Classes

Contents:

Classes

Classes used in solver callbacks, for a bi-directional communication with the solver engine

Model

class `Model`**(*name=''*, *sense='MIN'*, *solver_name=''*, *solver=None*)** Mixed Integer Programming Model

This is the main class, providing methods for building, optimizing, querying optimization results and re-optimizing Mixed-Integer Programming Models.

To check how models are created please see the examples included.

`vars` list of problem variables (`Var`)

Type mip.VarList

`constrs` list of constraints (`Constr`)

Type mip.ConstrList

Examples

```
>>> from mip import Model, MAXIMIZE, CBC, INTEGER, OptimizationStatus
>>> model = Model(sense=MAXIMIZE, solver_name=CBC)
>>> x = model.add_var(name='x', var_type=INTEGER, lb=0, ub=10)
>>> y = model.add_var(name='y', var_type=INTEGER, lb=0, ub=10)
>>> model += x + y <= 10
>>> model.objective = x + y
>>> status = model.optimize(max_seconds=2)
```

`add_constr`(*lin_expr*, *name=''*, *priority=None*) Creates a new constraint (row).

Adds a new constraint to the model, returning its reference.

Parameters **lin_expr** (*mip.LinExpr*) – linear expression

- **name** (*str*) – optional constraint name, used when saving model to lp or mps files
- **priority** (*mip.constants.ConstraintPriority*) – optional constraint priority

Examples:

The following code adds the constraint $x_1 + x_2 \leq 1$ (x1 and x2 should be created first using `add_var()`):

```
m += x1 + x2 <= 1
```

Which is equivalent to:

```
m.add_constr( x1 + x2 <= 1 )
```

Summation expressions can be used also, to add the constraint $\sum_{i=0}^{n-1} x_i = y$ and name this constraint `cons1`:

```
m += xsum(x[i] for i in range(n)) == y, "cons1"
```

Which is equivalent to:

```
m.add_constr( xsum(x[i] for i in range(n)) == y, "cons1" )
```

Return type mip.Constr

`add_cut`(*cut*) Adds a violated inequality (cutting plane) to the linear programming model. If called outside the cut callback performs exactly as `add_constr()`. When called inside the cut callback the cut is included in the solver's cut pool, which will later decide if this cut should be added or not to the model. Repeated cuts, or cuts which will probably be less effective, e.g. with a very small violation, can be discarded.

Parameters **cut** (*mip.LinExpr*) – violated inequality

`add_lazy_constr`(*expr*) Adds a lazy constraint

A lazy constraint is a constraint that is only inserted into the model after the first integer solution that violates it is found. When lazy constraints are used a restricted pre-processing is executed since the complete model is not available at the beginning. If the number of lazy constraints is too large then they can be added during the search process by implementing a `ConstrsGenerator` and setting the property `lazy_constrs_generator` of `Model`.

Parameters **expr** (*mip.LinExpr*) – the linear constraint

`add_sos`(*sos*, *sos_type*) Adds an Special Ordered Set (SOS) to the model

An explanation on Special Ordered Sets is provided here.

Parameters **sos** (*List[Tuple[Var, numbers.Real]]*) – list including variables (not necessarily binary) and respective weights in the model

- **sos_type** (*int*) – 1 for Type 1 SOS, where at most one of the binary variables can be set to one and 2 for Type 2 SOS, where at most two variables from the list may be selected. In type 2 SOS the two selected variables will be consecutive in the list.

`add_var`(*name=''*, *lb=0.0*, *ub=inf*, *obj=0.0*, *var_type='C'*, *column=None*) Creates a new variable in the model, returning its reference

Parameters **name** (*str*) – variable name (optional)

- **lb** (*numbers.Real*) – variable lower bound, default 0.0
- **ub** (*numbers.Real*) – variable upper bound, default infinity
- **obj** (*numbers.Real*) – coefficient of this variable in the objective function, default 0
- **var_type** (*str*) – CONTINUOUS ("C"), BINARY ("B") or INTEGER ("I")

Examples

To add a variable `x` which is continuous and greater or equal to zero to model `m`:

```
x = m.add_var()
```

The following code adds a vector of binary variables `x[0], ..., x[n-1]` to the model `m`:

```
x = [m.add_var(var_type=BINARY) for i in range(n)]
```

