 seconds)
New solution: 1374197 (37 backtracks, 468 nodes, depth 1, 0.597 seconds)
New solution: 1364197 (37 backtracks, 468 nodes, depth 1, 0.598 seconds)
New solution: 1364196 (48 backtracks, 527 nodes, depth 1, 0.632 seconds)
New solution: 1354399 (66 backtracks, 608 nodes, depth 1, 0.666 seconds)
New solution: 1354398 (72 backtracks, 626 nodes, depth 1, 0.674 seconds)
New solution: 1354397 (72 backtracks, 626 nodes, depth 1, 0.675 seconds)
New solution: 1354396 (74 backtracks, 633 nodes, depth 1, 0.682 seconds)
New solution: 1344097 (76 backtracks, 644 nodes, depth 2, 0.694 seconds)
New solution: 1344000 (85 backtracks, 667 nodes, depth 1, 0.705 seconds)
New solution: 1343999 (86 backtracks, 669 nodes, depth 1, 0.708 seconds)
New solution: 1343998 (88 backtracks, 676 nodes, depth 1, 0.713 seconds)
New solution: 1343800 (88 backtracks, 676 nodes, depth 1, 0.714 seconds)
New solution: 1343797 (89 backtracks, 678 nodes, depth 1, 0.719 seconds)
New solution: 1343698 (129 backtracks, 906 nodes, depth 1, 0.858 seconds)
New solution: 1343695 (140 backtracks, 979 nodes, depth 1, 0.889 seconds)
New solution: 373807 (142 backtracks, 987 nodes, depth 2, 0.898 seconds)
New solution: 363805 (155 backtracks, 1052 nodes, depth 2, 0.920 seconds)
New solution: 343800 (155 backtracks, 1058 nodes, depth 2, 0.923 seconds)
New solution: 343799 (158 backtracks, 1070 nodes, depth 1, 0.931 seconds)
New solution: 343798 (160 backtracks, 1074 nodes, depth 1, 0.934 seconds)
New solution: 343697 (161 backtracks, 1078 nodes, depth 1, 0.939 seconds)
New solution: 343593 (162 backtracks, 1084 nodes, depth 2, 0.945 seconds)
***** Restart 2 with 2 discrepancies and UB=343593 ***** (2778 nodes)

Time limit expired... Aborting...
Dual bound: 10000
Primal bound: 343593 in 1127 backtracks and 7051 nodes ( 31649 removals by DEE) and
↪4.997 seconds.
end.

```

- Download file 404.wcsp.xz. Solve it using Depth-First Branch 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.
Reverse original DAC dual bound: 51 (+19.608%)
Reverse original DAC dual bound: 53 (+3.774%)
Reverse original DAC dual bound: 54 (+1.852%)
Cost function decomposition time : 0.000 seconds.
Preprocessing time: 0.010 seconds.
88 unassigned variables, 228 values in all current domains (med. size:2, max

```

(continues on next page)

(continued from previous page)

```

↪size:4) and 591 non-unary cost functions (med. arity:2, med. degree:13)
Initial lower and upper bounds: [54, 164] 67.073%
Get root cluster C44 with max. size: 20
Get root cluster C27 with max. size: 14
Tree decomposition width : 19
Tree decomposition height : 43
Number of clusters : 47
Tree decomposition time: 0.003 seconds.
New solution: 126 (33 backtracks, 61 nodes, depth 3, 0.014 seconds)
Optimality gap: [58, 126] 53.968 % (33 backtracks, 61 nodes, 0.014 seconds)
New solution: 125 (46 backtracks, 88 nodes, depth 3, 0.014 seconds)
Optimality gap: [64, 125] 48.800 % (46 backtracks, 88 nodes, 0.014 seconds)
New solution: 117 (136 backtracks, 278 nodes, depth 3, 0.016 seconds)
Optimality gap: [84, 117] 28.205 % (136 backtracks, 278 nodes, 0.017 seconds)
Optimality gap: [86, 117] 26.496 % (185 backtracks, 416 nodes, 0.018 seconds)
Optimality gap: [88, 117] 24.786 % (256 backtracks, 662 nodes, 0.020 seconds)
Optimality gap: [93, 117] 20.513 % (316 backtracks, 812 nodes, 0.021 seconds)
Optimality gap: [96, 117] 17.949 % (327 backtracks, 866 nodes, 0.022 seconds)
Optimality gap: [97, 117] 17.094 % (339 backtracks, 911 nodes, 0.022 seconds)
New solution: 114 (353 backtracks, 969 nodes, depth 3, 0.023 seconds)
Optimality gap: [98, 114] 14.035 % (353 backtracks, 969 nodes, 0.023 seconds)
Optimality gap: [100, 114] 12.281 % (388 backtracks, 1087 nodes, 0.024 seconds)
Optimality gap: [101, 114] 11.404 % (397 backtracks, 1195 nodes, 0.025 seconds)
Optimality gap: [102, 114] 10.526 % (410 backtracks, 1248 nodes, 0.026 seconds)
Optimality gap: [103, 114] 9.649 % (418 backtracks, 1271 nodes, 0.026 seconds)
Optimality gap: [104, 114] 8.772 % (456 backtracks, 1358 nodes, 0.027 seconds)
Optimality gap: [105, 114] 7.895 % (456 backtracks, 1405 nodes, 0.028 seconds)
Optimality gap: [106, 114] 7.018 % (468 backtracks, 1448 nodes, 0.028 seconds)
Optimality gap: [107, 114] 6.140 % (468 backtracks, 1467 nodes, 0.029 seconds)
Optimality gap: [108, 114] 5.263 % (468 backtracks, 1499 nodes, 0.029 seconds)
Optimality gap: [110, 114] 3.509 % (491 backtracks, 1581 nodes, 0.030 seconds)
Optimality gap: [111, 114] 2.632 % (502 backtracks, 1635 nodes, 0.030 seconds)
Optimality gap: [112, 114] 1.754 % (502 backtracks, 1652 nodes, 0.030 seconds)
Optimality gap: [113, 114] 0.877 % (502 backtracks, 1819 nodes, 0.031 seconds)
Optimality gap: [114, 114] 0.000 % (502 backtracks, 1942 nodes, 0.032 seconds)
HBFS open list restarts: 0.000 % and reuse: 16.119 % of 335
Node redundancy during HBFS: 49.691
Optimum: 114 in 502 backtracks and 1942 nodes ( 38 removals by DEE) and 0.032↪
↪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.
Reverse original DAC dual bound: 51 (+19.608%)
Reverse original DAC dual bound: 53 (+3.774%)
Reverse original DAC dual bound: 54 (+1.852%)
Cost function decomposition time : 0.000 seconds.
Preprocessing time: 0.011 seconds.

```

(continues on next page)



(continued from previous page)

```

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: [54, 164] 67.073%
Get root cluster C44 with max. size: 20
Get root cluster C27 with max. size: 14
Tree decomposition width : 19
Tree decomposition height : 43
Number of clusters : 47
Tree decomposition time: 0.003 seconds.
--- Solving cluster subtree 5 ...
New solution: 0 (0 backtracks, 0 nodes, depth 2, 0.014 seconds)
--- done cost = [0,0] (0 backtracks, 0 nodes, depth 2)

--- Solving cluster subtree 6 ...
New solution: 0 (0 backtracks, 0 nodes, depth 2, 0.014 seconds)
--- done cost = [0,0] (0 backtracks, 0 nodes, depth 2)

--- Solving cluster subtree 7 ...

...

--- Solving cluster subtree 44 ...
New solution: 53 (493 backtracks, 921 nodes, depth 7, 0.026 seconds)
New solution: 52 (504 backtracks, 938 nodes, depth 7, 0.027 seconds)
New solution: 48 (515 backtracks, 973 nodes, depth 22, 0.027 seconds)
--- done cost = [48,48] (636 backtracks, 1160 nodes, depth 2)

--- Solving cluster subtree 46 ...
New solution: 114 (636 backtracks, 1160 nodes, depth 2, 0.031 seconds)
--- done cost = [114,114] (636 backtracks, 1160 nodes, depth 2)

Optimum: 114 in 636 backtracks and 1160 nodes ( 80 removals by DEE) and 0.031.
↪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 : 0.007695 seconds.
Preprocessing time: 0.031135 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: [2, 34347] 99.994%
Get root cluster C3 with max. size: 5
Tree decomposition width : 59
Tree decomposition height : 233
Number of clusters : 239
Tree decomposition time: 0.026 seconds.

