Python Library of toulbar2

Release 1.0.0

INRAE

Sep 26, 2025

CONTENTS

pytoulbar2 software is the Python interface of toulbar2.

class pytoulbar2.**CFN**(*ubinit=None*, *resolution=0*, *vac=0*, *configuration=False*, *vns=None*, *seed=1*, *verbose=-1*, *init=True*)

pytoulbar2 base class used to manipulate and solve a cost function network.

Parameters

- **ubinit** (*decimal cost or None*) – initial upper bound.
- **resolution** (*int*) – decimal precision of costs.
- **vac** (*int*) – if non zero, maximum solver depth minus one where virtual arc consistency algorithm is applied (1: VAC only in preprocessing).
- **configuration** (*bool*) – if True then special settings for preference learning using incremental solving (see car configuration tutorial).
- **vns** (*int or None*) – if None then solves using branch-and-bound methods else using variable neighborhood search heuristic (-1: initial solution at random, -2: minimum domain values, -3: maximum domain values,

  -4: first solution found by DFS, >=0: or by LDS with at most vns discrepancies).
- **seed** (*int*) – random seed.
- **verbose** (*int*) – verbosity control (-1: no message, 0: search statistics, 1: search tree, 2-7: propagation information).
- **init** (*bool*) – starts from scratch, forgets previous CFNs (default value is True). Must be set to False if this CFN depends on a previous CFN (e.g. this CFN is given as input to a AddWeightedCSPConstraint).

Members:

CFN (WeightedCSPSolver): python interface to C++ class WeightedCSPSolver.

Contradiction (exception): python exception corresponding to the same C++ class.

Limit (exception|None): contains the last SolverOut exception or None if no exception occurs when solving with SolveNext.

Option (TouBar2): python interface to C++ class ToulBar2.

SolverOut (exception): python exception corresponding to the same C++ class.

Top (decimal cost): maximum decimal cost (it can be used to represent a forbidden cost).

Variables (dict): associative array returning the original domain (list or iterable) associated to a given variable name (str).

VariableIndices (dict): associative array returning the variable name (str) associated to a given index (int).

VariableNames (list): array of created variable names (str) sorted by their index number.

See pytoulbar2test.py example in src repository.

AddAllDifferent(*scope*, *encoding='binary'*, *excepted=None*, *incremental=False*)

Add AllDifferent hard global constraint.

Parameters

- **scope** (*list*) – input variables of the function. A variable can be represented by its name (str) or its index (int).

- **encoding** (*str*) – encoding used to represent AllDifferent (available choices are 'binary' or 'salldiff' or 'salldiffdp' or 'salldiffkp' or 'walldiff').
- **excepted** (*None or list*) – list of excepted domain values which can be taken by any variable without violating the constraint.
- **incremental** (*bool*) – if True then the constraint is backtrackable (i.e., it disappears when restoring at a lower depth, see Store/Restore).

AddCompactFunction(*scope*, *defcost*, *tuples*, *tcosts*, *incremental=False*)

AddCompactFunction creates a cost function in extension. The scope corresponds to the input variables of the function. The costs are given by a list of assignments with the corresponding list of costs, all the other assignments taking the default cost.

Parameters

- **scope** (*list*) – input variables of the function. A variable can be represented by its name (str) or its index (int).
- **defcost** (*decimal cost*) – default cost.
- **tuples** (*list*) – array of assignments (each assignment is a list of domain values, following the scope order).
- **tcosts** (*list*) – array of corresponding decimal costs (tcosts and tuples have the same size).
- **incremental** (*bool*) – if True then the function is backtrackable (i.e., it disappears when restoring at a lower depth, see Store/Restore).

Example

AddCompactFunction(['x','y','z'],0,[[0,0,0],[1,1,1]],[1,-1]) encodes a ternary cost function with the null assignment having a cost of 1, the identity assignment having a cost of -1, and all the other assignments a cost of 0.

AddFunction(*scope*, *costs*, *incremental=False*)

AddFunction creates a cost function in extension. The scope corresponds to the input variables of the function. The costs are given by a flat array the size of which corresponds to the product of initial domain sizes (see note in AddVariable and also GetDomainInitSize).

Parameters

- **scope** (*list*) – input variables of the function. A variable can be represented by its name (str) or its index (int).
- **costs** (*list*) – array of decimal costs for all possible assignments (iterating first over the domain values of the last variable in the scope).
- **incremental** (*bool*) – if True then the function is backtrackable (i.e., it disappears when restoring at a lower depth, see Store/Restore).

