Mixed Integer Linear Programming with Python

Haroldo G. Santos Túlio A.M. Toffolo

Oct 09, 2019

Contents:

Chapter 1

Introduction

The Python-MIP package provides tools for modeling and solving Mixed Integer Linear Programming Problems (MIPs) [Jung09] in Python. The default installation includes the COIN-OR Linear Programming Solver - CLP, which is currently the fastest open source linear programming solver and the COIN-OR Branch-and-Cut solver - CBC, a highly configurable MIP solver. It also works with the state-of-the-art Gurobi MIP solver. Python-MIP was written in modern, statically typed Python and works with the fast just-in-time Python compiler Pypy.

In the modeling layer, models can be written very concisely, as in high-level mathematical programming languages such as MathProg. Modeling examples for some applications can be viewed in *Chapter 3*.

Python-MIP eases the development of high-performance MIP based solvers for custom applications by providing a tight integration with the branch-and-cut algorithms of the supported solvers. Strong formulations with an exponential number of constraints can be handled by the inclusion of *Cut Generators*. Heuristics can be integrated for *providing initial feasible solutions* to the MIP solver. These features can be used in both solver engines, CBC and GUROBI, without changing a single line of code.

This document is organized as follows: in the *next Chapter* installation and configuration instructions for different platforms are presented. In *Chapter 3* an overview of some common model creation and optimization code included. Commented examples are included in *Chapter 4*. *Chapter 5* includes some common solver customizations that can be done to improve the performance of application specific solvers. Finally, the detailed reference information for the main classes is included in *Chapter 6*.

1.1 Acknowledgments

We would like to thank for the support of the Combinatorial Optimization and Decision Support (CODeS) research group in KU Leuven through the senior research fellowship of Prof. Haroldo in 2018-2019, CNPq "Produtividade em Pesquisa" grant, FAPEMIG and the GOAL research group in the Computing Department of UFOP.

Chapter 2

Installation

Python-MIP requires Python 3.5 or newer. Since Python-MIP is included in the Python Package Index, once you have a Python installation, installing it is as easy as entering, in the command prompt:

```
pip install mip
```

If the command fails, it may be due to lack of permission to install globally available Python modules. In this case, use:

```
pip install mip --user
```

The default installation includes re-compiled libraries of the MIP Solver CBC for Windows, Linux and MacOS. If you have the commercial solver Gurobi installed in your computer, Python-MIP will automatically use it as long as it finds the Gurobi dynamic loadable library. Gurobi is free for academic use and has an outstanding performance for solving MIPs. Instructions to make it accessible on different operating systems are included bellow.

2.1 Gurobi Installation and Configuration (optional)

2.1.1 Linux

Linux Gurobi installation notes are available here. In Linux, the Gurobi dynamic loadable library file is `libgurobixx.so`, where `xx` stands for Gurobi's version. You must add the library installation directory to the `/etc/ld.so.conf` file and call `ldconfig` after to make the library visible for all applications. If Gurobi was installed in `/opt/gurobi810` then you would have to add `/opt/gurobi810/linux64/lib/` to `/etc/ld.so.conf`. Since this is a system wide configuration file, you will require super user permission to modify it. To add this Path as a configuration only for your user account, thus not requiring super user privileges, enter this command before re-starting your session:

```
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/gurobi810/linux64/lib/' >> ~/.profile
```

2.1.2 Windows

Windows Gurobi installation notes are available here. In Windows, Gurobi's dynamically loadable library is the file `gurobixx.dll`, where `xx` stands for Gurobi's version. Be sure to set your Path environment variable to include the installation folder of this file.

2.2 Pypy installation (optional)

Python-MIP is compatible with the just-in-time Python compiler Pypy. Generally, Python code executes much faster in Pypy. Pypy is also more memory efficient. To install Python-MIP as a Pypy package, just call:

```
pypy3 -m pip install mip
```

Chapter 3

Quick start

This chapter presents the main components needed to build and optimize models using Python-MIP. A full description of the methods and their parameters can be found at *Chapter 4*.

The first step to enable Python-MIP in your Python code is to add:

```
from mip.model import *
```

When loaded, Python-MIP will display its installed version:

```
Using Python-MIP package version 1.4.2
```

3.1 Creating Models

The model class represents a the optimization model. The code below creates an empty Mixed Integer Linear Programming problem with default settings.

```
m = Model()
```

By default, the optimization sense is set to *Minimize* and the selected solver is set to CBC. In case Gurobi is installed and configured, it will be used instead. You can change the model objective sense or force the selection of a specific solve using additional parameters for the constructor:

```
m = Model(sense=MAXIMIZE, solver_name=CBC) # use GRB for Gurobi
```

After creating the model, you should include your decision variables, objective function and constraints. These tasks will be discussed in the next sections.

3.1.1 Variables

Decision variables are added to the model using the *add_var()* method. Without parameters, a single variable with domain in $\mathbb{R}^+$ is created and its reference is returned:

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

By using Python list initialization syntax, you can easily create a vector of variables. Let's say that your model will have n binary decision variables (n=10 in the example below) indicating if an items is selected or not. The code below creates 10 binary variables `y[0]`, ..., `y[n-1]`.

```
n = 10
y = [ m.add_var(var_type=BINARY) for i in range(n) ]
```

Additional variable types are `CONTINUOUS` (default) and `INTEGER`. Some additional properties that can be specified for variables are their lower and upper bounds (*lb* and *ub*, respectively), and names (property *name*). Naming a variable is optional but particularly useful if you plan to save you model (see *Saving, Loading and Checking Model Properties*) in .LP or .MPS file formats, for instance. The following code creates an integer variable named `zCost` which is restricted to be in range $\{-10, \ldots, 10\}$. Note that the variable's reference is stored in a Python variable named `z`.

```
z = m.add_var(name='zCost', var_type=INTEGER, lb=-10, ub=10)
```

You don't need to store references for variables, even though it is usually easier to do so to write constraints. If you do not store these references, you can get them afterwards using the Model function *var_by_name()*. The following code retrieves the reference of a variable named `zCost` and sets its upper bound to 5:

```
vz = m.var_by_name('zCost')
vz.ub = 5
```

3.1.2 Constraints

Constraints are linear expressions involving variables, a sense of ==, <= or >= for equal, less or equal and greater or equal, respectively, and a constant. The constraint $x + y \leq 10$ can be easily included within model `m`:

```
m += x + y <= 10
```

Summation expressions can be implemented with the function *xsum()*. If for a knapsack problem with n items, each one with weight w_i, we would like to include a constraint to select items with binary variables x_i respecting the knapsack capacity c, then the following code could be used to include this constraint within the model `m`:

```
m += xsum(w[i]*x[i] for i in range(n)) <= c
```

Conditional inclusion of variables in the summation is also easy. Let's say that only even indexed items are subjected to the capacity constraint:

```
m += xsum(w[i]*x[i] for i in range(n) if i%2 == 0) <= c
```

Finally, it may be useful to name constraints. To do so is straightforward: include the constraint's name after the linear expression, separating it with a comma. An example is given below:

```
m += xsum(w[i]*x[i] for i in range(n) if i%2 == 0) <= c, 'even_sum'
```

As with variables, reference of constraints can be retrieved by their names. Model function *constr_by_name()* is responsible for this:

```
constraint = m.constr_by_name('even_sum')
```

3.1.3 Objective Function

By default a model is created with the *Minimize* sense. The following code alters the objective function to $\sum_{i=0}^{n-1} c_i x_i$ by setting the *objective* attribute of our example model `m`:

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

To specify whether the goal is to *Minimize* or *Maximize* the objetive function, two useful functions were included: *minimize()* and *maximize()*. Below are two usage examples:

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

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

You can also change the optimization direction by setting the *sense* model property to `MINIMIZE` or `MAXIMIZE`.