```

(continues on next page)

(continued from previous page)

```

--- Solving cluster subtree 0 ...
New solution: 0 (0 backtracks, 0 nodes, depth 2, 0.058 seconds)
--- done cost = [0,0] (0 backtracks, 0 nodes, depth 2)

--- Solving cluster subtree 1 ...
New solution: 0 (0 backtracks, 0 nodes, depth 2, 0.059 seconds)
--- 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, 5.361 seconds)
New solution: 21251 (26991 backtracks, 48883 nodes, depth 4, 5.367 seconds)
--- done cost = [21251,21251] (26992 backtracks, 48883 nodes, depth 2)

--- Solving cluster subtree 238 ...
New solution: 21253 (26992 backtracks, 48883 nodes, depth 2, 5.367 seconds)
--- done cost = [21253,21253] (26992 backtracks, 48883 nodes, depth 2)

Optimum: 21253 in 26992 backtracks and 48883 nodes ( 0 removals by DEE) and 5.377
↪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.
Reverse original DAC dual bound: 5086 (+59.339%)
Reverse original DAC dual bound: 5089 (+0.059%)
Cost function decomposition time : 0.014 seconds.
Preprocessing time: 0.137 seconds.
233 unassigned variables, 666 values in all current domains (med. size:2, max
↪size:4) and 1969 non-unary cost functions (med. arity:2, med. degree:16)
Initial lower and upper bounds: [5089, 34354] 85.187%
Tree decomposition time: 0.018 seconds.
Problem decomposition in 89 clusters with size distribution: min: 4 median: 11
↪mean: 11.831 max: 23
New solution: 26266 (0 backtracks, 55 nodes, depth 56, 0.167 seconds)
New solution: 26264 in 0.182 seconds.
New solution: 26263 in 0.187 seconds.
New solution: 26262 in 0.188 seconds.
New solution: 26261 in 0.207 seconds.
New solution: 26260 in 0.221 seconds.
New solution: 25261 in 0.228 seconds.
New solution: 25257 in 0.239 seconds.
New solution: 24262 in 0.243 seconds.
New solution: 23262 in 0.258 seconds.

```

(continues on next page)

(continued from previous page)

```

New solution: 23261 in 0.266 seconds.
New solution: 23259 in 0.270 seconds.
New solution: 22260 in 0.307 seconds.
New solution: 21262 in 0.328 seconds.
New solution: 21261 in 0.334 seconds.
New solution: 21260 in 0.347 seconds.
New solution: 21258 in 0.452 seconds.
New solution: 21257 in 0.554 seconds.
New solution: 21256 in 0.625 seconds.
New solution: 21255 in 0.640 seconds.
New solution: 21254 in 0.649 seconds.
New solution: 21253 in 1.060 seconds.

```

- 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.
Reverse original DAC dual bound: 20 (+10.000%)
Cost function decomposition time : 0.000 seconds.
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: 29 (0 backtracks, 6 nodes, depth 7, 0.001 seconds)
Problem decomposition in 7 clusters with size distribution: min: 11 median: 15
↳mean: 15.143 max: 17
***** Restart 1 with 1 discrepancies and UB=29 ***** (6 nodes)
New solution: 28 (0 backtracks, 6 nodes, depth 1, 0.002 seconds)
New solution: 27 (12 backtracks, 52 nodes, depth 2, 0.004 seconds)
***** Restart 2 with 2 discrepancies and UB=27 ***** (101 nodes)
***** Restart 3 with 4 discrepancies and UB=27 ***** (217 nodes)
***** Restart 4 with 8 discrepancies and UB=27 ***** (376 nodes)
***** Restart 5 with 16 discrepancies and UB=27 ***** (724 nodes)
Optimum: 27 in 446 backtracks and 1002 nodes ( 3035 removals by DEE) and 0.034
↳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.
Reverse original DAC dual bound: 20 (+10.000%)
Cost function decomposition time : 0.000 seconds.
Preprocessing time: 0.001 seconds.
24 unassigned variables, 116 values in all current domains (med. size:5, max

```

(continues on next page)



(continued from previous page)

```

↪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: 29 (0 backtracks, 7 nodes, depth 8, 0.002 seconds)
New solution: 28 in 0.003 seconds.
New solution: 27 in 0.003 seconds.
Optimum: 27 in 0 backtracks and 7 nodes ( 32 removals by DEE) and 0.063 seconds.
Total CPU time = 0.267 seconds.
Solving real-time = 0.066 seconds (not including reading and preprocessing time).
end.

```

- Download file `example.order`. Solve a WCSP using BTB-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.
Reverse original DAC dual bound: 19 (+5.263%)
Reverse original DAC dual bound: 21 (+9.524%)
Cost function decomposition time : 0.000 seconds.
Preprocessing time: 0.001 seconds.
24 unassigned variables, 118 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%
Get root cluster C14 with max. size: 9
Tree decomposition width : 8
Tree decomposition height : 16
Number of clusters : 18
Tree decomposition time: 0.001 seconds.
New solution: 28 (19 backtracks, 30 nodes, depth 3, 0.002 seconds)
Optimality gap: [23, 28] 17.857 % (26 backtracks, 45 nodes, 0.003 seconds)
New solution: 27 (60 backtracks, 127 nodes, depth 3, 0.004 seconds)
Optimality gap: [23, 27] 14.815 % (60 backtracks, 127 nodes, 0.004 seconds)
Optimality gap: [24, 27] 11.111 % (102 backtracks, 214 nodes, 0.006 seconds)
Optimality gap: [25, 27] 7.407 % (148 backtracks, 348 nodes, 0.009 seconds)
Optimality gap: [26, 27] 3.704 % (157 backtracks, 397 nodes, 0.010 seconds)
Optimality gap: [27, 27] 0.000 % (157 backtracks, 410 nodes, 0.010 seconds)
HBFS open list restarts: 0.000 % and reuse: 13.333 % of 45
Node redundancy during HBFS: 25.366
Optimum: 27 in 157 backtracks and 410 nodes ( 238 removals by DEE) and 0.010
↪seconds.
end.

```

- Download file `example.cov`. Solve a WCSP using BTB-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.

```

(continues on next page)

(continued from previous page)

```
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.
Reverse original DAC dual bound: 19 (+10.526%)
Reverse original DAC dual bound: 20 (+5.000%)
Cost function decomposition time : 0.000 seconds.
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: [20, 64] 68.750%
Tree decomposition width : 16
Tree decomposition height : 24
Number of clusters : 9
Tree decomposition time: 0.001 seconds.
New solution: 28 (23 backtracks, 29 nodes, depth 3, 0.003 seconds)
Optimality gap: [21, 28] 25.000 % (61 backtracks, 124 nodes, 0.005 seconds)
Optimality gap: [22, 28] 21.429 % (154 backtracks, 321 nodes, 0.009 seconds)
New solution: 27 (390 backtracks, 814 nodes, depth 3, 0.018 seconds)
Optimality gap: [23, 27] 14.815 % (390 backtracks, 814 nodes, 0.018 seconds)
Optimality gap: [24, 27] 11.111 % (417 backtracks, 884 nodes, 0.020 seconds)
Optimality gap: [25, 27] 7.407 % (435 backtracks, 966 nodes, 0.021 seconds)
Optimality gap: [26, 27] 3.704 % (450 backtracks, 1044 nodes, 0.021 seconds)
Optimality gap: [27, 27] 0.000 % (450 backtracks, 1072 nodes, 0.022 seconds)
HBFS open list restarts: 0.000 % and reuse: 33.333 % of 21
Node redundancy during HBFS: 17.724
Optimum: 27 in 450 backtracks and 1072 nodes ( 248 removals by DEE) and 0.022.
↳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.evid
No evidence file.
Reverse original DAC dual bound: 287536650 energy: 240.632 (+7.642%)
Generic variable elimination of degree 4
Maximum degree of generic variable elimination: 4
Cost function decomposition time : 0.004 seconds.
Preprocessing time: 0.086 seconds.
235 unassigned variables, 523 values in all current domains (med. size:2, max.
↳size:4) and 433 non-unary cost functions (med. arity:2, med. degree:6)
Initial lower and upper bounds: [560511104, 13246577993] 95.769%
Get root cluster C1034 with max. size: 30
Tree decomposition width : 230
Tree decomposition height : 233
Number of clusters : 887
```

(continues on next page)

(continued from previous page)