Example

AddFunction(['x','y'], [0,1,1,0]) encodes a binary cost function on Boolean variables x and y such that (x=0,y=0) has a cost of 0, (x=0,y=1) has a cost of 1, (x=1,y=0) has a cost of 1, and (x=1,y=1) has a cost of 0.

AddGeneralizedLinearConstraint(*tuples*, *operand='==', rightcoef=0*)

AddGeneralizedLinearConstraint creates a linear constraint with integer coefficients associated to domain values. The scope implicitely corresponds to the variables involved in the tuples. Missing domain values have an implicit zero coefficient. All constant terms must belong to the right part.

Parameters

- **tuples** (*list*) – array of triplets (variable, domain value, coefficient) in the left part of the constraint.
- **operand** (*str*) – can be either '==' or '<=' or '<' or '>=' or '>'.
- **rightcoef** (*int*) – constant term in the right part.

Example

AddGeneralizedLinearConstraint([('x',1,1),('y',1,1),('z',0,2)], '==', 1) encodes (x==1) + (y==1) + 2*(z==0) = 1 assuming 0/1 variables and (x==u) is equal to 1 if value u is assigned to x else equal to 0.

AddGlobalFunction(*scope*, *gcname*, **parameters*)

AddGlobalFunction creates a soft global cost function.

Parameters

- **scope** (*list*) – input variables of the function. A variable can be represented by its name (str) or its index (int).
- **gcname** (*str*) – name of the global cost function (see toulbar2 user documentation).
- **parameters** (*list*) – list of parameters (str or int) for this global cost function.

Example

AddGlobalFunction(['x1','x2','x3','x4'], 'wamong', 'hard', 1000, 2, 1, 2, 1, 3) encodes a hard among constraint satisfied iff values {1,2} are assigned to the given variables at least once and at most 3 times, otherwise it returns a cost of 1000.

AddLinearConstraint(*coefs*, *scope*, *operand='=='*, *rightcoef=0*)

AddLinearConstraint creates a linear constraint with integer coefficients. The scope corresponds to the variables involved in the left part of the constraint. All variables must belong to the left part (change their coefficient sign if they are originally in the right part). All constant terms must belong to the rigt part.

Parameters

- **coefs** (*list or int*) – array of integer coefficients associated to the left-part variables (or the same integer coefficient is applied to all variables).
- **scope** (*list*) – variables involved in the left part of the constraint. A variable can be represented by its name (str) or its index (int).
- **operand** (*str*) – can be either '==' or '<=' or '<' or '>=' or '>'.
- **rightcoef** (*int*) – constant term in the right part.

Example

AddLinearConstraint([1,1,-2], [x,y,z], '==', -1) encodes x + y -2z = -1.

AddSumConstraint(*scope*, *operand='=='*, *rightcoef=0*)

AddSumConstraint creates a linear constraint with unit coefficients. The scope corresponds to the variables involved in the left part of the constraint.

Parameters

- **scope** (*list*) – variables involved in the left part of the constraint. A variable can be represented by its name (str) or its index (int).

- **operand** (*str*) – can be either '==' or '<=' or '<' or '>=' or '>'.
- **rightcoef** (*int*) – constant term in the right part.

Example

AddSumConstraint([x,y,z], '<', 3) encodes x + y + z < 3.

AddVariable(*name*, *values*)

AddVariable creates a new discrete variable.

Parameters

- **name** (*str*) – variable name.
- **values** (*list or iterable*) – list of domain values represented by numerical (int) or symbolic (str) values.

Returns

Index of the created variable in the problem (int).

> **Note**
>
> Symbolic values are implicitely associated to integer values (starting from zero) in the other functions. In case of numerical values, the initial domain size is equal to max(values)-min(values)+1 and not equal to len(values). Otherwise (symbolic case), the initial domain size is equal to len(values).

AddWeightedCSPConstraint(*problem*, *lb*, *ub*, *duplicateHard=False*, *strongDuality=False*)

AddWeightedCSPConstraint creates a hard global constraint on the cost of an input weighted constraint satisfaction problem such that its valid solutions must have a cost value in [lb,ub[.

Parameters

- **problem** (CFN) – input problem.
- **lb** (*decimal cost*) – any valid solution in the input problem must have a cost greater than or equal to lb.
- **ub** (*decimal cost*) – any valid solution in the input problem must have a cost strictly less than ub.
- **duplicateHard** (*bool*) – 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** (*bool*) – 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.