3.2 Saving, Loading and Checking Model Properties

Model methods *write()* and *read()* can be used to save and load, respectively, MIP models. Supported file formats for models are the LP file format, which is more readable and suitable for debugging, and the MPS file format, which is recommended for extended compatibility, since it is an older and more widely adopted format. When calling the *write()* method, the file name extension (.lp or .mps) is used to define the file format. Therefore, to save a model `m` using the lp file format to the file model.lp we can use:

```
m.write('model.lp')
```

Likewise, we can read a model, which results in creating variables and constraints from the LP or MPS file read. Once a model is read, all its attributes become available, like the number of variables, constraints and non-zeros in the constraint matrix:

```
m.read('model.lp')
print('model has {} vars, {} constraints and {} nzs'.format(m.num_cols, m.num_rows, m.num_nz))
```

3.3 Optimizing and Querying Optimization Results

MIP solvers execute a Branch-&-Cut (BC) algorithm that in *finite time* will provide the optimal solution. This time may be, in many cases, too large for your needs. Fortunately, even when the complete tree search is too expensive, results are often available in the beginning of the search. Sometimes a feasible solution is produced when the first tree nodes are processed and a lot of additional effort is spent improving the *dual bound*, which is a valid estimate for the cost of the optimal solution. When this estimate, the lower bound for minimization, matches exactly the cost of the best solution found, the upper bound, the search is concluded. For practical applications, usually a truncated search is executed. The *optimize()* method, that executes the optimization of a formulation, accepts optionally processing limits as parameters. The following code executes the branch-&-cut algorithm to solve a model `m` for up to 300 seconds.

```
1  m.max_gap = 0.05
2  status = m.optimize(max_seconds=300)
3  if status == OptimizationStatus.OPTIMAL:
4      print('optimal solution cost {} found'.format(m.objective_value))
5  elif status == OptimizationStatus.FEASIBLE:
6      print('sol.cost {} found, best possible: {}'.format(m.objective_value, m.objective_bound))
7  elif status == OptimizationStatus.NO_SOLUTION_FOUND:
8      print('no feasible solution found, lower bound is: {}'.format(m.objective_bound))
9  if status == OptimizationStatus.OPTIMAL or status == OptimizationStatus.FEASIBLE:
10     print('solution:')
11     for v in m.vars:
12        if abs(v.x) > 1e-6: # only printing non-zeros
13           print('{} : {}'.format(v.name, v.x))
```

Additional processing limits may be used: `max_nodes` restricts the maximum number of explored nodes in the search tree and `max_solutions` finishes the BC algorithm after a number of feasible solutions are obtained. It is also wise to specify how tight the bounds should be to conclude the search. The model

attribute `max_gap` specifies the allowable percentage deviation of the upper bound from the lower bound for concluding the search. In our example, whenever the distance of the lower and upper bounds is less or equal 5% (see line 1), the search can be finished.

The `optimize` method returns the status (*OptimizationStatus*) of the BC search: `OPTIMAL` if the search was concluded and the optimal solution was found; `FEASIBLE` if a feasible solution was found but there was no time to prove whether the current solution was optimal or not; `NO_SOLUTION_FOUND` if in the truncated search no solution was found; `INFEASIBLE` or `INT_INFEASIBLE` if no feasible solution exists for the model; `UNBOUNDED` if there are missing constraints or `ERROR` if some error occurred during optimization. In the example above, if a feasible solution is available (line 8), variables which have value different from zero are printed. Observe also that even when no feasible solution is available the lower bound is available (line 8). If a truncated execution was performed, i.e., the solver stopped due to the time limit, you can check an estimate of the quality of the solution found checking the *gap* property.

During the tree search, it is often the case that many different feasible solutions are found. The solver engine stores this solutions in a solution pool. The following code prints all routes found while optimizing the *Traveling Salesman Problem*.

```
for k in range(model.num_solutions):
    print('route {} with length {}'.format(k, model.objective_values[k]))
    for (i, j) in product(range(n), range(n)):
        if x[i][j].xi(k) >= 0.98:
            print('\tarc ({},{})'.format(i,j))
```

3.3.1 Performance Tuning

Tree search algorithms of MIP solvers deliver a set of improved feasible solutions and lower bounds. Depending on your application you will be more interested in the quick production of feasible solutions than in improved lower bounds that may require expensive computations, even if in the long term these computations prove worthy to prove the optimality of the solution found. The model property *emphasis* provides three different settings:

0. **default setting:** tries to balance between the search of improved feasible solutions and improved lower bounds;
1. **feasibility:** focus on finding improved feasible solutions in the first moments of the search process, activates heuristics;
2. **optimality:** activates procedures that produce improved lower bounds, focusing in pruning the search tree even if the production of the first feasible solutions is delayed.

Changing this setting to 1 or 2 triggers the activation/deactivation of several algorithms that are processed at each node of the search tree that impact the solver performance. Even though in average these settings change the solver performance as described previously, depending on your formulation the impact of these changes may be very different and it is usually worth to check the solver behavior with these different settings in your application.

Another parameter that may be worth tuning is the *cuts* attribute, that controls how much computational effort should be spent in generating cutting planes.

Chapter 4

Modeling Examples

This chapter includes commented examples on modeling and solving optimization problems with Python-MIP.

4.1 The 0/1 Knapsack Problem

As a first example, consider the solution of the 0/1 knapsack problem: given a set I of items, each one with a weight w_i and estimated profit p_i, one wants to select a subset with maximum profit such that the summation of the weights of the selected items is less or equal to the knapsack capacity c. Considering a set of decision binary variables x_i that receive value 1 if the i-th item is selected, or 0 if not, the resulting mathematical programming formulation is:

$$\text{Maximize:} \quad \sum_{i \in I} p_i \cdot x_i$$

$$\text{Subject to:} \quad \sum_{i \in I} w_i \cdot x_i \leq c$$

$$x_i \in \{0, 1\} \ \ \forall i \in I$$

The following python code creates, optimizes and prints the optimal solution for the 0/1 knapsack problem

Listing 1: Solves the 0/1 knapsack problem: knapsack.py

```
"""0/1 Knapsack example"""

from mip import Model, xsum, maximize, BINARY

p = [10, 13, 18, 31, 7, 15]
w = [11, 15, 20, 35, 10, 33]
c = 47
n = len(w)

m = Model('knapsack')

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

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

m += xsum(w[i] * x[i] for i in range(n)) <= c
```

(continues on next page)

(continued from previous page)

```
17
18 m.optimize()
19
20 selected = [i for i in range(n) if x[i].x >= 0.99]
21 print('selected items: {}'.format(selected))
```

Line 3 imports the required classes and definitions from Python-MIP. Lines 5-8 define the problem data. Line 10 creates an empty maximization problem `m` with the (optional) name of "knapsack". Line 12 adds the binary decision variables to model `m` and stores their references in a list `x`. Line 14 defines the objective function of this model and line 16 adds the capacity constraint. The model is optimized in line 18 and the solution, a list of the selected items, is computed at line 20.