```

Tree decomposition time: 0.021 seconds.
New solution: 839721541 energy: 295.850 prob: 3.265e-129 (70 backtracks, 133 nodes,
↳ depth 3, 0.118 seconds)
New solution: 837627377 energy: 295.641 prob: 4.025e-129 (363 backtracks, 808 nodes,
↳ depth 3, 0.148 seconds)
New solution: 815835377 energy: 293.462 prob: 3.558e-128 (590 backtracks, 1287
↳ nodes, depth 3, 0.183 seconds)
New solution: 771837572 energy: 289.062 prob: 2.898e-126 (1153 backtracks, 2462
↳ nodes, depth 3, 0.266 seconds)
New solution: 749865327 energy: 286.865 prob: 2.608e-125 (2257 backtracks, 4726
↳ nodes, depth 3, 0.452 seconds)
New solution: 743190828 energy: 286.197 prob: 5.083e-125 (2962 backtracks, 6589
↳ nodes, depth 3, 0.529 seconds)
New solution: 718546229 energy: 283.733 prob: 5.976e-124 (5872 backtracks, 12559
↳ nodes, depth 3, 1.012 seconds)
New solution: 712604395 energy: 283.139 prob: 1.083e-123 (9785 backtracks, 20539
↳ nodes, depth 3, 1.667 seconds)
New solution: 711184935 energy: 282.997 prob: 1.248e-123 (10606 backtracks, 22820
↳ nodes, depth 3, 1.778 seconds)
HBFS open list restarts: 0.000 % and reuse: 74.124 % of 3424
Node redundancy during HBFS: 20.718
Optimum: 711184935 energy: 282.997 prob: 1.248e-123 in 165886 backtracks and 418438
↳ nodes ( 566654 removals by DEE) and 29.071 seconds.
end.

```

- Download another MRF file `GeomSurf-7-gm256.uai.xz`. Solve it using Virtual Arc Consistency (VAC) in pre-processing [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.evid
No evidence file.
Reverse original DAC dual bound: 5874975174 energy: 1073.679 (+0.003%)
Cost function decomposition time : 0.005 seconds.
Number of VAC iterations: 768
Number of times is VAC: 360 Number of times isvac and itThreshold > 1: 351
Preprocessing time: 2.085 seconds.
729 unassigned variables, 4818 values in all current domains (med. size:7, max
↳ size:7) and 3132 non-unary cost functions (med. arity:2, med. degree:6)
Initial lower and upper bounds: [5899313907, 111615203929] 94.715%
New solution: 5926230227 energy: 1078.805 prob: 3.027e-469 (0 backtracks, 1 nodes,
↳ depth 3, 2.088 seconds)
New solution: 5922482579 energy: 1078.430 prob: 4.404e-469 (0 backtracks, 1 nodes,
↳ depth 2, 2.092 seconds)
Optimality gap: [5922482579, 5922482579] 0.000 % (0 backtracks, 1 nodes, 2.093
↳ seconds)
Number of VAC iterations: 928
Number of times is VAC: 520 Number of times isvac and itThreshold > 1: 507
Node redundancy during HBFS: 0.000

```

(continues on next page)

(continued from previous page)

```
Optimum: 5922482579 energy: 1078.430 prob: 4.404e-469 in 0 backtracks and 1 nodes (↪
↪481 removals by DEE) and 2.093 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.evid
No evidence file.
Reverse original DAC dual bound: 103967050722 energy: -12488.257 (+0.000%)
Cost function decomposition time : 0.001 seconds.
Number of VAC iterations: 3936
Number of times is VAC: 316 Number of times isvac and itThreshold > 1: 310
Threshold: 2326140792 NbAssignedVar: 0 Ratio: 0.00000000 SumOfDomainSize: 1783
Threshold: 2320180664 NbAssignedVar: 0 Ratio: 0.00000000 SumOfDomainSize: 1775
Threshold: 21288438 NbAssignedVar: 16 Ratio: 0.00000000 SumOfDomainSize: 84
Threshold: 11824293 NbAssignedVar: 17 Ratio: 0.00000000 SumOfDomainSize: 65
Threshold: 8188677 NbAssignedVar: 18 Ratio: 0.00000001 SumOfDomainSize: 52
Threshold: 6858840 NbAssignedVar: 19 Ratio: 0.00000001 SumOfDomainSize: 46
Threshold: 6060563 NbAssignedVar: 20 Ratio: 0.00000001 SumOfDomainSize: 42
Threshold: 5504852 NbAssignedVar: 20 Ratio: 0.00000001 SumOfDomainSize: 42
Threshold: 3972364 NbAssignedVar: 21 Ratio: 0.00000001 SumOfDomainSize: 37
Threshold: 3655432 NbAssignedVar: 21 Ratio: 0.00000002 SumOfDomainSize: 37
Threshold: 3067826 NbAssignedVar: 21 Ratio: 0.00000002 SumOfDomainSize: 37
Threshold: 2174446 NbAssignedVar: 22 Ratio: 0.00000003 SumOfDomainSize: 35
Threshold: 1641827 NbAssignedVar: 22 Ratio: 0.00000004 SumOfDomainSize: 35
Threshold: 1374852 NbAssignedVar: 22 Ratio: 0.00000004 SumOfDomainSize: 35
Threshold: 206113 NbAssignedVar: 23 Ratio: 0.00000030 SumOfDomainSize: 31
Threshold: 103056 NbAssignedVar: 24 Ratio: 0.00000063 SumOfDomainSize: 29
Threshold: 51528 NbAssignedVar: 25 Ratio: 0.0000131 SumOfDomainSize: 27
Threshold: 25764 NbAssignedVar: 25 Ratio: 0.0000262 SumOfDomainSize: 26
Threshold: 12882 NbAssignedVar: 25 Ratio: 0.0000525 SumOfDomainSize: 26
Threshold: 6441 NbAssignedVar: 25 Ratio: 0.0001049 SumOfDomainSize: 26
Threshold: 3220 NbAssignedVar: 25 Ratio: 0.0002098 SumOfDomainSize: 26
Threshold: 1610 NbAssignedVar: 25 Ratio: 0.0004197 SumOfDomainSize: 26
Threshold: 805 NbAssignedVar: 25 Ratio: 0.0008393 SumOfDomainSize: 26
Threshold: 402 NbAssignedVar: 25 Ratio: 0.0016808 SumOfDomainSize: 26
Threshold: 201 NbAssignedVar: 25 Ratio: 0.0033616 SumOfDomainSize: 26
Threshold: 100 NbAssignedVar: 25 Ratio: 0.0067568 SumOfDomainSize: 26
Threshold: 50 NbAssignedVar: 25 Ratio: 0.0135135 SumOfDomainSize: 26
Threshold: 25 NbAssignedVar: 25 Ratio: 0.0270270 SumOfDomainSize: 26
Threshold: 12 NbAssignedVar: 25 Ratio: 0.0563063 SumOfDomainSize: 26
Threshold: 6 NbAssignedVar: 25 Ratio: 0.1126126 SumOfDomainSize: 26
Threshold: 3 NbAssignedVar: 25 Ratio: 0.2252252 SumOfDomainSize: 26
Threshold: 1 NbAssignedVar: 25 Ratio: 0.6756757 SumOfDomainSize: 26
RASPS/VAC threshold: 201
Preprocessing time: 42.878 seconds.
37 unassigned variables, 3387 values in all current domains (med. size:38, max↪
```

(continues on next page)