> **Note**
>
> If a variable in the input problem does not exist in the current problem (with the same name), it is automatically added.

Example

m=tb2.CFN(); m.Read("master.cfn");s=tb2.CFN();s.Read("slave.cfn");m.AddWeightedCSPConstraint(s, lb, ub);m.Solve()

Assign(*var*, *value*)

Assign assigns a variable to a domain value.

Parameters

- **var** (*int* | *str*) – variable name or its index as returned by AddVariable.
- **value** (*int*) – domain value.

ClearPropagationQueues()

ClearPropagationQueues resets propagation queues. It should be called when an exception Contradiction occurs.

Deconnect(*var*)

Deconnect deconnects a variable from the rest of the problem and assigns it to its support value.

Parameters
var (*int* | *str*) – variable name or its index as returned by AddVariable.

Decrease(*var*, *value*)

Decrease removes the last values strictly greater than a given value in the domain of a variable.

Parameters

- **var** (*int* | *str*) – variable name or its index as returned by AddVariable.
- **value** (*int*) – domain value.

Depth()

Depth returns the current solver depth value.

Returns
Current solver depth value (int).

Domain(*var*)

Domain returns the current domain of a given variable.

Parameters
var (*int* | *str*) – variable name or its index as returned by AddVariable.

Returns
List of domain values (list).

Dump(*filename*)

Dump outputs the problem in a file (without doing any preprocessing).

Parameters
filename (*str*) – problem filename. The suffix must be '.wcsp' or '.cfn' to select in which format to save the problem.

GetDDualBound()

GetDDualBound returns the global problem lower bound in minimization (resp. upper bound in maximization) found after doing an incomplete search with Solve.

Returns
Global lower bound (decimal cost).

GetDomainInitSize(*var*)

GetDomainInitSize returns the initial domain size of a given variable.

Parameters
var (*int* | *str*) – variable name or its index as returned by AddVariable.

Returns
Initial domain size (int). See also GetValue, GetValueName, and GetValueIndex.

GetLB()

GetLB returns the current problem lower bound.

Returns
Current lower bound (decimal cost).

GetName()

GetName get the name of the CFN.

Returns
Name of the CFN (string).

GetNbBacktracks()

GetNbBacktracks returns the number of backtracks done so far.

Returns
Current number of backtracks (int).

GetNbConstrs()

GetNbConstrs returns the number of non-unary cost functions.

Returns
Number of non-unary cost functions (int).

GetNbNodes()

GetNbNodes returns the number of search nodes explored so far.

Returns
Current number of search nodes (int).

GetNbVars()

GetNbVars returns the number of variables.

Returns
Number of variables (int).

GetSolutions()

GetSolutions returns all the solutions found so far with their associated costs.

Returns
List of pairs (decimal cost, solution) where a solution is a list of domain values.

GetUB()

GetUB returns the initial upper bound.

Returns
Current initial upper bound (decimal cost).

GetValue(*var*, *index*)

GetValue returns the value at position index in the initial domain of a given variable.

Parameters

- **var** (*int*|*str*) – variable name or its index as returned by AddVariable.
- **index** (*int*) – index of the value in the initial domain of the variable (see also GetDomainInitSize).

Returns
domain value (int). In symbolic domains, the returned value is equal to index (see note in AddVariable).

GetValueIndex(*var*, *value*)

GetValueIndex returns the position index in the initial domain of a given variable which corresponds to the given value.

Parameters

- **var** (*int*|*str*) – variable name or its index as returned by AddVariable.
- **value** (*int*|*str*) – domain value (or its symbolic name).

Returns
Index of the value in the initial domain of the variable (int).

GetValueName(*var*, *index*)

GetValueName returns the symbolic value at position index in the initial domain of a given variable.

Parameters

- **var** (*int*|*str*) – variable name or its index as returned by AddVariable.
- **index** (*int*) – index of the value in the initial domain of the variable (see also GetDomainInitSize).

Returns
symbolic name corresponding to a domain value (str).

Increase(*var*, *value*)

Increase removes the first values strictly lower than a given value in the domain of a variable.

Parameters

- **var** (*int*|*str*) – variable name or its index as returned by AddVariable.
- **value** (*int*) – domain value.