4.2 The Traveling Salesman Problem

The traveling salesman problem (TSP) is one of the most studied combinatorial optimization problems, with the first computational studies dating back to the 50s [DFJ54][ABCC06]. To to illustrate this problem, consider that you will spend some time in Belgium and wish to visit some of its main tourist attractions, depicted in the map bellow:

You want to find the shortest possible tour to visit all these places. More formally, considering n points $V = \{0, \ldots, n-1\}$ and a distance matrix $D_{n \times n}$ with elements $c_{i,j} \in \mathbb{R}^+$, a solution consists in a set of exactly n (origin, destination) pairs indicating the itinerary of your trip, resulting in the following formulation:

Minimize:

$$\sum_{i \in I, j \in I} c_{i,j} . x_{i,j}$$

Subject to:

$$\sum_{j \in V \setminus \{i\}} x_{i,j} = 1 \;\; \forall i \in V$$

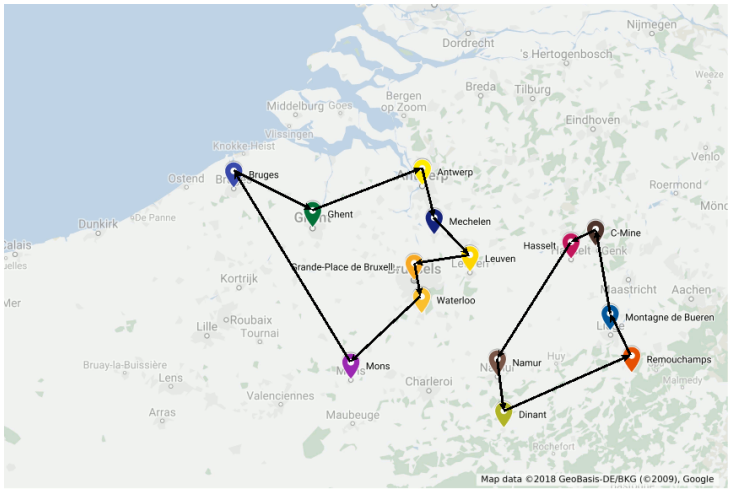
$$\sum_{i \in V \setminus \{j\}} x_{i,j} = 1 \;\; \forall j \in V$$

$$y_i - (n+1) . x_{i,j} \geq y_j - n \;\; \forall i \in V \setminus \{0\}, j \in V \setminus \{0, i\}$$

$$x_{i,j} \in \{0, 1\} \;\; \forall i \in V, j \in V$$

$$y_i \geq 0 \;\; \forall i \in V$$

The first two sets of constraints enforce that we leave and arrive only once at each point. The optimal solution for the problem including only these constraints could result in a solution with sub-tours, such as the one bellow.



To enforce the production of connected routes, additional variables $y_i \geq 0$ are included in the model indicating the sequential order of each point in the produced route. Point zero is arbitrarily selected as the initial point and conditional constraints linking variables $x_{i,j}$, y_i and y_j are created for all nodes except the the initial one to ensure that the selection of the arc $x_{i,j}$ implies that $y_j \geq y_i + 1$.

The Python code to create, optimize and print the optimal route for the TSP is included bellow:

Listing 2: Traveling salesman problem solver with compact formulation: tsp-compact.py

```
"""Example that solves the Traveling Salesman Problem using the simple compact
formulation presented in Miller, C.E., Tucker, A.W and Zemlin, R.A. "Integer
Programming Formulation of Traveling Salesman Problems". Journal of the ACM
7(4). 1960."""

from itertools import product
from sys import stdout as out
from mip import Model, xsum, minimize, BINARY

# names of places to visit
places = ['Antwerp', 'Bruges', 'C-Mine', 'Dinant', 'Ghent',
          'Grand-Place de Bruxelles', 'Hasselt', 'Leuven',
          'Mechelen', 'Mons', 'Montagne de Bueren', 'Namur',
          'Remouchamps', 'Waterloo']

# distances in an upper triangular matrix
dists = [[83, 81, 113, 52, 42, 73, 44, 23, 91, 105, 90, 124, 57],
         [161, 160, 39, 89, 151, 110, 90, 99, 177, 143, 193, 100],
         [90, 125, 82, 13, 57, 71, 123, 38, 72, 59, 82],
         [123, 77, 81, 71, 91, 72, 64, 24, 62, 63],
         [51, 114, 72, 54, 69, 139, 105, 155, 62],
         [70, 25, 22, 52, 90, 56, 105, 16],
         [45, 61, 111, 36, 61, 57, 70],
         [23, 71, 67, 48, 85, 29],
         [74, 89, 69, 107, 36],
         [117, 65, 125, 43],
         [54, 22, 84],
         [60, 44],
         [97],
         []]

# number of nodes and list of vertices
n, V = len(dists), range(len(dists))

# distances matrix
```

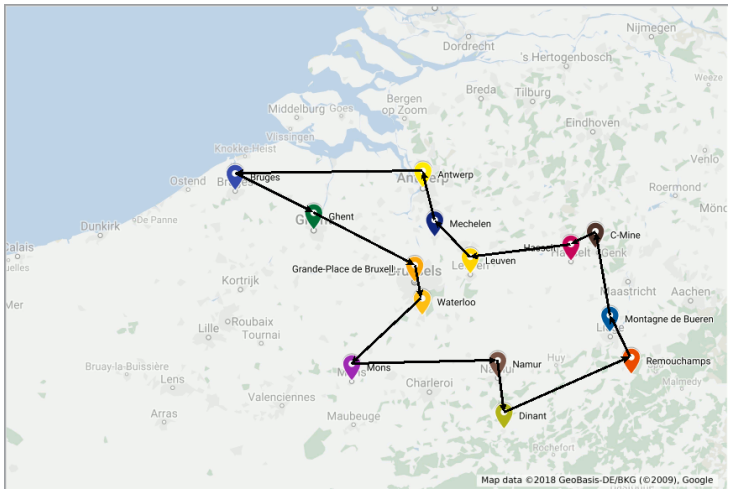
(continues on next page)

(continued from previous page)

```
36 c = [[0 if i == j
37       else dists[i][j-i-1] if j > i
38       else dists[j][i-j-1]
39       for j in V] for i in V]
40
41 model = Model()
42
43 # binary variables indicating if arc (i,j) is used on the route or not
44 x = [[model.add_var(var_type=BINARY) for j in V] for i in V]
45
46 # continuous variable to prevent subtours: each city will have a
47 # different sequential id in the planned route except the first one
48 y = [model.add_var() for i in V]
49
50 # objective function: minimize the distance
51 model.objective = minimize(xsum(c[i][j]*x[i][j] for i in V for j in V))
52
53 # constraint : leave each city only once
54 for i in V:
55     model += xsum(x[i][j] for j in set(V) - {i}) == 1
56
57 # constraint : enter each city only once
58 for i in V:
59     model += xsum(x[j][i] for j in set(V) - {i}) == 1
60
61 # subtour elimination
62 for (i, j) in set(product(set(V) - {0}, set(V) - {0})):
63         model += y[i] - (n+1)*x[i][j] >= y[j]-n
64
65 # optimizing
66 model.optimize(max_seconds = 30)
67
68 # checking if a solution was found
69 if model.num_solutions:
70     out.write('route with total distance %g found: %s' % (model.objective_value, places[0]))
71     nc = 0
72     while True:
73         nc = [i for i in V if x[nc][i].x >= 0.99][0]
74         out.write(' -> %s' % places[nc])
75         if nc == 0:
76             break
77     out.write('\n')
```

In line 10 names of the places to visit are informed. In line 17 distances are informed in an upper triangular matrix. Line 33 stores the number of nodes and a list with nodes sequential ids starting from 0. In line 36 a full $n \times n$ distance matrix is filled. Line 41 creates an empty MIP model. In line 44 all binary decision variables for the selection of arcs are created and their references are stored a $n \times n$ matrix named `x`. Differently from the x variables, y variables (line 48) are not required to be binary or integral, they can be declared just as continuous variables, the default variable type. In this case, the parameter `var_type` can be omitted from the `add_var` call.