(continued from previous page)

```

↪size:322) and 627 non-unary cost functions (med. arity:2, med. degree:35)
Initial lower and upper bounds: [103967050722, 239074058140] 56.513%
New solution: 104206588507 energy: -12464.303 prob: inf (0 backtracks, 4 nodes, ↪
↪depth 7, 42.886 seconds)
RASPS done in preprocessing at threshold 201 (backtrack: 2 nodes: 6)
New solution: 104174015034 energy: -12467.560 prob: inf (2 backtracks, 9 nodes, ↪
↪depth 5, 42.910 seconds)
Optimality gap: [104174015034, 104174015034] 0.000 % (4 backtracks, 11 nodes, 42.
↪910 seconds)
Number of VAC iterations: 4279
Number of times is VAC: 617 Number of times isvac and itThreshold > 1: 606
Node redundancy during HBFS: 0.000
Optimum: 104174015034 energy: -12467.560 prob: inf in 4 backtracks and 11 nodes ( ↪
↪921 removals by DEE) and 42.911 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.
Reverse original DAC dual bound: 91 (+86.813%)
Reverse original DAC dual bound: 92 (+1.087%)
Cost function decomposition time : 0.000 seconds.
Preprocessing time: 0.061 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, 0.066 seconds)
New solution: 188 (45 backtracks, 143 nodes, depth 37, 0.103 seconds)
New solution: 187 (155 backtracks, 473 nodes, depth 47, 0.144 seconds)
New solution: 186 (273 backtracks, 802 nodes, depth 37, 0.170 seconds)
New solution: 185 (1870 backtracks, 4112 nodes, depth 63, 0.279 seconds)
New solution: 184 (19563 backtracks, 41433 nodes, depth 19, 1.430 seconds)
New solution: 183 (207409 backtracks, 459653 nodes, depth 12, 12.669 seconds)
Node redundancy during HBFS: 25.824
Optimum: 183 in 268922 backtracks and 725087 nodes ( 8484 removals by DEE) and 20.
↪814 seconds.
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.
Reverse original DAC dual bound: 48 (+2.083%)
Cost function decomposition time : 0.000 seconds.
Preprocessing time: 0.008 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)

```

(continues on next page)

(continued from previous page)

```
Initial lower and upper bounds: [48, 1000] 95.200%
Optimality gap: [49, 1000] 95.100 % (17 backtracks, 41 nodes, 0.024 seconds)
Optimality gap: [58, 1000] 94.200 % (355 backtracks, 812 nodes, 0.175 seconds)
Optimality gap: [72, 1000] 92.800 % (575 backtracks, 1309 nodes, 0.265 seconds)
Optimality gap: [1000, 1000] 0.000 % (575 backtracks, 1318 nodes, 0.266 seconds)
Number of solutions      : = 576
Time                    : 0.266 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.
Reverse original DAC dual bound: 48 (+2.083%)
Cost function decomposition time : 0.000 seconds.
Preprocessing time: 0.018 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, 0.023 seconds)
New solution: 48 (2 backtracks, 11 nodes, depth 3, 0.024 seconds)
Node redundancy during HBFS: 18.182
Optimum: 48 in 2 backtracks and 11 nodes ( 0 removals by DEE) and 0.024 seconds.
+++++ Search for solution 2 +++++
New solution: 67 (2 backtracks, 15 nodes, depth 7, 0.029 seconds)
New solution: 65 (3 backtracks, 20 nodes, depth 9, 0.032 seconds)
Optimality gap: [49, 65] 24.615 % (9 backtracks, 26 nodes, 0.035 seconds)
New solution: 58 (9 backtracks, 31 nodes, depth 4, 0.038 seconds)
Optimality gap: [50, 58] 13.793 % (10 backtracks, 32 nodes, 0.038 seconds)
New solution: 52 (16 backtracks, 53 nodes, depth 4, 0.046 seconds)
New solution: 51 (17 backtracks, 54 nodes, depth 3, 0.046 seconds)
Optimality gap: [51, 51] 0.000 % (17 backtracks, 54 nodes, 0.046 seconds)
Node redundancy during HBFS: 25.926
Optimum: 51 in 17 backtracks and 54 nodes ( 0 removals by DEE) and 0.046 seconds.
+++++ Search for solution 3 +++++
+++++ predictive bounding: 57.000
Optimality gap: [51, 57] 10.526 % (36 backtracks, 92 nodes, 0.074 seconds)
Optimality gap: [57, 57] 0.000 % (37 backtracks, 98 nodes, 0.078 seconds)
Node redundancy during HBFS: 18.367
No solution in 37 backtracks and 98 nodes ( 1 removals by DEE) and 0.078 seconds.
+++++ Search for solution 3 +++++
New solution: 64 (37 backtracks, 102 nodes, depth 7, 0.083 seconds)
New solution: 57 (38 backtracks, 105 nodes, depth 7, 0.085 seconds)
Optimality gap: [57, 57] 0.000 % (60 backtracks, 145 nodes, 0.111 seconds)
Node redundancy during HBFS: 12.414
Optimum: 57 in 60 backtracks and 145 nodes ( 2 removals by DEE) and 0.111 seconds.
+++++ Search for solution 4 +++++
```

(continues on next page)



(continued from previous page)

```

+++++ predictive bounding: 69.000
New solution: 65 (66 backtracks, 159 nodes, depth 5, 0.121 seconds)
New solution: 61 (80 backtracks, 188 nodes, depth 6, 0.137 seconds)
New solution: 58 (96 backtracks, 219 nodes, depth 5, 0.156 seconds)
Optimality gap: [58, 58] 0.000 % (110 backtracks, 245 nodes, 0.172 seconds)
Node redundancy during HBFS: 7.347
Optimum: 58 in 110 backtracks and 245 nodes ( 9 removals by DEE) and 0.173 seconds.
+++++ Search for solution 5 +++++
+++++ predictive bounding: 70.000
New solution: 62 (116 backtracks, 260 nodes, depth 6, 0.182 seconds)
New solution: 58 (146 backtracks, 320 nodes, depth 6, 0.221 seconds)
Node redundancy during HBFS: 5.625
Optimum: 58 in 146 backtracks and 320 nodes ( 11 removals by DEE) and 0.221 seconds.
+++++ Search for solution 6 +++++
+++++ predictive bounding: 70.000
New solution: 65 (176 backtracks, 382 nodes, depth 5, 0.253 seconds)
New solution: 62 (233 backtracks, 495 nodes, depth 4, 0.323 seconds)
Optimality gap: [62, 62] 0.000 % (238 backtracks, 504 nodes, 0.331 seconds)
Node redundancy during HBFS: 3.571
Optimum: 62 in 238 backtracks and 504 nodes ( 13 removals by DEE) and 0.331 seconds.
+++++ Search for solution 7 +++++
+++++ predictive bounding: 74.000
c 2097152 Bytes allocated for long long stack.
New solution: 68 (246 backtracks, 523 nodes, depth 6, 0.345 seconds)
New solution: 65 (254 backtracks, 539 nodes, depth 6, 0.354 seconds)
Optimality gap: [49, 65] 24.615 % (304 backtracks, 636 nodes, 0.416 seconds)
Optimality gap: [58, 65] 10.769 % (324 backtracks, 682 nodes, 0.442 seconds)
Optimality gap: [65, 65] 0.000 % (325 backtracks, 692 nodes, 0.444 seconds)
Node redundancy during HBFS: 4.624
Optimum: 65 in 325 backtracks and 692 nodes ( 22 removals by DEE) and 0.444 seconds.
+++++ Search for solution 8 +++++
+++++ predictive bounding: 77.000
Optimality gap: [49, 77] 36.364 % (380 backtracks, 835 nodes, 0.524 seconds)
Optimality gap: [50, 77] 35.065 % (389 backtracks, 874 nodes, 0.538 seconds)
Optimality gap: [51, 77] 33.766 % (392 backtracks, 887 nodes, 0.542 seconds)
Optimality gap: [52, 77] 32.468 % (397 backtracks, 905 nodes, 0.547 seconds)
New solution: 72 (403 backtracks, 923 nodes, depth 4, 0.555 seconds)
Optimality gap: [53, 72] 26.389 % (407 backtracks, 930 nodes, 0.559 seconds)
Optimality gap: [54, 72] 25.000 % (413 backtracks, 960 nodes, 0.567 seconds)
Optimality gap: [57, 72] 20.833 % (419 backtracks, 977 nodes, 0.572 seconds)
New solution: 71 (425 backtracks, 1004 nodes, depth 6, 0.582 seconds)
New solution: 70 (427 backtracks, 1009 nodes, depth 7, 0.584 seconds)
New solution: 65 (435 backtracks, 1022 nodes, depth 4, 0.591 seconds)
Node redundancy during HBFS: 13.796
Optimum: 65 in 435 backtracks and 1022 nodes ( 34 removals by DEE) and 0.591
↪seconds.
+++++ Search for solution 9 +++++
+++++ predictive bounding: 77.000
New solution: 73 (452 backtracks, 1059 nodes, depth 6, 0.614 seconds)
New solution: 68 (472 backtracks, 1101 nodes, depth 8, 0.637 seconds)
New solution: 66 (488 backtracks, 1130 nodes, depth 5, 0.652 seconds)
Optimality gap: [66, 66] 0.000 % (537 backtracks, 1226 nodes, 0.713 seconds)