Return type mip.Var

`add_var_tensor`(***shape***, ***name***, ******kwargs***) Creates new variables in the model, arranging them in a numpy tensor and returning its reference

Parameters **shape** (*Tuple[int, ..]*) – shape of the numpy tensor

- **name** (*str*) – variable name
- ****kwargs** – all other named arguments will be used as `add_var()` arguments

Examples

To add a tensor of variables `x` with shape (3, 5) and which is continuous in any variable and have all values greater or equal to zero to model `m`:

```
x = m.add_var_tensor((3, 5), "x")
```

Return type mip.LinExprTensor

`check_optimization_results`() Checks the consistency of the optimization results, i.e., if the solution(s) produced by the MIP solver respect all constraints and variable values are within acceptable bounds and are integral when requested.

`clear`() Clears the model

All variables, constraints and parameters will be reset. In addition, a new solver instance will be instantiated to implement the formulation.

property `clique` Controls the generation of clique cuts. -1 means automatic, 0 disables it, 1 enables it and 2 enables more aggressive clique generation.

Return type `int`

`clique_merge`(***constrs=None***) This procedure searches for constraints with conflicting variables and attempts to group these constraints in larger constraints with all conflicts merged.

For example, if your model has the following constraints:

$$x_1 + x_2 \leq 1$$
$$x_2 + x_3 \leq 1$$
$$x_1 + x_3 \leq 1$$

Then they can all be removed and replaced by the stronger inequality:

$$x_1 + x_2 + x_3 \leq 1$$

Parameters **constrs** (*Optional[List[mip.Constr]]*) – constraints that should be checked for merging. All constraints will be checked if `constrs` is None.

property `conflict_graph` Returns the `ConflictGraph` of a MIP model.

Return type mip.ConflictGraph

`constr_by_name`(***name***) Queries a constraint by its name

Parameters **name** (*str*) – constraint name

Return type Optional[mip.Constr]

Returns constraint or None if not found

`copy`(***solver_name=''***) Creates a copy of the current model

Parameters **solver_name** (*str*) – solver name (optional)

v: latest

Returns clone of current model

property `cut_passes` Maximum number of rounds of cutting planes. You may set this parameter to low values if you see that a significant amount of time is being spent generating cuts without any improvement in the lower bound. -1 means automatic, values greater than zero specify the maximum number of rounds.

Return type `int`

property `cutoff` upper limit for the solution cost, solutions with cost > cutoff will be removed from the search space, a small cutoff value may significantly speedup the search, but if cutoff is set to a value too low the model will become infeasible

Return type `Real`

property `cuts` Controls the generation of cutting planes, -1 means automatic, 0 disables completely, 1 (default) generates cutting planes in a moderate way, 2 generates cutting planes aggressively and 3 generates even more cutting planes. Cutting planes usually improve the LP relaxation bound but also make the solution time of the LP relaxation larger, so the overall effect is hard to predict and experimenting different values for this parameter may be beneficial.

Return type `int`

property `cuts_generator` A cuts generator is an `ConstrsGenerator` object that receives a fractional solution and tries to generate one or more constraints (cuts) to remove it. The cuts generator is called in every node of the branch-and-cut tree where a solution that violates the integrality constraint of one or more variables is found.

Return type Optional[mip.ConstrsGenerator]

property `emphasis` defines the main objective of the search, if set to 1 (FEASIBILITY) then the search process will focus on try to find quickly feasible solutions and improving them; if set to 2 (OPTIMALITY) then the search process will try to find a provable optimal solution, procedures to further improve the lower bounds will be activated in this setting, this may increase the time to produce the first feasible solutions but will probably pay off in longer runs; the default option if 0, where a balance between optimality and feasibility is sought.

Return type mip.SearchEmphasis

property `gap` The optimality gap considering the cost of the best solution found (`objective_value`) b and the best objective bound l (`objective_bound`) g is computed as: $g =$ $$frac|b - l||b|$$. If no solution was found or if $b = 0$ then $g = \infty$. If the optimal solution was found then $g = 0$.

Return type `float`

`generate_cuts`(***cut_types=None, depth=0, npass=0, max_cuts=8192, min_viol=0.0001***) Tries to generate cutting planes for the current fractional solution. To optimize only the linear programming relaxation and not discard integrality information from variables you must call first `model.optimize(relax=True)`.

This method only works with the CBC mip solver, as Gurobi does not supports calling only cut generators.