Line 51 sets the total traveled distance as objective function and lines 54-62 include the constraints. In line 66 we call the optimizer specifying a time limit of 30 seconds. This will surely not be necessary for our Belgium example, which will be solved instantly, but may be important for larger problems: even though high quality solutions may be found very quickly by the MIP solver, the time required to *prove* that the current solution is optimal may be very large. With a time limit, the search is truncated and the best solution found during the search is reported. In line 69 we check for the availability of a feasible solution. To repeatedly check for the next node in the route we check for the solution value (`.x` attribute) of all variables of outgoing arcs of the current node in the route (line 73). The optimal solution for our trip has length 547 and is depicted bellow:



4.3 n-Queens

In the n-queens puzzle n chess queens should to be placed in a board with $n \times n$ cells in a way that no queen can attack another, i.e., there must be at most one queen per row, column and diagonal. This is a constraint satisfaction problem: any feasible solution is acceptable and no objective function is defined. The following binary programming formulation can be used to solve this problem:

$$\sum_{j=1}^{n} x_{ij} = 1 \;\; \forall i \in \{1,\ldots,n\}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \;\; \forall j \in \{1,\ldots,n\}$$

$$\sum_{i=1}^{n} \sum_{j=1:i-j=k}^{n} x_{i,j} \leq 1 \;\; \forall i \in \{1,\ldots,n\}, k \in \{2-n,\ldots,n-2\}$$

$$\sum_{i=1}^{n} \sum_{j=1:i+j=k}^{n} x_{i,j} \leq 1 \;\; \forall i \in \{1,\ldots,n\}, k \in \{3,\ldots,n+n-1\}$$

$$x_{i,j} \in \{0,1\} \;\; \forall i \in \{1,\ldots,n\}, j \in \{1,\ldots,n\}$$

The following code builds the previous model, solves it and prints the queen placements:

Listing 3: Solver for the n-queens problem: queens.py

```
"""Example of a solver to the n-queens problem:  n chess queens should be
placed in a n x n chess board so that no queen can attack another, i.e., just
one queen per line, column and diagonal.  """

from sys import stdout
from mip import Model, xsum, MAXIMIZE, BINARY

# number of queens
n = 75

queens = Model()

x = [[queens.add_var('x({},{})'.format(i, j), var_type=BINARY)
      for j in range(n)] for i in range(n)]

# one per row
for i in range(n):
```

(continues on next page)

(continued from previous page)

```
    queens += xsum(x[i][j] for j in range(n)) == 1, 'row({})'.format(i)

# one per column
for j in range(n):
    queens += xsum(x[i][j] for i in range(n)) == 1, 'col({})'.format(j)

# diagonal \
for p, k in enumerate(range(2 - n, n - 2 + 1)):
    queens += xsum(x[i][j] for i in range(n) for j in range(n)
                   if i - j == k) <= 1, 'diag1({})'.format(p)

# diagonal /
for p, k in enumerate(range(3, n + n)):
    queens += xsum(x[i][j] for i in range(n) for j in range(n)
                   if i + j == k) <= 1, 'diag2({})'.format(p)

queens.optimize()

if queens.num_solutions:
        stdout.write('\n')
        for i, v in enumerate(queens.vars):
                stdout.write('0 ' if v.x >= 0.99 else '. ')
                if i % n == n-1:
                        stdout.write('\n')
```

4.4 Frequency Assignment

The design of wireless networks, such as cell phone networks, involves assigning communication frequencies to devices. These communication frequencies can be separated into channels. The geographical area covered by a network can be divided into hexagonal cells, where each cell has a base station that covers a given area. Each cell requires a different number of channels, based on usage statistics and each cell has a set of neighbor cells, based on the geographical distances. The design of an efficient mobile network involves selecting subsets of channels for each cell, avoiding interference between calls in the same cell and in neighboring cells. Also, for economical reasons, the total bandwidth in use must be minimized, i.e., the total number of different channels used. One of the first real cases discussed in literature are the Philadelphia [And73] instances, with the structure depicted bellow:

Each cell has a demand with the required number of channels drawn at the center of the hexagon, and a sequential id at the top left corner. Also, in this example, each cell has a set of at most 6 adjacent neighboring cells (distance 1). The largest demand (8) occurs on cell 2. This cell has the following adjacent cells, with distance 1: (1, 6). The minimum distances between channels in the same cell in this example is 3 and channels in neighbor cells should differ by at least 2 units.

A generalization of this problem (not restricted to the hexagonal topology), is the Bandwidth Multicoloring Problem (BMCP), which has the following input data:

N**:** set of cells, numbered from 1 to n;

$r_i \in \mathbb{Z}^+$**:** demand of cell $i \in N$, i.e., the required number of channels;

$d_{i,j} \in \mathbb{Z}^+$**:** minimum distance between channels assigned to nodes i and j, $d_{i,i}$ indicates the minimum distance between different channels allocated to the same cell.

Given an upper limit $\overline{u}$ on the maximum number of channels $U = \{1, \ldots, \overline{u}\}$ used, which can be obtained using a simple greedy heuristic, the BMPC can be formally stated as the combinatorial optimization problem of defining subsets of channels $C_1, \ldots, C_n$ while minimizing the used bandwidth and avoiding

interference:

$$\begin{aligned}
&\text{Minimize:}\\
&\qquad \max_{c \in C_1 \cup C_2, \ldots, C_n} c\\
&\text{Subject to:}\\
&\mid c_1 - c_2 \mid \geq d_{i,j} \;\; \forall (i,j) \in N \times N, (c_1, c_2) \in C_i \times C_j\\
&\qquad C_i \subseteq U \;\; \forall i \in N\\
&\qquad \mid C_i \mid = r_i \;\; \forall i \in N
\end{aligned}$$

This problem can be formulated as a mixed integer program with binary variables indicating the composition of the subsets: binary variables $x_{(i,c)}$ indicate if for a given cell i channel c is selected ($x_{(i,c)} = 1$) or not ($x_{(i,c)} = 0$). The BMCP can be modeled with the following MIP formulation:

$$\begin{aligned}
&\text{Minimize:}\\
&\qquad z\\
&\text{Subject to:}\\
&\sum_{c=1}^{\overline{u}} x_{(i,c)} = r_i \;\; \forall i \in N\\
&\qquad z \geq c \cdot x_{(i,c)} \;\; \forall i \in N, c \in U\\
&x_{(i,c)} + x_{(j,c')} \leq 1 \;\; \forall (i,j,c,c') \in N \times N \times U \times U : i \neq j \wedge \mid c - c' \mid < d_{(i,j)}\\
&x_{(i,c} + x_{(i,c')} \leq 1 \;\; \forall i, c \in N \times U, c' \in \{c, +1 \ldots, \min(c + d_{i,i}, \overline{u})\}\\
&\qquad x_{(i,c)} \in \{0,1\} \;\; \forall i \in N, c \in U\\
&\qquad z \geq 0
\end{aligned}$$

Follows the example of a solver for the BMCP using the previous MIP formulation:

Listing 4: Solver for the bandwidth multi coloring problem: bmcp.py

```
"""Bandwidth multi coloring problem, more specificially the Frequency
assignment problem as described here: http://fap.zib.de/problems/Philadelphia/
"""

from itertools import product
from mip import Model, xsum, minimize, BINARY

# number of channels per node
r = [3, 5, 8, 3, 6, 5, 7, 3]

# distance between channels in the same node (i, i) and in adjacent nodes
#      0  1  2  3  4  5  6  7
d = [[3, 2, 0, 0, 2, 2, 0, 0],   # 0
     [2, 3, 2, 0, 0, 2, 2, 0],   # 1
     [0, 2, 3, 0, 0, 0, 3, 0],   # 2
     [0, 0, 0, 3, 2, 0, 0, 2],   # 3
     [2, 0, 0, 2, 3, 2, 0, 0],   # 4
     [2, 2, 0, 0, 2, 3, 2, 0],   # 5
     [0, 2, 2, 0, 0, 2, 3, 0],   # 6
     [0, 0, 0, 2, 0, 0, 0, 3]]   # 7

N = range(len(r))

# in complete applications this upper bound should be obtained from a feasible
# solution produced with some heuristic
U = range(sum(d[i][j] for (i, j) in product(N, N)) + sum(el for el in r))

m = Model()
```

(continues on next page)

(continued from previous page)

```
29
30 x = [[m.add_var('x({},{})'.format(i, c), var_type=BINARY)
31       for c in U] for i in N]
32
33 z = m.add_var('z')
34 m.objective = minimize(z)
35
36 for i in N:
37     m += xsum(x[i][c] for c in U) == r[i]
38
39 for i, j, c1, c2 in product(N, N, U, U):
40     if i != j and c1 <= c2 < c1+d[i][j]:
41         m += x[i][c1] + x[j][c2] <= 1
42
43 for i, c1, c2 in product(N, U, U):
44     if c1 < c2 < c1+d[i][i]:
45         m += x[i][c1] + x[i][c2] <= 1
46
47 for i, c in product(N, U):
48     m += z >= (c+1)*x[i][c]
49
50 m.optimize(max_seconds=100)
51
52 if m.num_solutions:
53     for i in N:
54         print('Channels of node %d: %s' % (i, [c for c in U if x[i][c].x >=
55                                               0.99]))
```

4.5 Resource Constrained Project Scheduling

The Resource-Constrained Project Scheduling Problem (RCPSP) is a combinatorial optimization problem that consists of finding a feasible scheduling for a set of n jobs subject to resource and precedence constraints. Each job has a processing time, a set of successors jobs and a required amount of different resources. Resources are scarce but are renewable at each time period. Precedence constraints between jobs mean that no jobs may start before all its predecessors are completed. The jobs must be scheduled non-preemptively, i.e., once started, their processing cannot be interrupted.