```

(continues on next page)

(continued from previous page)

```

Node redundancy during HBFS: 11.501
Optimum: 66 in 537 backtracks and 1226 nodes ( 51 removals by DEE) and 0.713↵
↵seconds.
+++++++ Search for solution 10 ++++++
+++++++ predictive bounding: 74.000
New solution: 70 (546 backtracks, 1248 nodes, depth 7, 0.729 seconds)
New solution: 68 (560 backtracks, 1275 nodes, depth 6, 0.743 seconds)
Optimality gap: [68, 68] 0.000 % (637 backtracks, 1426 nodes, 0.835 seconds)
Node redundancy during HBFS: 9.888
Optimum: 68 in 637 backtracks and 1426 nodes ( 63 removals by DEE) and 0.835↵
↵seconds.
+++++++ Search for solution 11 ++++++
+++++++ predictive bounding: 76.000
New solution: 72 (638 backtracks, 1431 nodes, depth 6, 0.842 seconds)
New solution: 71 (648 backtracks, 1453 nodes, depth 8, 0.855 seconds)
New solution: 68 (703 backtracks, 1560 nodes, depth 5, 0.918 seconds)
Node redundancy during HBFS: 9.038
Optimum: 68 in 703 backtracks and 1560 nodes ( 71 removals by DEE) and 0.918↵
↵seconds.
+++++++ Search for solution 12 ++++++
+++++++ predictive bounding: 76.000
New solution: 74 (740 backtracks, 1636 nodes, depth 5, 0.966 seconds)
New solution: 72 (770 backtracks, 1696 nodes, depth 5, 1.002 seconds)
New solution: 71 (781 backtracks, 1718 nodes, depth 5, 1.016 seconds)
Optimality gap: [71, 71] 0.000 % (835 backtracks, 1824 nodes, 1.077 seconds)
Node redundancy during HBFS: 7.730
Optimum: 71 in 835 backtracks and 1824 nodes ( 81 removals by DEE) and 1.077↵
↵seconds.
+++++++ Search for solution 13 ++++++
+++++++ predictive bounding: 77.000
c 4194304 Bytes allocated for long long stack.
New solution: 76 (881 backtracks, 1919 nodes, depth 6, 1.144 seconds)
New solution: 72 (950 backtracks, 2056 nodes, depth 5, 1.220 seconds)
Optimality gap: [72, 72] 0.000 % (978 backtracks, 2110 nodes, 1.258 seconds)
Node redundancy during HBFS: 6.682
Optimum: 72 in 978 backtracks and 2110 nodes ( 88 removals by DEE) and 1.258↵
↵seconds.
+++++++ Search for solution 14 ++++++
+++++++ predictive bounding: 78.000
New solution: 77 (1030 backtracks, 2217 nodes, depth 6, 1.321 seconds)
New solution: 76 (1050 backtracks, 2256 nodes, depth 5, 1.345 seconds)
New solution: 74 (1077 backtracks, 2312 nodes, depth 7, 1.384 seconds)
Optimality gap: [74, 74] 0.000 % (1109 backtracks, 2372 nodes, 1.418 seconds)
Node redundancy during HBFS: 5.944
Optimum: 74 in 1109 backtracks and 2372 nodes ( 103 removals by DEE) and 1.418↵
↵seconds.
+++++++ Search for solution 15 ++++++
+++++++ predictive bounding: 80.000
New solution: 79 (1109 backtracks, 2376 nodes, depth 7, 1.427 seconds)
New solution: 78 (1163 backtracks, 2483 nodes, depth 6, 1.483 seconds)
New solution: 76 (1204 backtracks, 2564 nodes, depth 5, 1.533 seconds)
Optimality gap: [76, 76] 0.000 % (1248 backtracks, 2650 nodes, 1.588 seconds)

```

(continues on next page)



(continued from previous page)

```

Node redundancy during HBFS: 5.321
Optimum: 76 in 1248 backtracks and 2650 nodes ( 116 removals by DEE) and 1.588↵
↵seconds.
+++++++ Search for solution 16 ++++++++
+++++++ predictive bounding: 82.000
New solution: 80 (1266 backtracks, 2688 nodes, depth 5, 1.615 seconds)
New solution: 79 (1283 backtracks, 2722 nodes, depth 5, 1.636 seconds)
New solution: 78 (1328 backtracks, 2813 nodes, depth 6, 1.691 seconds)
Optimality gap: [78, 78] 0.000 % (1371 backtracks, 2896 nodes, 1.744 seconds)
Node redundancy during HBFS: 4.869
Optimum: 78 in 1371 backtracks and 2896 nodes ( 128 removals by DEE) and 1.744↵
↵seconds.
+++++++ Search for solution 17 ++++++++
+++++++ predictive bounding: 84.000
New solution: 80 (1371 backtracks, 2900 nodes, depth 7, 1.753 seconds)
New solution: 79 (1413 backtracks, 2983 nodes, depth 6, 1.798 seconds)
Optimality gap: [79, 79] 0.000 % (1482 backtracks, 3118 nodes, 1.883 seconds)
Node redundancy during HBFS: 4.522
Optimum: 79 in 1482 backtracks and 3118 nodes ( 135 removals by DEE) and 1.883↵
↵seconds.
+++++++ Search for solution 18 ++++++++
+++++++ predictive bounding: 85.000
New solution: 80 (1488 backtracks, 3133 nodes, depth 6, 1.898 seconds)
Optimality gap: [59, 80] 26.250 % (1610 backtracks, 3374 nodes, 2.039 seconds)
Optimality gap: [80, 80] 0.000 % (1621 backtracks, 3404 nodes, 2.057 seconds)
Node redundancy during HBFS: 4.377
Optimum: 80 in 1621 backtracks and 3404 nodes ( 136 removals by DEE) and 2.057↵
↵seconds.
+++++++ Search for solution 19 ++++++++
+++++++ predictive bounding: 84.000
New solution: 80 (1665 backtracks, 3496 nodes, depth 5, 2.122 seconds)
Node redundancy during HBFS: 4.319
Optimum: 80 in 1665 backtracks and 3496 nodes ( 138 removals by DEE) and 2.122↵
↵seconds.
+++++++ Search for solution 20 ++++++++
+++++++ predictive bounding: 84.000
Optimality gap: [49, 84] 41.667 % (1705 backtracks, 3586 nodes, 2.190 seconds)
Optimality gap: [51, 84] 39.286 % (1708 backtracks, 3597 nodes, 2.198 seconds)
Optimality gap: [52, 84] 38.095 % (1716 backtracks, 3640 nodes, 2.219 seconds)
Optimality gap: [53, 84] 36.905 % (1717 backtracks, 3650 nodes, 2.222 seconds)
Optimality gap: [54, 84] 35.714 % (1724 backtracks, 3687 nodes, 2.237 seconds)
Optimality gap: [55, 84] 34.524 % (1725 backtracks, 3694 nodes, 2.242 seconds)
Optimality gap: [56, 84] 33.333 % (1728 backtracks, 3715 nodes, 2.248 seconds)
Optimality gap: [57, 84] 32.143 % (1733 backtracks, 3751 nodes, 2.260 seconds)
Optimality gap: [58, 84] 30.952 % (1734 backtracks, 3760 nodes, 2.263 seconds)
Optimality gap: [60, 84] 28.571 % (1745 backtracks, 3816 nodes, 2.288 seconds)
Optimality gap: [61, 84] 27.381 % (1746 backtracks, 3824 nodes, 2.291 seconds)
Optimality gap: [62, 84] 26.190 % (1747 backtracks, 3834 nodes, 2.293 seconds)
Optimality gap: [63, 84] 25.000 % (1752 backtracks, 3866 nodes, 2.305 seconds)
Optimality gap: [64, 84] 23.810 % (1753 backtracks, 3880 nodes, 2.308 seconds)
Optimality gap: [65, 84] 22.619 % (1758 backtracks, 3908 nodes, 2.320 seconds)
Optimality gap: [66, 84] 21.429 % (1761 backtracks, 3930 nodes, 2.324 seconds)