Parameters **cut_types** (*List[CutType]*) – types of cuts that can be generated, if an empty list is specified then all available cut generators will be called.

- **depth** (`int`) – depth of the search tree, when informed the cut generator may decide to generate more/less cuts depending on the depth.
- **max_cuts** (*int*) – cut separation will stop when at least max_cuts violated cuts were found.
- **min_viol** (*float*) – cuts which are not violated by at least min_viol will be discarded.

Return type mip.CutPool

property `infeas_tol` Maximum allowed violation for constraints.

Default value: 1e-6. Tightening this value can increase the numerical precision but also probably increase the running time. As floating point computations always involve some loss of precision, values too close to zero will likely render some models impossible to optimize.

Return type `float`

property `integer_tol` Maximum distance to the nearest integer for a variable to be considered with an integer value. Default value: 1e-6. Tightening this value can increase the numerical precision but also probably increase the running time. As floating point computations always involve some loss of precision, values too close to zero will likely render some models impossible to optimize.

property **lazy_constrs_generator** A lazy constraints generator is an `ConstrsGenerator` object that receives an integer solution and checks its feasibility. If the solution is not feasible then one or more constraints can be generated to remove it. When a lazy constraints generator is informed it is assumed that the initial formulation is incomplete. Thus, a restricted pre-processing routine may be applied. If the initial formulation is incomplete, it may be interesting to use the same `ConstrsGenerator` to generate cuts *and* lazy constraints. The use of *only* lazy constraints may be useful then integer solutions rarely violate these constraints.

Return type Optional[mip.ConstrsGenerator]

property **lp_method** Which method should be used to solve the linear programming problem. If the problem has integer variables that this affects only the solution of the first linear programming relaxation.

Return type mip.LP_Method

property **max_mip_gap** value indicating the tolerance for the maximum percentage deviation from the optimal solution cost, if a solution with cost c and a lower bound l are available and $(c-l)/l <$ `max_mip_gap` the search will be concluded. Default value: 1e-4.

Return type `float`

property **max_mip_gap_abs** Tolerance for the quality of the optimal solution, if a solution with cost c and a lower bound l are available and $c-l <$ `mip_gap_abs`, the search will be concluded, see `max_mip_gap` to determine a percentage value. Default value: 1e-10.

Return type `float`

property **max_nodes** maximum number of nodes to be explored in the search tree

Return type `int`

property **max_seconds** time limit in seconds for search

Return type `float`

property **max_solutions** solution limit, search will be stopped when `max_solutions` were found

Return type `int`

property **name** The problem (instance) name.

This name should be used to identify the instance that this model refers, e.g.: productionPlanningMay19. This name is stored when saving (`write()`) the model in `.LP` or `.MPS` file formats.

Return type `str`

property **num_cols** number of columns (variables) in the model

Return type `int`

property **num_int** number of integer variables in the model

Return type `int`

property **num_nz** number of non-zeros in the constraint matrix

Return type `int`

property **num_rows** number of rows (constraints) in the model

Return type `int`

property **num_solutions** Number of solutions found during the MIP search

Return type `int`

Returns number of solutions stored in the solution pool

property **objective** The objective function of the problem as a linear expression.

Examples

The following code adds all `x` variables `x[0], ..., x[n-1]`, to the objective function of model `m` with the same cost `w`:

```
m.objective = xsum(w*x[i] for i in range(n))
```


A simpler way to define the objective function is the use of the model operator +

Note that the only difference of adding a constraint is the lack of a sense and a rhs.

Return type mip.LinExpr

property `objective_bound` A valid estimate computed for the optimal solution cost, lower bound in the case of minimization, equals to `objective_value` if the optimal solution was found.

Return type `Optional`[`Real`]

property `objective_const` Returns the constant part of the objective function

Return type `float`

property `objective_value` Objective function value of the solution found or None if model was not optimized

Return type `Optional`[`Real`]

property `objective_values` List of costs of all solutions in the solution pool

Return type `List`[`Real`]

Returns costs of all solutions stored in the solution pool as an array from 0 (the best solution) to `num_solutions`-1.

property `opt_tol` Maximum reduced cost value for a solution of the LP relaxation to be considered optimal. Default value: 1e-6. Tightening this value can increase the numerical precision but also probably increase the running time. As floating point computations always involve some loss of precision, values too close to zero will likely render some models impossible to optimize.