InitFromMultiCFN(*multicfn*, *vars=[]*, *scopes=[]*, *constrs=[]*)

InitFromMultiCFN initializes the cfn from a multiCFN instance (linear combination of multiple CFNs).

Parameters

- **multicfn** (MultiCFN) – the instance containing the CFNs.
- **vars** (*list<int*|*str>*) – the list of variable indexes to extract the induced graph (if empty then no restriction)
- **scopes** (*list<set<int*|*str>>*) – the list of allowed scopes to extract the partial graph (if empty then no restriction)
- **constrs** (*list<int>*) – the list of allowed cost function indexes (same index as stored in MultiCFN) to extract the partial graph (if empty then no restriction)

> **Note**
>
> After beeing initialized, it is possible to add cost functions to the CFN but the upper bound may be inconsistent. The problem lower bound of multicfn is exported only if no restrictions are given (or if scopes contains the emptyset).

MultipleAssign(*vars*, *values*)

MultipleAssign assigns several variables at once.

Parameters

- **vars** (*list*) – list of indexes or names of variables.
- **values** (*list*) – list of domain values.

MultipleDeconnect(*vars*)

MultipleDeconnect deconnects a set of variables from the rest of the problem and assigns them to their support value.

Parameters

vars (*list*) – list of indexes or names of variables.

NoPreprocessing()

NoPreprocessing deactivates most preprocessing methods.

Parse(*certificate*)

Parse performs a list of elementary reduction operations on domains of variables.

Parameters

certificate (*str*) – a string composed of a list of operations on domains, each operation in the form ',varIndex[=#<>]value' where varIndex (int) is the index of a variable as returned by AddVariable and value (int) is a domain value (comma is mandatory even for the first operation, add no space). Possible operations are: assign ('='), remove ('#'), decrease maximum value ('<'), increase minimum value ('>').

Example

Parse(',0=1,1=1,2#0'): assigns the first and second variable to value 1 and remove value 0 from the third variable.

Print()

Print prints the content of the CFN (variables, cost functions).

Read(*filename*)

Read reads the problem from a file.

Parameters

filename (*str*) – problem filename.

Remove(*var*, *value*)

Remove removes a value from the domain of a variable.

Parameters

- **var** (*int* | *str*) – variable name or its index as returned by AddVariable.
- **value** (*int*) – domain value.

Restore(*depth*)

Restore retrieves the copy made at a given solver depth value.

Parameters
depth (*int*) – solver depth value. It must be lower than the current solver depth.

SetName(*name*)

SetName set the name of the CFN.

Parameters
name (*str*) – the new name of the CFN.

SetUB(*cost*)

SetUB resets the initial upper bound to a given value. It should be done before modifying the problem.

Parameters
cost (*decimal cost*) – new initial upper bound.

> **Warning**
>
> This operation should be called after SolveFirst and before SolveNext.

Solve(*showSolutions=0*, *allSolutions=0*, *diversityBound=0*, *timeLimit=0*, *writeSolution=''*)

Solve solves the problem (i.e., finds its optimum and proves optimality). It can also enumerate (diverse) solutions depending on the arguments.

Parameters

- **showSolutions** (*int*) – prints solution(s) found (0: show nothing, 1: domain values, 2: variable names with their assigned values, 3: variable and value names).
- **allSolutions** (*int*) – if non-zero, enumerates all the solutions with a cost strictly better than the initial upper bound until a given limit on the number of solutions is reached.
- **diversityBound** (*int*) – if non-zero, finds a greedy sequence of diverse solutions where a solution in the list is optimal such that it also has a Hamming-distance from the previously found solutions greater than a given bound. The number of diverse solutions is bounded by the argument value of allSolutions.
- **timeLimit** (*int*) – CPU-time limit in seconds (or 0 if no time limit)
- **writeSolution** (*str*) – write best solution found in a file using a given file name and using the same format as showSolutions (or write all solutions if allSolutions is non-zero)

Returns
The best (or last if enumeration/diversity) solution found as a list of domain values, its associated cost, always strictly lower than the initial upper bound, and the number of solutions found (returned type: tuple(list, decimal cost, int)). or None if no solution has been found (the problem has no solution better than the initial upper bound or a search limit occurs). See GetSolutions to retrieve of the solutions found so far. See GetDDualBound to retrieve of the global problem dual bound found so far.

> **Warning**
>
> This operation cannot be called multiple times on the same CFN object (it may modify the problem or its upper bound).

SolveFirst()

SolveFirst performs problem preprocessing before doing incremental solving.