The RCPSP has the following input data:

$\mathcal{J}$ jobs set

$\mathcal{R}$ renewable resources set

$\mathcal{S}$ set of precedences between jobs $(i, j) \in \mathcal{J} \times \mathcal{J}$

$\mathcal{T}$ planning horizon: set of possible processing times for jobs

p_j processing time of job j

$u_{(j,r)}$ amount of resource r required for processing job j

c_r capacity of renewable resource r

In addition to the jobs that belong to the project, the set $\mathcal{J}$ contains the jobs x_0 and x_{n+1}. These jobs are dummy jobs and represent the beginning of the planning and the end of the planning. The processing time for the dummy jobs is zero and does not consume resources.

A binary programming formulation was proposed by Pritsker et al. [PWW69]. In this formulation, decision variables $x_{jt} = 1$ if job j is assigned a completion time at the end of time t; otherwise, $x_{jt} = 0$. All jobs must finish in a single instant of time without violating the relationships of precedence and

amount of available resources. The model proposed by Pristker can be stated as follows:

$$\text{Minimize}$$

$$\sum_{t \in \mathcal{T}} (t-1).x_{(n+1,t)}$$

$$\text{Subject to:}$$

$$\sum_{t \in \mathcal{T}} x_{(j,t)} = 1 \;\; \forall j \in J \cup \{n+1\}$$

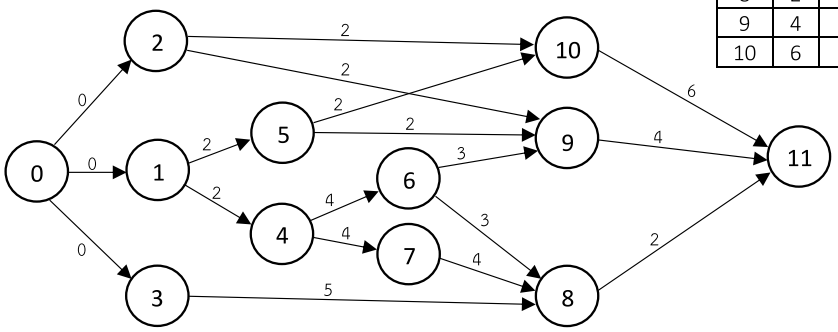
$$\sum_{j \in J} \sum_{t'=t-p_j+1} u_{(j,r)} x_{(j,t')} \leq c_r \;\; \forall t \in \mathcal{T}, r \in R$$

$$\sum_{t \in \mathcal{T}} t.x_{(s,t)} - \sum_{t \in \mathcal{T}} t.x_{(j,t)} \geq p_j \;\; \forall (j,s) \in S$$

$$x_{(j,t)} \in \{0,1\} \;\; \forall j \in J \cup \{n+1\}, t \in \mathcal{T}$$

An instance is shown below. The figure shows a graph where jobs $\mathcal{J}$ are represented by nodes and precedence relations $\mathcal{S}$ are represented by directed edges. Arc weights represent the time-consumption p_j, while the information about resource consumption $u_{(j,r)}$ is included next to the graph. This instance contains 10 jobs and 2 renewable resources ($\mathcal{R} = \{r_1, r_2\}$), where $c_1 = 6$ and $c_2 = 8$. The time horizon $\mathcal{T}$ can be estimated by summing the duration of all jobs.

Job	p_j	$u_{(j,1)}$	$u_{(j,2)}$
1	3	5	1
2	2	0	4
3	5	1	4
4	4	1	3
5	2	3	2
6	3	3	1
7	4	2	4
8	2	4	0
9	4	5	2
10	6	2	5



The Python code for creating the binary programming model, optimize it and print the optimal scheduling for RCPSP is included below:

Listing 5: Solves the Resource Constrained Project Scheduling Problem: rcpsp.py

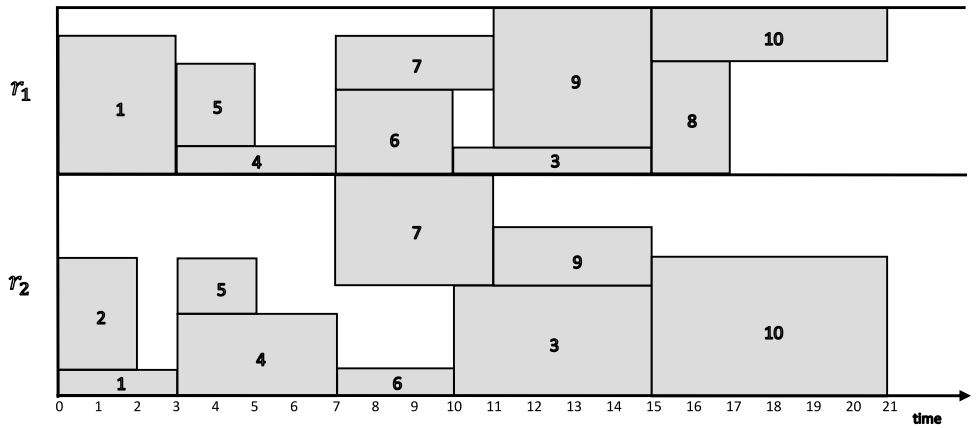
```
"""Resource Constrained Project Scheduling Problem solver"""

from itertools import product
from mip import Model, xsum, BINARY

p = [0, 3, 2, 5, 4, 2, 3, 4, 2, 4, 6, 0]

u = [[0, 0], [5, 1], [0, 4], [1, 4], [1, 3], [3, 2], [3, 1], [2, 4], [4, 0],
     [5, 2], [2, 5], [0, 0]]

c = [6, 8]

S = [[0, 1], [0, 2], [0, 3], [1, 4], [1, 5], [2, 9], [2, 10], [3, 8], [4, 6],
```

(continues on next page)

(continued from previous page)

```
     [4, 7], [5, 9], [5, 10], [6, 8], [6, 9], [7, 8], [8, 11], [9, 11],
     [10, 11]]

(R, J, T) = (range(len(c)), range(len(p)), range(sum(p)))

model = Model()

x = [[model.add_var(name='x({},{})'.format(j, t), var_type=BINARY)
      for t in T] for j in J]

model.objective = xsum(x[len(J)-1][t] * t for t in T)

for j in J:
    model += xsum(x[j][t] for t in T) == 1

for (r, t) in product(R, T):
    model += xsum(u[j][r] * x[j][t2] for j in J
                  for t2 in range(max(0, t - p[j] + 1), t + 1)) <= c[r]

for (j, s) in S:
    model += xsum(t * x[s][t] - t * x[j][t] for t in T) >= p[j]

model.optimize()

print('Schedule: ')
for (j, t) in product(J, T):
    if x[j][t].x >= 0.99:
        print('({},{})'.format(j, t))
print('Makespan = {}'.format(model.objective_value))
```