Return type `float`

`optimize`(***max_seconds=inf, max_nodes=1073741824, max_solutions=1073741824, max_seconds_same_incumbent=inf, max_nodes_same_incumbent=1073741824, relax=False***)

Optimizes current model

Optimizes current model, optionally specifying processing limits.

To optimize model `m` within a processing time limit of 300 seconds:

```
m.optimize(max_seconds=300)
```

Parameters **max_seconds** (*numbers.Real*) – Maximum runtime in seconds (default: inf)

- **max_nodes** (*int*) – Maximum number of nodes (default: inf)
- **max_solutions** (*int*) – Maximum number of solutions (default: inf)
- **max_seconds_same_incumbent** (*numbers.Real*) – Maximum time in seconds that the search can go on if a feasible solution is available and it is not being improved
- **max_nodes_same_incumbent** (*int*) – Maximum number of nodes that the search can go on if a feasible solution is available and it is not being improved
- **relax** (*bool*) – if true only the linear programming relaxation will be solved, i.e. integrality constraints will be temporarily discarded.

Returns optimization status, which can be OPTIMAL(0), ERROR(-1), INFEASIBLE(1), UNBOUNDED(2). When optimizing problems with integer variables some additional cases may happen, FEASIBLE(3) for the case when a feasible solution was found but optimality was not proved, INT_INFEASIBLE(4) for the case when the lp relaxation is feasible but no feasible integer solution exists and NO_SOLUTION_FOUND(5) for the case when an integer solution was not found in the optimization.

Return type mip.OptimizationStatus

property `preprocess` Enables/disables pre-processing. Pre-processing tries to improve your MIP formulation. -1 means automatic, 0 means off and 1 means on.

Return type `int`

property `pump_passes` Number of passes of the Feasibility Pump [FGL05] heuristic. You may increase this value if you not getting feasible solutions.

Return type `int`

One of the following file name extensions should be used to define the contents of what will be loaded:

`.lp` mip model stored in the LP file format

`.mps` mip model stored in the MPS file format

`.sol` initial integer feasible solution

`.bas` optimal basis for the linear programming relaxation.

Note: if a new problem is readed, all variables, constraints and parameters from the current model will be cleared.

Parameters **path** (*str*) – file name

`relax()` Relax integrality constraints of variables

Changes the type of all integer and binary variables to continuous. Bounds are preserved.

`remove`(*objects*) removes variable(s) and/or constraint(s) from the model

Parameters **objects** (*Union[mip.Var, mip.Constr, List[Union[mip.Var, mip.Constr]]]*) – can be a `Var`, a `Constr` or a list of these objects

property **`round_int_vars`** MIP solvers perform computations using *limited precision* arithmetic. Thus a variable with value 0 may appear in the solution as 0.000000000001. Thus, comparing this var to zero would return false. The safest approach would be to use something like abs(v.x) < 1e-7. To simplify code the solution value of integer variables can be automatically rounded to the nearest integer and then, comparisons like v.x == 0 would work. Rounding is not always a good idea specially in models with numerical instability, since it can increase the infeasibilities.

Return type `bool`

property **`search_progress_log`** Log of bound improvements in the search. The output of MIP solvers is a sequence of improving incumbent solutions (primal bound) and estimates for the optimal cost (dual bound). When the costs of these two bounds match the search is concluded. In truncated searches, the most common situation for hard problems, at the end of the search there is a `gap` between these bounds. This property stores the detailed events of improving these bounds during the search process. Analyzing the evolution of these bounds you can see if you need to improve your solver w.r.t. the production of feasible solutions, by including an heuristic to produce a better initial feasible solution, for example, or improve the formulation with cutting planes, for example, to produce better dual bounds. To enable storing the `search_progress_log` set `store_search_progress_log` to True.

Return type mip.ProgressLog

property **`seed`** Random seed. Small changes in the first decisions while solving the LP relaxation and the MIP can have a large impact in the performance, as discussed in [Fisch14]. This behaviour can be exploited with multiple independent runs with different random seeds.

Return type `int`

property **`sense`** The optimization sense

Return type `str`

Returns the objective function sense, MINIMIZE (default) or (MAXIMIZE)

property **`sol_pool_size`** Maximum number of solutions that will be stored during the search. To check how many solutions were found during the search use `num_solutions()`.