```

(continues on next page)

(continued from previous page)

```

Optimality gap: [67, 84] 20.238 % (1765 backtracks, 3961 nodes, 2.333 seconds)
Optimality gap: [68, 84] 19.048 % (1767 backtracks, 3987 nodes, 2.336 seconds)
Optimality gap: [71, 84] 15.476 % (1769 backtracks, 3999 nodes, 2.339 seconds)
Optimality gap: [84, 84] 0.000 % (1770 backtracks, 4006 nodes, 2.341 seconds)
Node redundancy during HBFS: 11.258
No solution in 1770 backtracks and 4006 nodes ( 141 removals by DEE) and 2.341
seconds.
+++++ Search for solution 20 +++++
Optimality gap: [1000, 1000] 0.000 % (1863 backtracks, 4192 nodes, 2.479 seconds)
Node redundancy during HBFS: 10.759
No solution in 1863 backtracks and 4192 nodes ( 146 removals by DEE) and 2.479
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 : 1.3e-05 seconds.
Preprocessing time: 0.001466 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%
Get root cluster C1 with max. size: 5
Tree decomposition width : 5
Tree decomposition height : 20
Number of clusters : 29
Tree decomposition time: 0.001 seconds.
Number of solutions : = 411110802705928379432960
Number of #goods : 3993
Number of used #goods : 17190
Size of sep : 4
Time : 0.052 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: 0.001419 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%

```

(continues on next page)

(continued from previous page)

```

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.

Get root cluster C22 with max. size: 5
Get root cluster C25 with max. size: 2
Get root cluster C30 with max. size: 2
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 : <= 17199267840000000000000000
Number of solutions : ~= 48000000000000000000000000
Number of #goods : 468
Number of used #goods : 4788
Size of sep : 3
Time : 0.011 seconds
... in 3738 backtracks and 7476 nodes
end.

```

## 10.6 Potential issues

Toulbar2 implements tree search methods using a recursive procedure. It may overcome the current stack memory on very-large problems. On Linux, you can control stacksize limit (in bash, `'ulimit -s unlimited'`).

If your problem is huge, you may also consider recompiling toulbar2 with some specific cmake options depending on your needs: `BINARYWCSP` (restricted to binary cost functions only, without on-the-fly variable elimination nor VAC), `TERNARYWCSP` (restricted to binary and ternary cost functions only), `INT_COSTS` (32-bit integers for representing costs instead of 64-bit by default), `SHORT_COSTS` (16-bit integers for representing costs), `SHORT_VALUES` (16-bit integers for representing domain values instead of 32-bit by default).

## 10.7 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.

### 10.7.1 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)

### 10.7.2 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: -pwc: -hve:)

**-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, or f=3 to do it before and after PWC) (resp. bijective (f=2)) variables (default value is 1)

**-dec** preprocessing only: pairwise decomposition [Favier2011a] of cost functions with arity  $\geq 3$  into smaller arity cost functions (default option)

- card** preprocessing only: when reading opb or lp format, decomposes cardinality equality constraints into a network of binary and ternary constraints (option deactivated by default)
- 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 constraints (positive value) and/or directly in the cost function network up to a given absolute number (non-zero value except -1)
- mst** find a maximum spanning tree ordering for DAC
- S=[integer]** preprocessing only: performs singleton consistency restricted to the first variables following the DAC ordering (or all the variables if no parameter is given).
- 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)

### 10.7.3 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*.

**-lrbcd=["string"]**

initial upperbound found by LR-BCD local search solver. The string parameter is optional, using "5 -2 3" by default with the following meaning: *maxiter rank nbroundings*. (a negative rank means dividing the theoretical rank by the given absolute value)

**-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

## 10.7.4 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) (usage for parallel version: "mpirun -n [NbOfProcess] toulbar2 -hbfs problem.wcsp")

**-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)

**-sopen=[integer]**

number of visited open nodes before sorting the remaining open nodes based on weighted degree heuristics (double this limit for the next sorting) (see also option -q) (default value is 0, i.e., no sorting)

- 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
  - (-9) topological ordering (for Bayesian networks only)
- 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 BT-D-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)

## 10.7.5 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”. For doing large neighborhood search (LNS) instead of VNS, and bounded backtrack search instead of LDS, use e.g., “toulbar2 -vns -l: -bt=1000 -kmin=50 -kmax=50 -L=100 problem.wcsp”, for exploring 100 random neighborhoods of size 50 variables with at most 1000 backtracks per neighborhood search.

**-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)

## 10.7.6 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, except for the cfn format where it is equal to the decimal cost precision, e.g. 0.001 if 3 digits of precision)

**-P=[decimal]**

threshold cost value for VAC during the preprocessing phase only (default value is 1, except for the cfn format where it is equal to the decimal cost precision, e.g. 0.001 if 3 digits of precision)

**-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)

**-vaclin** VAC applied on linear constraints (must be combined with option -A)

**-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 ( $dee \geq 1$  and  $dee \leq 3$ )) 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)

## 10.7.7 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.

<b>-sortd</b>	sorts domains in preprocessing based on increasing unary costs (works only for binary WCSPs).
<b>-sortc</b>	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.
<b>-solr</b>	solution-based phase saving (reuse last found solution as preferred value assignment in the value ordering heuristic) (default option).
<b>-vacint</b>	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 WeightedC-SPConstraint), 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)

## 10.7.8 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)

## 10.7.9 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.

### 10.7.10 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 wcsv or cfn format in filename (or “problem.wcsv”/“problem.cfn” if no parameter is given) writes also the graphviz dot file and the degree distribution of the input problem (except if using a negative verbosity `-v=-1`)

**-z=[integer]**

1 or 3: saves original instance in 1-wcsv or 3-cfn format (1 by default), 2 or 4: saves after preprocessing in 2-wcsv 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`).

### 10.7.11 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/LP/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 (if greater than 1, default value is infinity) or floating-point precision (if smaller than 1, default value is 1e-9)

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.

### 10.7.12 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*.

## 10.8 Input formats

### 10.8.1 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)
- Weighted Partial Max-SAT (*.cnf/.wcnf* file extension)
- Quadratic Unconstrained Pseudo-Boolean Optimization (*.qpbo* file extension)
- Pseudo-Boolean Optimization (*.opb* and *.wbo* file extensions)
- Integer Linear Programming (*.lp* 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.