The optimal solution is shown bellow, from the viewpoint of resource consumption:



4.6 Job Shop Scheduling Problem

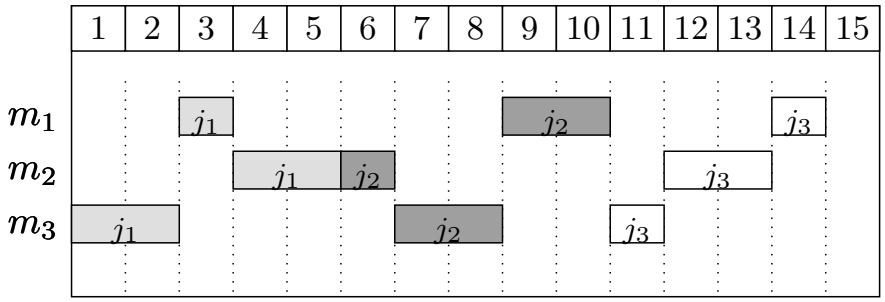
The Job Shop Scheduling Problem (JSSP) is an NP-hard problem defined by a set of jobs that must be executed by a set of machines in a specific order for each job. Each job has a defined execution time for each machine and a defined processing order of machines. Also, each job must use each machine only once. The machines can only execute a job at a time and once started, the machine cannot be interrupted until the completion of the assigned job. The objective is to minimize the makespan, i.e. the maximum completion time among all jobs.

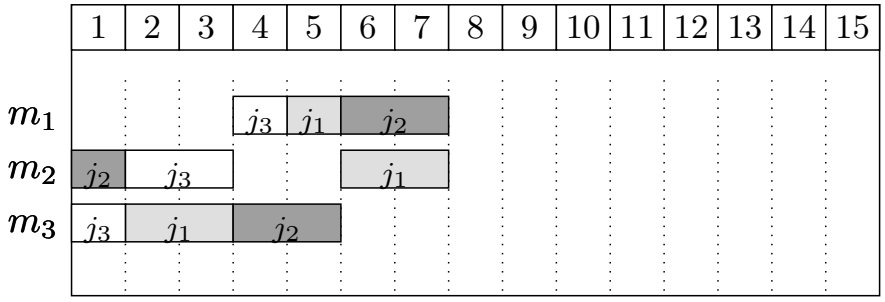
For instance, suppose we have 3 machines and 3 jobs. The processing order for each job is as follows (the processing time of each job in each machine is between parenthesis):

- Job j_1: m_3 (2) $\rightarrow$ m_1 (1) $\rightarrow$ m_2 (2)

- Job j_2: m_2 (1) $\rightarrow$ m_3 (2) $\rightarrow$ m_1 (2)
- Job j_3: m_3 (1) $\rightarrow$ m_2 (2) $\rightarrow$ m_1 (1)

Bellow there are two feasible schedules:





The first schedule shows a naive solution: jobs are processed in a sequence and machines stay idle quite often. The second solution is the optimal one, where jobs execute in parallel.

The JSSP has the following input data:

$\mathcal{J}$ set of jobs, $\mathcal{J} = \{1, ..., n\}$,

$\mathcal{M}$ set of machines, $\mathcal{M} = \{1, ..., m\}$,

o_r^j the machine that processes the r-th operation of job j, the sequence without repetition $O^j = (o_1^j, o_2^j, ..., o_m^j)$ is the processing order of j,

p_{ij} non-negative integer processing time of job j in machine i.

A JSSP solution must respect the following constraints:

- All jobs j must be executed following the sequence of machines given by O^j,
- Each machine can process only one job at a time,
- Once a machine starts a job, it must be completed without interruptions.

The objective is to minimize the makespan, the end of the last job to be executed. The JSSP is NP-hard for any fixed $n \geq 3$ and also for any fixed $m \geq 3$.

The decision variables are defined by:

x_{ij} starting time of job $j \in J$ on machine $i \in M$

$$y_{ijk} = \begin{cases} 1, & \text{if job } j \text{ precedes job } k \text{ on machine } i, \\ & i \in \mathcal{M},\ j, k \in \mathcal{J},\ j \neq k \\ 0, & \text{otherwise} \end{cases}$$

C variable for the makespan

Follows a MIP formulation [Man60] for the JSSP. The objective function is computed in the auxiliary variable C. The first set of constraints are the precedence constraints, that ensure that a job on a machine only starts after the processing of the previous machine concluded. The second and third set of disjunctive constraints ensure that only one job is processing at a given time in a given machine. The M constant must be large enough to ensure the correctness of these constraints. A valid (but weak) estimate for this value can be the summation of all processing times. The fourth set of constrains ensure that the makespan value is computed correctly and the last constraints indicate variable domains.

$$
\begin{aligned}
&\text{min:} \\
&\quad C \\
&\text{s.t.:} \\
&x_{o^j_r j} \geq x_{o^j_{r-1} j} + p_{o^j_{r-1} j} \;\; \forall r \in \{2, .., m\}, j \in \mathcal{J} \\
&x_{ij} \geq x_{ik} + p_{ik} - M \cdot y_{ijk} \;\; \forall j, k \in \mathcal{J}, j \neq k, i \in \mathcal{M} \\
&x_{ik} \geq x_{ij} + p_{ij} - M \cdot (1 - y_{ijk}) \;\; \forall j, k \in \mathcal{J}, j \neq k, i \in \mathcal{M} \\
&C \geq x_{o^j_m j} + p_{o^j_m j} \;\; \forall j \in \mathcal{J} \\
&x_{ij} \geq 0 \;\; \forall i \in \mathcal{J}, i \in \mathcal{M} \\
&y_{ijk} \in \{0, 1\} \;\; \forall j, k \in \mathcal{J}, i \in \mathcal{M} \\
&C \geq 0
\end{aligned}
$$

The following Python-MIP code creates the previous formulation, optimizes it and prints the optimal solution found:

```
from itertools import product
from sys import argv
from jssp_instance import JSSPInstance
from mip.model import Model, minimize
from mip.constants import BINARY

inst = JSSPInstance(argv[1])
n, m, machines, times, M = inst.n, inst.m, inst.machines, inst.times, inst.M

model = Model('JSSP')

c = model.add_var(name="C")
x = [[model.add_var(name='x({},{})'.format(j+1, i+1))
      for i in range(m)] for j in range(n)]
y = [[[model.add_var(var_type=BINARY, name='y({},{},{})'.format(j+1, k+1, i+1))
       for i in range(m)] for k in range(n)] for j in range(n)]

model.objective = minimize(c)

for (j, i) in product(range(n), range(1, m)):
    model += x[j][machines[j][i]] - x[j][machines[j][i-1]] >= \
        times[j][machines[j][i-1]]

for (j, k) in product(range(n), range(n)):
    if k != j:
        for i in range(m):
            model += x[j][i] - x[k][i] + M*y[j][k][i] >= times[k][i]
            model += -x[j][i] + x[k][i] - M*y[j][k][i] >= times[j][i] - M

for j in range(n):
    model += c - x[j][machines[j][m - 1]] >= times[j][machines[j][m - 1]]

model.optimize()

print("C: ", c.x)
for j in range(n):
    for i in range(m):
```

(continues on next page)

(continued from previous page)

```
        print('x({},{}) = {} '.format(j+1, i+1, x[j][i].x), end='')
    print()
```

Chapter 5

Developing Customized Branch-&-Cut algorithms

This chapter discusses some features of Python-MIP that allow the development of improved Branch-&-Cut algorithms by linking application specific routines to the generic algorithm included in the solver engine. We start providing an introduction to cutting planes and cut separation routines in the next section, following with a section describing how these routines can be embedded in the Branch-&-Cut solver engine using the generic cut callbacks of Python-MIP.