Return type `int`

property **`start`** Initial feasible solution

Enters an initial feasible solution. Only the main binary/integer decision variables which appear with non-zero values in the initial feasible solution need to be informed. Auxiliary or continuous variables are automatically computed.

Return type Optional[List[Tuple[mip.Var, numbers.Real]]]

property **`status`** optimization status, which can be OPTIMAL(0), ERROR(-1), INFEASIBLE(1), UNBOUNDED(2). When optimizing problems with integer variables some additional cases may happen, FEASIBLE(3) for the case when a feasi solution was found but optimality was not proved, INT_INFEASIBLE(4) for the case when the lp relaxation is feasible but no feasible integer solution exists and NO_SOLUTION_FOUND(5) for the case when an integer solution was not found in

property **store_search_progress_log** Wether search_progress_log will be stored or not when optimizing. Default False. Activate it if you want to analyze bound improvements over time.

Return type bool

property **threads** number of threads to be used when solving the problem. 0 uses solver default configuration, -1 uses the number of available processing cores and ≥ 1 uses the specified number of threads. An increased number of threads may improve the solution time but also increases the memory consumption.

Return type int

translate(*ref*) Translates references of variables/containers of variables from another model to this model. Can be used to translate references of variables in the original model to references of variables in the pre-processed model.

Return type Union[List[Any], Dict[Any, Any], mip.Var]

validate_mip_start() Validates solution entered in MIPStart

If the solver engine printed messages indicating that the initial feasible solution that you entered in start is not valid then you can call this method to help discovering which set of variables is causing infeasibility. The current version is quite simple: the model is relaxed and one variable entered in mipstart is fixed per iteration, indicating if the model still feasible or not.

var_by_name(***name***) Searchers a variable by its name

Return type Optional[mip.Var]

Returns Variable or None if not found

property **verbose** 0 to disable solver messages printed on the screen, 1 to enable

Return type int

write(*file_path*) Saves a MIP model or an initial feasible solution.

One of the following file name extensions should be used to define the contents of what will be saved:

`.lp` mip model stored in the LP file format

`.mps` mip model stored in the MPS file format

`.sol` initial feasible solution

`.bas` optimal basis for the linear programming relaxation.

Parameters **file_path** (*str*) – file name

LinExpr

class **LinExpr**(***variables=None, coeffs=None, const=0.0, sense=''***) Linear expressions are used to enter the objective function and the model constraints. These expressions are created using operators and variables.

Consider a model object m, the objective function of `m` can be specified as:

```
m.objective = 10*x1 + 7*x4
```

In the example bellow, a constraint is added to the model

```
m += xsum(3*x[i] i in range(n)) - xsum(x[i] i in range(m))
```

A constraint is just a linear expression with the addition of a sense (==, <= or >=) and a right hand side, e.g.:

```
m += x1 + x2 + x3 == 1
```

If used in intermediate calculations, the solved value of the linear expression can be obtained with the `x` parameter, just as with a `Var`.


```
a = 10*x1 + 7*x4
print(a.x)
```

hand side

Parameters **val** (*numbers.Real*) – a real number

add_expr(*expr*, *coeff=1*) Extends a linear expression with the contents of another.

Parameters • **expr** (*LinExpr*) – another linear expression

• **coeff** (*numbers.Real*) – coefficient which will multiply the linear expression added

add_term(*term*, *coeff=1*) Adds a term to the linear expression.

Parameters • **expr** (*Union[mip.Var, LinExpr, numbers.Real]*) – can be a variable, another linear expression or a real number.

• **coeff** (*numbers.Real*) – coefficient which will multiply the added term

add_var(*var*, *coeff=1*) Adds a variable with a coefficient to the linear expression.

Parameters • **var** (*mip.Var*) – a variable

• **coeff** (*numbers.Real*) – coefficient which the variable will be added

property **const** constant part of the linear expression

Return type `Real`

equals(*other*) returns true if a linear expression equals to another, false otherwise

Return type `bool`

property **expr** the non-constant part of the linear expression

Dictionary with pairs: (variable, coefficient) where coefficient is a real number.

Return type Dict[mip.Var, numbers.Real]

property **model** Model which this LinExpr refers to, None if no variables are involved.