Returns

Initial upper bound (decimal cost), possibly improved by considering a worst-case situation based on the sum of maximum finite cost per function plus one. or None if the problem has no solution (a contradiction occurs during preprocessing).

Warning

This operation must be done at solver depth 0 (see Depth).

Warning

This operation cannot be called multiple times on the same CFN object.

SolveNext(*showSolutions=0*, *timeLimit=0*)

SolveNext solves the problem (i.e., finds its optimum and proves optimality). It should be done after calling SolveFirst and modifying the problem if necessary using SetUB, Assign, MultipleAssign, Remove, Increase, Decrease, or adding an incremental cost function.

Parameters

- **showSolutions** (*int*) – prints solution(s) found (0: show nothing, 1: domain values, 2: variable names with their assigned values, 3: variable and value names).
- **timeLimit** (*int*) – CPU-time limit in seconds (or 0 if no time limit)

Returns

The best solution found as a list of domain values, its associated cost, always strictly lower than the initial upper bound, and None (returned type: tuple(list, decimal cost, None)). or None if no solution has been found (the problem has no solution better than the initial upper bound or a search limit occurs, see Limit).

Store()

Store makes a copy (incremental) of the current problem and increases the solver depth by one.

UpdateUB(*cost*)

UpdateUB decreases the initial upper bound to a given value. Does nothing if this value is greater than the current upper bound.

Parameters

cost (*decimal cost*) – new initial upper bound.

Warning

This operation might generate a Contradiction if the new upper bound is lower than or equal to the problem lower bound.

static flatten(*S*)

Warning: this recursive method might exceed maximum recursion depth

class pytoulbar2.**MultiCFN**

pytoulbar2 base class used to combine linearly multiple CFNs. See (InitFromMultiCFN) to extract and solve it.

Members:

MultiCFN: python interface to C++ class MultiCFN.

> **Note**
>
> It is important to set the parameter (resolution) to the same ***nonzero*** value when creating every CFN to be pushed in a MultiCFN object.

ApproximateParetoFront(*first_criterion*, *first_direction*, *second_criterion*, *second_direction*, *showSolutions=0*, *timeLimit=0*, *timeLimit_per_solution=0*, *max_sol_count=0*)

ApproximateParetoFront returns the set of supported solutions of the problem on two criteria (on the convex hull of the non dominated solutions).

Parameters

- **first_criterion** (*int*) – index of the first CFN to optimize.
- **first_direction** (*str*) – direction of the first criterion: 'min' or 'max'.
- **second_criterion** (*int*) – index of the second CFN to optimize.
- **second_direction** (*str*) – direction of the second criterion: 'min' or 'max'.
- **showSolutions** (*int*) – prints all intermediate (dominated and nondominated) solution(s) found (0: show nothing, 1: domain values, 2: variable names with their assigned values, 3: variable and value names).
- **timeLimit** (*int*) – CPU-time limit in seconds for the whole method (0 by default, meaning no time limit)
- **timeLimit_per_solution** (*int*) – CPU-time limit in seconds for the computation of each solution (0 by default, meaning no time limit)
- **max_sol_count** (*int*) – limit the maximum number of solutions to compute (0 by default, meaning no limit)

Returns

The non dominated solutions belonging to the convex hull of the pareto front and their costs (tuple).

GetNbCFN()

GetNbCFN returns the number of CFN pushd in the MultiCFN.

Returns

Number of CFN (int).

GetSolution()

GetSolution returns the solution of the combined cfn after being solved.

Returns

The solution of the cfn (dic).

GetSolutionCosts()

GetSolutionCosts returns the costs of the combined cfn after being solved.

Returns

The costs of the solution of the cfn (list).

GetVariableIndex(*name*)

GetVariableIndex returns the index of the variable in the combined cfn

Parameters
name (*str*) – name of the variable.

Returns
The index of the variable or -1 if not found (int).

Print()

Print print the content of the multiCFN: variables, cost functions.

PushCFN(*CFN*, *weight=1.0*)

PushCFN add a CFN to the instance.

Parameters

- **CFN** (CFN) – the new CFN to add.
- **weight** (*float*) – the initial weight of the CFN in the combination.

SetWeight(*cfn_index*, *weight*)

SetWeight set a weight of a CFN.

Parameters

- **cfn_index** (*int*) – index of the CFN (in addition order).
- **weight** (*float*) – the new weight of the CFN.

INDEX

U