5.1 Cutting Planes

In many applications there are strong formulations that include an exponential number of constraints. These formulations cannot be direct handled by the MIP Solver: entering all these constraints at once is usually not practical. In the Cutting Planes [DFJ54] method the LP relaxation is solved and only constraints which are *violated* are inserted. The model is re-optimized and at each iteration a stronger formulation is obtained until no more violated inequalities are found. The problem of discovering which are the missing violated constraints is also an optimization problem (finding *the most* violated inequality) and it is called the *Separation Problem*.

As an example, consider the Traveling Salesman Problem. The compact formulation (Section 4.2) is a *weak* formulation: dual bounds produced at the root node of the search tree are distant from the optimal solution cost and improving these bounds requires a potentially intractable number of branchings. In this case, the culprit are the sub-tour elimination constraints involving variables x and y. A much stronger TSP formulation can be written as follows: consider a graph $G = (N, A)$ where N is the set of nodes and A is the set of directed edges with associated traveling costs $c_a \in A$. Selection of arcs is done with binary variables $x_a \ \forall a \in A$. Consider also that edges arriving and leaving a node n are indicated in A_n^+ and A_n^-, respectively. The complete formulation follows:

Minimize:

$$\sum_{a \in A} c_a . x_a$$

Subject to:

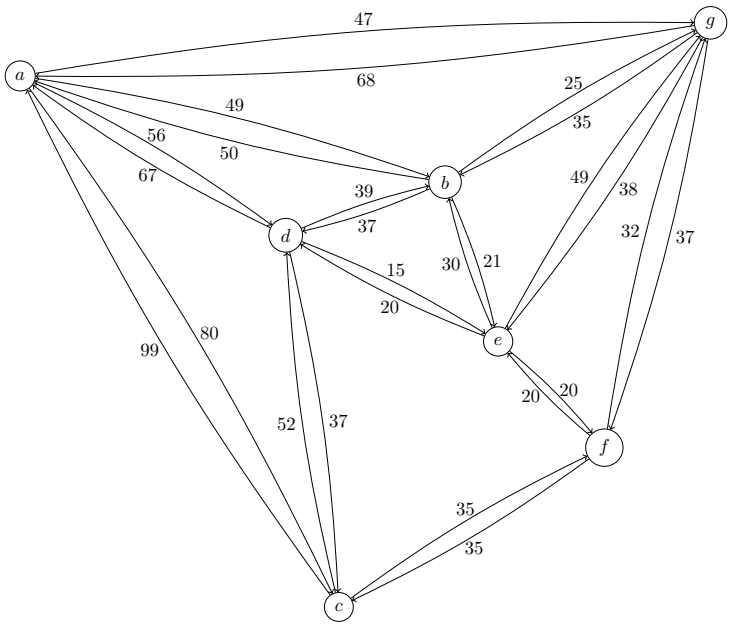
$$\sum_{a \in A_n^+} x_a = 1 \ \forall n \in N$$

$$\sum_{a \in A_n^-} x_a = 1 \ \forall n \in N$$

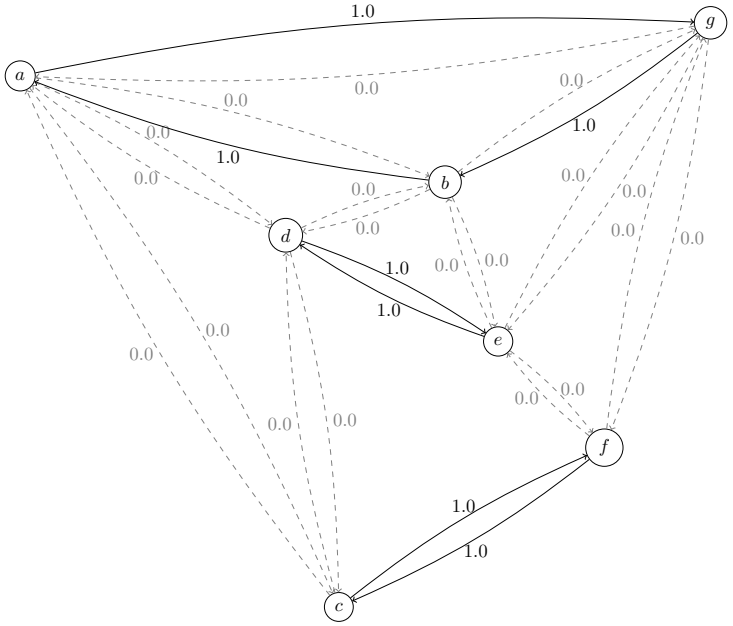
$$\sum_{(i,j) \in A : i \in S \wedge j \in S} x_{(i,j)} \leq |S| - 1 \ \forall S \subset I$$

$$x_a \in \{0, 1\} \ \forall a \in A$$

The third constraints are sub-tour elimination constraints. Since these constraints are stated for *every subset* of nodes, the number of these constraints is $O(2^{|N|})$. These are the constraints that will be separated by our cutting pane algorithm. As an example, consider the following graph:

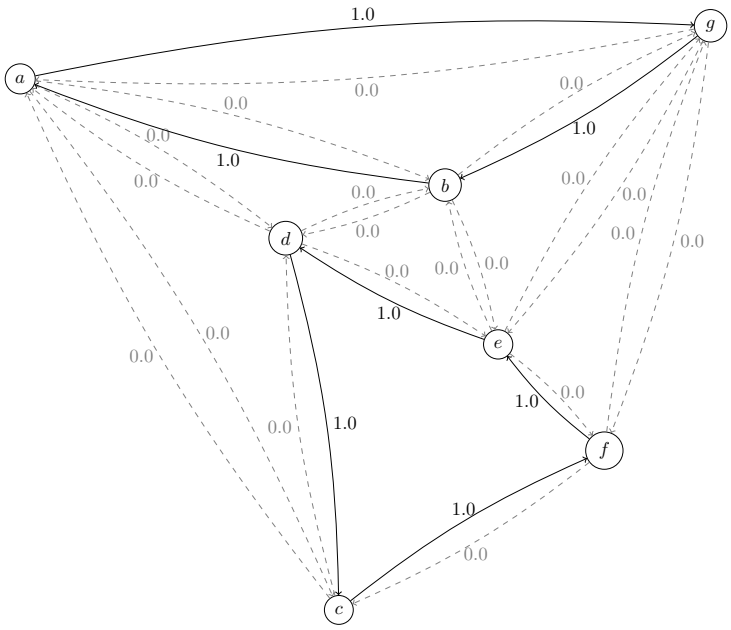


The optimal LP relaxation of the previous formulation without the sub-tour elimination constraints has cost 237:



As it can be seen, there are tree disconnected sub-tours. Two of these include only two nodes. Forbidding sub-tours of size 2 is quite easy: in this case we only need to include the additional constraints: $x_{(d,e)} + x_{(e,d)} \leq 1$ and $x_{(c,f)} + x_{(f,c)} \leq 1$.