Return type Optional[mip.Model]

property **sense** sense of the linear expression

sense can be EQUAL(“=”), LESS_OR_EQUAL(“<”), GREATER_OR_EQUAL(“>”) or empty (“”) if this is an affine expression, such as the objective function

Return type `str`

set_expr(*expr*) Sets terms of the linear expression

Parameters **expr** (*Dict[mip.Var, numbers.Real]*) – dictionary mapping variables to their coefficients in the linear expression.

property **violation** Amount that current solution violates this constraint

If a solution is available, than this property indicates how much the current solution violates this constraint.

Return type `Optional`[`Real`]

property **x** Value of this linear expression in the solution. None is returned if no solution is available.

Return type `Optional`[`Real`]

LinExprTensor

class **LinExprTensor**

Var

class **Var**(*model*, *idx*) Decision variable of the `Model`. The creation of variables is performed calling the `add_var()`.

property **column** Variable coefficients in constraints.

Return type mip.Column

property **lb** Variable lower bound.

Return type `Real`

property name Variable name.

Return type str

property obj Coefficient of variable in the objective function.

Return type Real

property rc Reduced cost, only available after a linear programming model (only continuous variables) is optimized. Note that None is returned if no optimum solution is available

Return type Optional[Real]

property ub Variable upper bound.

Return type Real

property var_type Variable type, ('B') BINARY, ('C') CONTINUOUS and ('I') INTEGER.

Return type str

property x Value of this variable in the solution. Note that None is returned if no solution is not available.

Return type Optional[Real]

xi(*i*) Value for this variable in the i-th solution from the solution pool. Note that None is returned if the solution is not available.

Return type Optional[Real]

Constr

class Constr(*model, idx, priority=None*) A row (constraint) in the constraint matrix.

A constraint is a specific LinExpr that includes a sense (<, > or == or less-or-equal, greater-or-equal and equal, respectively) and a right-hand-side constant value. Constraints can be added to the model using the overloaded operator += or using the method add_constr() of the Model class:

```
m += 3*x1 + 4*x2 <= 5
```

summation expressions are also supported:

```
m += xsum(x[i] for i in range(n)) == 1
```

property expr Linear expression that defines the constraint.

Return type mip.LinExpr

property name constraint name

Return type str

property pi Value for the dual variable of this constraint in the optimal solution of a linear programming Model. Only available if a pure linear programming problem was solved (only continuous variables).

Return type Optional[Real]

property priority priority value

Return type ConstraintPriority

property rhs The right-hand-side (constant value) of the linear constraint.

Return type Real

property slack Value of the slack in this constraint in the optimal solution. Available only if the formulation was solved.

Return type Optional[Real]

Column

ConflictGraph

***class* ConflictGraph(*model*)** A conflict graph stores conflicts between incompatible assignments in binary variables.

For example, if there is a constraint $x_1 + x_2 \leq 1$ then there is a conflict between $x_1 = 1$ and $x_2 = 1$. We can state that x_1 and x_2 are conflicting. Conflicts can also involve the complement of a binary variable. For example, if there is a constraint $x_1 \leq x_2$ then there is a conflict between $x_1 = 1$ and $x_2 = 0$. We now can state that x_1 and $\neg x_2$ are conflicting.

conflicting(*e1*, *e2*) Checks if two assignments of binary variables are in conflict.

Parameters
- **e1** (*Union[mip.LinExpr, mip.Var]*) – binary variable, if assignment to be tested is the assignment to one, or a linear expression like x == 0 to indicate that conflict with the complement of the variable should be tested.
- **e2** (*Union[mip.LinExpr, mip.Var]*) – binary variable, if assignment to be tested is the assignment to one, or a linear expression like x == 0 to indicate that conflict with the complement of the variable should be tested.

Return type `bool`

conflicting_assignments(*v*) Returns from the conflict graph all assignments conflicting with one specific assignment.

Parameters **v** (*Union[mip.Var, mip.LinExpr]*) – binary variable, if assignment to be tested is the assignment to one or a linear expression like x == 0 to indicate the complement.

Return type Tuple[List[mip.Var], List[mip.Var]]

Returns Returns a tuple with two lists. The first one indicates variables whose conflict occurs when setting them to one. The second list includes variable whose conflict occurs when setting them to zero.

VarList

***class* VarList(*model*)** List of model variables (`Var`).

The number of variables of a model `m` can be queried as `len(m.vars)` or as `m.num_cols`.

