Python Library of toulbar2

Release 1.0.0

INRAE

Jun 17, 2022

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*)

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

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. 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). 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** (*string*) – 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).

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 values have an implicit zero coefficient. All constant terms must belong to the rigt part.

Parameters

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

Example

AddGeneralizedLinearConstraint([(‘x’,1,1),(‘y’,1,1),(‘z’,1,-2)], ‘==’, -1) also encodes x + y -2z = -1 assuming 0/1 variables.

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*) – array of integer coefficients associated to the left-part 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 '>='.
- **rightcoef** (*int*) – constant term in the right part.

Example

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

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).

Assign(*varIndex*, *value*)

Assign assigns a variable to a domain value.

Parameters

- **varIndex** (*int*) – index of the variable as returned by AddVariable.
- **value** (*int*) – domain value.

ClearPropagationQueues()

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

Deconnect(*varIndex*)

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

Parameters **varIndex** (*int*) – index of the variable as returned by AddVariable.

Decrease(*varIndex*, *value*)

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

Parameters

- **varIndex** (*int*) – index of the variable as returned by AddVariable.
- **value** (*int*) – domain value.

Depth()

Depth returns the current solver depth value.

Returns Current solver depth value (int).

Domain(*varIndex*)

Domain returns the current domain of a given variable.

Parameters **varIndex** (*int*) – index of the variable 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.

GetLB()

GetLB returns the current problem lower bound.

Returns Current lower bound (decimal cost).

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).

Increase(*varIndex*, *value*)

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

Parameters

- **varIndex** (*int*) – index of the variable as returned by AddVariable.
- **value** (*int*) – domain value.

MultipleAssign(*varIndexes*, *values*)

MultipleAssign assigns several variables at once.

Parameters

- **varIndexes** (*list*) – list of indexes of variables.
- **values** (*list*) – list of domain values.

MultipleDeconnect(*varIndexes*)

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

Parameters **varIndexes** (*list*) – list of indexes 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.

Read(*filename*)

Read reads the problem from a file.

Parameters **filename** (*str*) – problem filename.

Remove(*varIndex*, *value*)

Remove removes a value from the domain of a variable.

Parameters

- **varIndex** (*int*) – index of the variable 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.

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.

Solve(*showSolutions=0*, *allSolutions=0*, *diversityBound=0*)

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.

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).

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*)

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.

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).

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*)

INDEX