## 10.8.2 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 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  $[0, \text{domainsize}-1]$ .

```
{ "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,
             "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 inside 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:

- " $\geq$ ": with **parameters** array  $[cst, \delta]$  where  $cst$  and  $\delta$  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  $\delta$ .
- ">": similar with a strict inequality and semantics  $\max(0, y + 1 + cst - x \leq \delta?y + 1 + cst - x : upperbound)$
- " $\leq$ ": 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  $[cstx, csty, xmax, ymax, wxy]$  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  $\geq$  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
      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]]
      }
}
```

- "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]
      }
}
```

- "salldifdp" 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: salldifdp
      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 `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  $X_i$  and  $Y_i$  take the same value (not equal to zero))

– example:

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

(continues on next page)

(continued from previous page)

```

        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]
        }
      }

```

- "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  $\text{Sum}$  the result of the sum over the variables. For each comparator, the gap is defined according to the distance as follows:

- if  $<>$  is  $==$  :  $\text{gap} = \text{abs}(K - \text{Sum})$
- if  $<>$  is  $\leq$  :  $\text{gap} = \max(0, \text{Sum} - K)$
- if  $<>$  is  $<$  :  $\text{gap} = \max(0, \text{Sum} - K - 1)$
- if  $<>$  is  $!=$  :  $\text{gap} = 1$  if  $\text{Sum} \neq K$  and  $\text{gap} = 0$  otherwise
- if  $<>$  is  $>$  :  $\text{gap} = \max(0, K - \text{Sum} + 1)$ ;
- if  $<>$  is  $\geq$  :  $\text{gap} = \max(0, K - \text{Sum})$ ;

Warning: the decomposition of  $\text{wsum}$  and  $\text{wvarsum}$  may use an exponential size (sum of domain sizes).  $\text{list\_size1}$  and  $\text{list\_size2}$  must be equal in  $\text{ssame}$ .

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  $\geq$  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  $\leq$  operator. See below a simple example encoding  $v1+v2+v3+v4 \geq 1$ .

– 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  $\geq$  operator) where capacity and coefficients are positive or negative integers. Use negative numbers to express a generalized linear constraint with  $\leq$  operator. Variables can be names or indices in the whole problem. They must also belong to the scope. See below a simple example encoding  $(v1=1)+(v2=1)+(v3=1)+(v4=1) \geq 1$ .

– example:

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

(continues on next page)

(continued from previous page)

```

        weightedvalues: [[v1 1 1] [v2 1 1] [v3 1 1] [v4 1 1]]
    }
}

```

- "salldiffkp" with parameters array [metric: "hard" cost: inf] to express a hard alldifferent constraint (decomposes into knapsackv cost functions)

– example:

```

name: {scope: [v1 v2 v3 v4]
      type: salldiffkp
      params: {
        metric: hard
        cost: inf
      }
}

```

- "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
           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
}

```

(continues on next page)



(continued from previous page)

```

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 floating-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* Weighted Constraint Satisfaction Problem file format (wcsp)

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 wcp 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  $x \geq y + cstx \vee y \geq x + cstx$  with associated cost function  $(x \geq y + cstx \vee y \geq x + cstx) ? 0 : penalty$
- `sdisj cstx csty xinfy yinfy costx costy` to express a special disjunctive constraint with three implicit hard constraints  $x \leq xinfy$  and  $y \leq yinfy$  and  $x < xinfy \wedge y < yinfy \Rightarrow (x \geq y + cstx \vee y \geq x + cstx)$  and an additional cost function  $((x = xinfy) ? costx : 0) + ((y = yinfy) ? costy : 0)$
- 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
  - `salldiffkp hard UB` to express a hard alldifferent constraint (decomposes into knapsack cost functions)
  - `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
  - `salldifdp 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
- `wvamong 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  $\langle \rangle$  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  $\langle \rangle$  is `==` :  $\text{gap} = \text{abs}(K - \text{Sum})$
- \* if  $\langle \rangle$  is `<=` :  $\text{gap} = \max(0, \text{Sum} - K)$
- \* if  $\langle \rangle$  is `<` :  $\text{gap} = \max(0, \text{Sum} - K - 1)$
- \* if  $\langle \rangle$  is `!=` :  $\text{gap} = 1$  if  $\text{Sum} \neq K$  and  $\text{gap} = 0$  otherwise
- \* if  $\langle \rangle$  is `>` :  $\text{gap} = \max(0, K - \text{Sum} + 1)$ ;
- \* if  $\langle \rangle$  is `>=` :  $\text{gap} = \max(0, K - \text{Sum})$ ;

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  $x_0 * x_1$  in extension with variable domains  $\{0, 1\}$  (equivalent to a soft clause  $\neg x_0 \vee \neg x_1$ ):

```
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
```

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

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

- knapsackc constraint ( $2 * x0 + 3 * x1 + 4 * x2 + 5 * x3 \geq 10, x1 + x2 \leq 1$ ) 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 ( $2 * (x0 = 0) + 3 * (x1 = 1) + 4 * (x2 = 2) + 5 * (x3 = 0 \vee x3 = 1) \geq 10$ ) 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 ( $2 * (x0 = 0) + 3 * (x1 = 1) + 4 * (x2 = 2) + 5 * (x3 = 0 \vee x3 = 1) \geq 10$ ) 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 ( $3 \leq 2 * x1 * x2 + 3 * x1 * x4 + 4 * x2 * x4 < 5$ ) 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  $\max_{i=0}^3 ((x_i \neq i)?1000 : (4 - i))$ :

```
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 0 0 1 0 0 1 0 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
```

- `wvamong({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 wvamong 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 wesp 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 wesp format:

```

4-QUEENS 4 4 10 5
4 4 4 4
2 0 1 0 10
0 0 5
0 1 5
1 0 5
1 1 5
1 2 5
2 1 5
2 2 5
2 3 5
3 2 5
3 3 5
2 0 2 0 8
0 0 5
0 2 5
1 1 5
1 3 5
2 0 5
2 2 5
3 1 5
3 3 5
2 0 3 0 6
0 0 5
0 3 5
1 1 5
2 2 5
3 0 5
3 3 5
2 1 2 0 10
0 0 5
0 1 5
1 0 5
1 1 5
1 2 5
2 1 5
2 2 5
2 3 5
3 2 5
3 3 5
2 1 3 0 8
0 0 5
0 2 5
1 1 5
1 3 5
2 0 5
2 2 5
3 1 5
3 3 5
2 2 3 0 10
0 0 5
0 1 5
1 0 5

```

(continues on next page)



(continued from previous page)

```

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

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

```

2
0.436 0.564

4
0.128 0.872
0.920 0.080

6
0.210 0.333 0.457
0.811 0.000 0.189

```

- **Expressing evidence**

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

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

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

```

2
1 0
2 1

```

## Partial Weighted MaxSAT format

### Max-SAT input format (.cnf)

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

```

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

```

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

### Weighted Max-SAT input format (.wcnf)

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

Example of Weighted Max-SAT formula:

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

#### Partial Max-SAT input format (.wcnf)

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

Example of Partial Max-SAT formula:

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

#### Weighted Partial Max-SAT input format (.wcnf)

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

Example of Weighted Partial Max-SAT formula:

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

#### QPBO format (.qpbo)

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

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

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

Note that for two indices  $i \neq j$ , coefficient  $W_{ij} = W_{ji}$  (symmetric matrix) and it appears twice in the previous sum. It can be controlled 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  $10^{\text{precision}}$  (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`).

### WBO format (.wbo)

The WBO file format is used to express pseudo-Boolean optimization models with hard and soft constraints. It should contain no objective function. Instead a first line with the keyword **soft:** followed by a positive integer cost corresponding to a forbidden assignment and **;**. Each soft constraint starts with **[cost]** where **cost** is a positive integer representing the violation cost of the constraint. The precision is forced to be 0.

An example with 4 soft linear constraints and 2 hard linear constraints is:

```

soft: 8 ;
[2] +1 x1 >= 1 ;
[3] +1 x2 >= 1 ;
[4] +1 x3 >= 1 ;
[5] +1 x4 >= 1 ;
-1 x1 -1 x2 >= -1 ;
-1 x3 -1 x4 >= -1 ;

```

### CPLEX format (.lp)

This textual format expresses a linear program with a linear or quadratic objective. Currently, quadratic constraints cannot be read and real variables are discretized by keeping only integer values inside their domains. Real coefficients in the objective or constraints are decimal numbers with the number of significant digits read from the file. This precision can be bounded by a command line option (-precision). When reading bound or linear constraints on a single variable, basic floating-point operations, directly reducing variable domains, are performed based on approximate operations (floor/ceil) with a bounded floating-point precision (see option -epsilon). Given the required numerical precisions, computations should be exact, and the solver should return optimal solutions without any floating-point errors. See, for example, MIPLIB instance app2-2.lp with optimum equal to 212040.500.