Specific variables can be retrieved by their indices or names. For example, to print the lower bounds of the first variable or of a varible named `z`, you can use, respectively:

```
print(m.vars[0].lb)
```

```
print(m.vars['z'].lb)
```

ConstrList

***class* ConstrList(*model*)** List of problem constraints

ConstrsGenerator

***class* ConstrsGenerator** Abstract class for implementing cuts and lazy constraints generators.

generate_constrs(*model*, *depth=0*, *npass=0*) Method called by the solver engine to generate *cuts* or *lazy constraints*.

> After analyzing the contents of the solution in model variables `vars`, whose solution values can be queried with the `x` attribute, one or more constraints may be generated and added to the solver with the `add_cut()` method for cuts. This method can be called by the solver engine in two situations, in the first one a fractional solution is found and one or more inequalities can be generated (cutting planes) to remove this fractional solution. In the second case an integer feasible solution is found and then a new constraint can be generated (lazy constraint) to report that this integer solution is not feasible. To control when the constraint generator will be called set your `ConstrsGenerator` object in the attributes `cuts_generator` or `lazy_constrs_generator` (adding to both is also possible).

Parameters
- **model** (*mip.Model*) – model for which cuts may be generated. Please note that this model may have fewer variables than the original model due to pre-processing. If you want to generate cuts in terms of the original variables, one alternative is to query variables by their names, checking which ones remain in this pre-processed problem. In this procedure you can query model properties and add cuts (`add_cut()`) or lazy constraints (`add_lazy_constr()`), but you cannot perform other model modifications, such as add columns.
- **depth** (*int*) – depth of the search tree (0 is the root node)

IncumbentUpdater

class `IncumbentUpdater`(*model*) To receive notifications whenever a new integer feasible solution is found. Optionally a new improved solution can be generated (using some local search heuristic) and returned to the MIP solver.

`update_incumbent`(*objective_value*, *best_bound*, *solution*) Method that is called when a new integer feasible solution is found

Parameters **objective_value** (*float*) – cost of the new solution found

- **best_bound** (*float*) – current lower bound for the optimal solution cost
- **solution** (*List[Tuple[mip.Var,float]]*) – non-zero variables in the solution

Return type List[Tuple[mip.Var, float]]

CutType

class `CutType`(*value*) Types of cuts that can be generated. Each cut type is an implementation in the COIN-OR Cut Generation Library. For some cut types multiple implementations are available. Sometimes these implementations were designed with different objectives: for the generation of Gomory cutting planes, for example, the GMI cuts are focused on numerical stability, while Forrest's implementation (GOMORY) is more integrated into the CBC code.

`CLIQUE` = *12* Clique cuts [Padb73].

`FLOW_COVER` = *5* Lifted Simple Generalized Flow Cover Cut Generator.

`GMI` = *2* Gomory Mixed Integer cuts [Gomo69], as implemented by Giacomo Nannicini, focusing on numerically safer cuts.

`GOMORY` = *1* Gomory Mixed Integer cuts [Gomo69], as implemented by John Forrest.

`KNAPSACK_COVER` = *14* Knapsack cover cuts [Bala75].

`LATWO_MIR` = *8* Lagrangean relaxation for two-phase Mixed-integer rounding cuts, as in LAGomory

`LIFT_AND_PROJECT` = *9* Lift-and-project cuts [BCC93], implemented by Pierre Bonami.

`MIR` = *6* Mixed-Integer Rounding cuts [Marc01].

`ODD_WHEEL` = *13* Lifted odd-hole inequalities.

`PROBING` = *0* Cuts generated evaluating the impact of fixing bounds for integer variables

`RED_SPLIT` = *3* Reduce and split cuts [AGY05], implemented by Francois Margot.

`RED_SPLIT_G` = *4* Reduce and split cuts [AGY05], implemented by Giacomo Nannicini.

`RESIDUAL_CAPACITY` = *10* Residual capacity cuts [AtRa02], implemented by Francisco Barahona.

`TWO_MIR` = *7* Two-phase Mixed-integer rounding cuts.

`ZERO_HALF` = *11* Zero/Half cuts [Capr96].

CutPool

class `CutPool` `add`(*cut*) tries to add a cut to the pool, returns true if this is a new cut, false if it is a repeated one

Parameters **cut** (*mip.LinExpr*) – a constraint

Return type `bool`

OptimizationStatus

class `OptimizationStatus`(*value*) Status of the optimization

`CUTOFF` = *7* No feasible solution exists for the current cutoff

`ERROR` = *-1* Solver returned an error

`FEASIBLE` = *3* An integer feasible solution was found during the search but the search was interrupted before concluding if this is the optimal solution or not.

`INT_INFEASIBLE` = *4* A feasible solution exist for the relaxed linear program but not for the problem with existing integer variables

`LOADED` = *6* The problem was loaded but no optimization was performed

`NO_SOLUTION_FOUND` = *5* A truncated search was executed and no integer feasible solution was found

`OPTIMAL` = *0* Optimal solution was computed

`UNBOUNDED` = *2* One or more variables that appear in the objective function are not included in binding constraints and the optimal objective value is infinity.

SearchEmphasis

class `SearchEmphasis`(***value***) An enumeration.

`DEFAULT` = *0* Default search emphasis, try to balance between improving the dual bound and producing integer feasible solutions.

`FEASIBILITY` = *1* More aggressive search for feasible solutions.

`OPTIMALITY` = *2* Focuses more on producing improved dual bounds even if the production of integer feasible solutions is delayed.

LP_Method

class `LP_Method`(***value***) Different methods to solve the linear programming problem.

`AUTO` = *0* Let the solver decide which is the best method

`BARRIER` = *3* The barrier algorithm

`DUAL` = *1* The dual simplex algorithm

`PRIMAL` = *2* The primal simplex algorithm

ProgressLog

class `ProgressLog` Class to store the improvement of lower and upper bounds over time during the search. Results stored here are useful to analyze the performance of a given formulation/parameter setting for solving a instance. To be able to automatically generate summarized experimental results, fill the `instance` and `settings` of this object with the instance name and formulation/parameter setting details, respectively.

`log` List of tuples in the format $(time, (lb, ub))$, where $time$ is the processing time in seconds and lb and ub are the lower and upper bounds, respectively

Type List[Tuple[float, Tuple[float, float]]]

`instance` instance name

Type str

`settings` identification of the formulation/parameter settings used in the optimization (whatever is relevant to identify a given computational experiment)

Type str

`read`(***file_name***) Reads a progress log stored in a file

`write`(***file_name=''***) Saves the progress log. If no extension is informed, the `.plog` extension will be used. If only a directory is informed then the name will be built considering the `instance` and `settings` attributes

Exceptions

class `MipBaseException` Base class for all exceptions specific to Python MIP. Only sub-classes of this exception are raised. Inherits from the Python builtin `Exception`.

class `ProgrammingError` Exception that is raised when the calling program performs an invalid or nonsensical operation. Inherits from `mip.MipBaseException`.

class **InvalidLinExpr** Exception that is raised when an invalid linear expression is created. Inherits from mip.MipBaseException.

class **InvalidParameter** Exception that is raised when an invalid/non-existent parameter is used or set. Inherits from mip.MipBaseException.

class **ParameterNotAvailable** Exception that is raised when some parameter is not available or can not be set. Inherits from mip.MipBaseException.

class **InfeasibleSolution** Exception that is raised the produced solution is unfeasible. Inherits from mip.MipBaseException.

class **SolutionNotAvailable** Exception that is raised when a method that requires a solution is queried but the solution is not available. Inherits from mip.MipBaseException.

Useful functions

minimize(*objective*) Function that should be used to set the objective function to MINIMIZE a given linear expression (passed as argument).

Parameters **objective** (*Union[mip.LinExpr, Var]*) – linear expression

Return type mip.LinExpr

maximize(*objective*) Function that should be used to set the objective function to MAXIMIZE a given linear expression (passed as argument).

Parameters **objective** (*Union[mip.LinExpr, Var]*) – linear expression

Return type mip.LinExpr

xsum(*terms*) Function that should be used to create a linear expression from a summation. While the python function sum() can also be used, this function is optimized version for quickly generating the linear expression.

Parameters **terms** – set (ideally a list) of terms to be summed

Return type mip.LinExpr

compute_features(*model*) This function computes instance features for a MIP. Features are instance characteristics, such as number of columns, rows, matrix density, etc. These features can be used in machine learning algorithms to recommend parameter settings. To check names of features that are computed in this vector use features()

Parameters **model** (*Model*) – the MIP model were features will be extracted

Return type List[float]

features() This function returns the list of problem feature names that can be computed compute_features()

Return type List[str]