A complete description of the lp file format can be found at <https://www.ibm.com/docs/en/icos/22.1.2?topic=cplex-lp-file-format-algebraic-representation> or <https://lpsolve.sourceforge.net/5.5/lp-format.htm>.

Warning, unbounded variable domains cannot be represented.

Thanks to Gauthier Quesnel (INRAE) for this reader.

### 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](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.

## 10.9 How do I use it ?

### 10.9.1 Using it as a C++ library

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

### 10.9.2 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>.

## 10.10 References

See ‘BIBLIOGRAPHY’ at the end of the document.



## REFERENCE MANUAL

### 11.1 Introduction

<b>Cost Function Network Solver</b>	toulbar2
<b>Copyright</b>	toulbar2 team
<b>Source</b>	<a href="https://github.com/toulbar2/toulbar2">https://github.com/toulbar2/toulbar2</a>

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`.

### 11.2 Exact optimization for cost function networks and additive graphical models

#### 11.2.1 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 (*first place* on Mini COP in 2023 *and* 2025, second place on Mini COP and Parallel COP tracks in 2022, third place in 2024)

- UAI 2022 Inference Competition [UAI 2022](#) (winner on all MPE and MMAP task categories)
- Pseudo-Boolean Competition 2025 [PB25](#) (OPT-LIN ranking 39/46 ; PARTIAL-LIN ranking 6/9, but it gave the best known answer from an incomplete solver point of view in [185](#) instances among 208)

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](#).

### 11.2.2 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.

### 11.2.3 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”.

### 11.2.4 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
```

### 11.2.5 Installation from sources

Compilation requires git, cmake and a C++-17 capable compiler (in C++17 mode).

Required library:

- libgmp-dev
- bc (used during cmake)

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
- libcuuc
- libcuil8n
- libicudata
- libxml2-dev
- libxcsp3parser
- libeigen3-dev

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 -march=native -flto -static -static-libgcc -
↳static-libstdc++ -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -DWCSPFORMATONLY \
-I. -I./pils/src tb2*.cpp applis/*.cpp convex/*.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 (must be run with `mpirun`, use `mpirun -n 1` for the sequential version of HBFS or VNS):

```
bash
cd src
echo '#define Toulbar_VERSION "1.2.0"' > ToulbarVersion.hpp
mpicxx -o toulbar2 -std=c++17 -O3 -DNDEBUG -march=native -flto -DBOOST -DLONGDOUBLE_PROB_
↳-DLONGLONG_COST -DWCSPFORMATONLY -DOPENMPI \
```

(continues on next page)

(continued from previous page)

```
-I. -I./pils/src tb2*.cpp applis/*.cpp convex/*.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 -march=native -flto -DBOOST -DLONGDOUBLE_PROB_
↪-DLONGLONG_COST -DXMLFLAG3 -DOPENMPI \
-I/usr/include/libxml2 -I. -I./pils/src -I./xmlcsp3 tb2*.cpp applis/*.cpp convex/*.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
```

Copyright (C) 2006-2025, toulbar2 team. toulbar2 is currently maintained by Simon de Givry, INRAE - MIAT, Toulouse, France ([simon.de-givry@inrae.fr](mailto:simon.de-givry@inrae.fr))

## 11.3 Modules

### 11.3.1 Variable and cost function modeling

#### *group* Variable and cost function 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 ( $n \geq 4$ ) 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*, *salldifdp*, *sgccdp*, *sregulardp*, *sgrammardp*, *smstdp*, *smaxdp*)
  - *network* propagator based on cost function network decomposition with “w” prefix (*wsum*, *wvarsum*, *walldiff*, *wgcc*, *wsame*, *wsamegcc*, *wregular*, *wamong*, *wvamong*, *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

## 11.3.2 Solving cost function networks

### group Solving cost function networks

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) {
```

(continues on next page)

(continued from previous page)

```

    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 1000000 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

```

(continues on next page)

(continued from previous page)

```

*
* warning: compile with the same compilation flags as for creating libtb2.so
* e.g., g++ -DBOOST -DLONGDOUBLE_PROB -DLONGLONG_COST -I./src -o sudoku_
↳ sudoku_tutorial.cpp -L./build/lib/Linux -ltb2
*/

#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 = VAC_SUPPORT_HEUR; // option -V
    // uncomment if partial Limited Discrepancy Search enable
    //      ToulBar2::lds = 1; // option -l=1
    // uncomment if INCOP local search enable

```

(continues on next page)

(continued from previous page)

```

//      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;
    for (unsigned int i = 0; i < 2; i++) {

```

(continues on next page)



(continued from previous page)

```

        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) {
#endif
#endif

```

(continues on next page)

(continued from previous page)

```

        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: */

```

### 11.3.3 Output messages, verbosity options and debugging

#### group Output messages, verbosity options and debugging

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 “f” number\_of\_cost\_functions* (see approximate degree in [Variable elimination](#)) “f” *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

### 11.3.4 Preprocessing techniques

#### group Preprocessing techniques

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&subtracts n-ary cost functions in extension on all the binary cost functions inside their scope ( $3 < n < \text{max}$ , see `toulbar2` options)
5. functional variable elimination (see [Variable elimination](#))
6. projects&subtracts 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* occurrences 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`)

### 11.3.5 Variable and value search ordering heuristics

#### group Variable and value search ordering 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.

### 11.3.6 Soft arc consistency and problem reformulation

#### group Soft arc consistency and problem reformulation

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.

### 11.3.7 Virtual Arc Consistency enforcing

#### group Virtual Arc Consistency enforcing

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.

Note: usual domain events such that assign/remove should not be called during VAC phase 1, use removeVAC instead.

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

### 11.3.8 NC bucket sort

#### group NC bucket sort

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.

### 11.3.9 Variable elimination

#### group Variable elimination

- *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.

### 11.3.10 Propagation loop

#### *group* **Propagation loop**

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

### 11.3.11 Backtrack management

#### *group* **Backtrack management**

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::storeKnapsack for backtrackable lists of knapsack constraints
- 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)

## 11.4 Libraries

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

## 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](#)

## PUBLICATIONS

### 13.1 Conference talks

- ANITI 2025 webinar on toulbar2 and related libraries (bi-objective and model learning): [slides](#) | [script](#)
- talk on toulbar2 at [ROADEF 2023](#), Rennes, France, February 21, 2023.
- ANITI 2021 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.

### 13.2 Related publications

#### 13.2.1 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 csp](#), 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.

- **Low rank block coordinate descent (LR-BCD)**

Efficient Low Rank Convex Bounds for Pairwise Discrete Graphical Models, V Durante, G Katsirelos, and T Schiex, In Proc. of the 39th International Conference on Machine Learning (ICML), PMLR 162:5726-5741, 2022.

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

Virtual Pairwise Consistency in Cost Function Networks, P Montalbano, D Allouche, S de Givry, G Katsirelos, and T Werner In Proc. of CP-AI-OR'2023, Nice, France, 2023.

- **Bi-Objective Combinatorial Optimization (global bounding constraint)**

Bi-Objective Discrete Graphical Model Optimization, S Buchet, D Allouche, S de Givry, and T Schiex In Proc. of CP-AI-OR'2024, Uppsala, Sweden, 2024.

- **Virtual arc consistency for linear constraints (VAC-lin)**

Virtual Arc Consistency for Linear Constraints in Cost Function Networks, P. Montalbano, S. de Givry, G. Katsirelos In Proc. of ICTAI-25, Athens, Greece, 2025.

## 13.2.2 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  $\beta$ -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 mmgsa 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**

[The Genetic Architecture of Recombination Rates is Polygenic and Differs Between the Sexes in Wild House Sparrows \(\*Passer domesticus\*\)](#), McAuley JB, Servin B, Burnett HA, Brekke C, Peters L, Hagen IJ, Niskanen AK, Ringsby TH, Husby A, Jensen H, Johnston SE, *Molecular Biology and Evolution*, Volume 41, Issue 9, September 2024, msae179.

[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 Frédéric Garcia, *RAIRO - Operations Research*, 47:151-172, 2013.

### 13.2.3 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 Mé-tivier, 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 Artificial 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.
- [Ficolof2012] 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/Ficolof2012poster.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](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 Discrepancy 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.