Optimizing with these two additional constraints the objective value increases to 244 and the following new solution is generated:



Now there are sub-tours of size 3 and 4. Let's consider the sub-tour defined by nodes $S = \{a, b, g\}$. The valid inequality for S is: $x_{(a,g)} + x_{(g,a)} + x_{(a,b)} + x_{(b,a)} + x_{(b,g)} + x_{(g,b)} \leq 2$. Adding this cut to our model increases the objective value to 261, s significant improvement. In our example, the visual identification of the isolated subset is easy, but how to automatically identify these subsets efficiently in the general case ? A subset is a *cut* in a Graph. To identify the most isolated subset we just have to solve the Minimum cut problem in graphs. In python you can use the networkx min-cut module. The following code implements a cutting plane algorithm for the asymmetric traveling salesman problem:

```
from mip.model import Model, xsum
from mip.constants import BINARY
from itertools import product
from networkx import minimum_cut, DiGraph

N = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
A = {('a', 'd'): 56, ('d', 'a'): 67, ('a', 'b'): 49, ('b', 'a'): 50,
     ('f', 'c'): 35, ('g', 'b'): 35, ('g', 'b'): 35, ('b', 'g'): 25,
     ('a', 'c'): 80, ('c', 'a'): 99, ('e', 'f'): 20, ('f', 'e'): 20,
     ('g', 'e'): 38, ('e', 'g'): 49, ('g', 'f'): 37, ('f', 'g'): 32,
     ('b', 'e'): 21, ('e', 'b'): 30, ('a', 'g'): 47, ('g', 'a'): 68,
     ('d', 'c'): 37, ('c', 'd'): 52, ('d', 'e'): 15, ('e', 'd'): 20,
     ('d', 'b'): 39, ('b', 'd'): 37, ('c', 'f'): 35}
Aout = {n: [a for a in A if a[0] == n] for n in N}
Ain = {n: [a for a in A if a[1] == n] for n in N}

m = Model()
x = {a: m.add_var(name='x({},{})'.format(a[0], a[1]), var_type=BINARY)
     for a in A}

m.objective = xsum(c*x[a] for a, c in A.items())

for n in N:
    m += xsum(x[a] for a in Aout[n]) == 1, 'out({})'.format(n)
    m += xsum(x[a] for a in Ain[n]) == 1, 'in({})'.format(n)

newConstraints = True
m.relax()

while newConstraints:
    m.optimize()
    print('objective value : {}'.format(m.objective_value))
```

(continues on next page)

(continued from previous page)

```
33
34      G = DiGraph()
35      for a in A:
36          G.add_edge(a[0], a[1], capacity=x[a].x)
37
38      newConstraints = False
39      for (n1, n2) in [(i, j) for (i, j) in product(N, N) if i != j]:
40          cut_value, (S, NS) = minimum_cut(G, n1, n2)
41          if (cut_value <= 0.99):
42              m += xsum(x[a] for a in A if (a[0] in S and a[1] in S)) <= len(S)-1
43              newConstraints = True
```

Lines 6-13 are the input data. Nodes are labeled with letters in a list `N` and a dictionary `A` is used to store the weighted directed graph. Lines 14 and 15 store output and input arcs per node. The mapping of binary variables x_a to arcs is made also using a dictionary in line 18. Line 21 sets the objective function and the following tree lines include constraints enforcing one entering and one leaving arc to be selected for each node. On line 28 we relax the integrality constraints of variables so that the optimization performed in line 31 will only solve the LP relaxation and the separation routine can be executed. Our separation routine is executed for each pair or nodes at line 40 and whenever a disconnected subset is found the violated inequality is generated and included at line 42. The process repeats while new violated inequalities are generated.

5.2 Cut Callback

The cutting plane method has some limitations: even though the first rounds of cuts improve significantly the lower bound, the overall number of iterations needed to obtain the optimal integer solution may be too large. Better results can be obtained with the Branch-&-Cut algorithm, where cut generation is *combined* with branching. If you have an algorithm like the one included in the previous Section to separate inequalities for your application you can combine it with the complete BC algorithm implemented in the solver engine using *callbacks*. Cut generation callbacks (CGC) are called at each node of the search tree where a fractional solution is found. Cuts are generated in the callback and returned to the MIP solver engine which adds these cuts to the *Cut Pool*. These cuts are merged with the cuts generated with the solver builtin cut generators and a *subset* of these cuts in included to the LP relaxation model. Please note that in the Branch-&-Cut algorithm context cuts are *optional* components and only those that are classified as *good* cuts by the solver engine will be accepted, i.e., cuts that are too dense and/or have a small violation could be discarded, since the cost of solving a much larger linear program may not be worth the resulting bound improvement.

When using cut callbacks be sure that cuts are used only to *improve* the LP relaxation but not to *define* feasible solutions, which need to be defined by the initial formulation. In other words, the initial model without cuts may be *weak* but needs to be *complete*. In the case of TSP, we can include the weak sub-tour elimination constraints presented in Section 4.2 in the initial model and then add the stronger sub-tour elimination constraints presented in the previous section as cuts.

In Python-MIP, CGC are implemented extending the *`ConstrsGenerator`* class. The following example implements the previous cut separation algorithm as a *`ConstrsGenerator`* class and includes it as a cut generator for the branch-and-cut solver engine. The method that needs to be implemented in this class is the `generate_cuts()` procedure. This method receives as parameter the object `model` of type *`Model`*. This object must be used to query the fractional values of the model *`vars`*, using the *`x`* property. Other model properties can be queried, such as the problem constraints (*`constrs`*). Please note that, depending on which solver engine you use, some variables/constraints from the original model may have been removed by pre-processing. Thus, direct references to the original problem variables may be invalid. Since for variables that remain in this model Python-MIP ensures that the names from the original variables are preserved, it is a good practice to query again the variables list. In our example, the relationship of the variables with the arcs of the input graph can be inferred by examining variable names which are in the format "`x(i,j)`" (lines 15-17). Whenever a violated inequality is discovered, it

can be added to the solver's engine cut pool using the *add_cut() Model* method (lines 33 and 35). In our example, we temporarily store the generated cuts in our *CutPool* object (line 30).

```
1  from sys import argv
2  from typing import List, Tuple
3  import networkx as nx
4  from tspdata import TSPData
5  from mip.model import Model, xsum, BINARY
6  from mip.callbacks import ConstrsGenerator, CutPool
7
8
9  class SubTourCutGenerator(ConstrsGenerator):
10     def __init__(self, Fl: List[Tuple[int, int]]):
11         self.F = Fl
12
13     def generate_constrs(self, model: Model):
14         G = nx.DiGraph()
15         r = [(v, v.x) for v in model.vars if v.name.startswith('x(')]
16         U = [int(v.name.split('(')[1].split(',')[0]) for v, f in r]
17         V = [int(v.name.split(')')[0].split(',')[1]) for v, f in r]
18         cp = CutPool()
19         for i in range(len(U)):
20             G.add_edge(U[i], V[i], capacity=r[i][1])
21         for (u, v) in F:
22             if u not in U or v not in V:
23                 continue
24             val, (S, NS) = nx.minimum_cut(G, u, v)
25             if val <= 0.99:
26                 arcsInS = [(v, f) for i, (v, f) in enumerate(r)
27                            if U[i] in S and V[i] in S]
28                 if sum(f for v, f in arcsInS) >= (len(S)-1)+1e-4:
29                     cut = xsum(1.0*v for v, fm in arcsInS) <= len(S)-1
30                     cp.add(cut)
31                     if len(cp.cuts) > 256:
32                         for cut in cp.cuts:
33                             model += cut
34                         return
35         for cut in cp.cuts:
36             model += cut
37         return
38
39
40 inst = TSPData(argv[1])
41 n, d = inst.n, inst.d
42
43 model = Model()
44
45 x = [[model.add_var(name='x({},{})'.format(i, j),
46                     var_type=BINARY) for j in range(n)] for i in range(n)]
47 y = [model.add_var(name='y({})'.format(i),
48                    lb=0.0, ub=n) for i in range(n)]
49
50 model.objective = xsum(d[i][j] * x[i][j] for j in range(n) for i in range(n))
51
52 for i in range(n):
53     model += xsum(x[j][i] for j in range(n) if j != i) == 1
54     model += xsum(x[i][j] for j in range(n) if j != i) == 1
55 for (i, j) in [(i, j) for (i, j) in
56                product(range(1, n), range(1, n)) if i != j]:
57     model += y[i] - (n + 1) * x[i][j] >= y[j] - n
58
59 F = []
```

(continues on next page)

(continued from previous page)

```
    for i in range(n):
        (md, dp) = (0, -1)
        for j in [k for k in range(n) if k != i]:
            if d[i][j] > md:
                (md, dp) = (d[i][j], j)
        F.append((i, dp))

    m.cuts_generator = SubTourCutGenerator(F)
    model.optimize()

    arcs = [(i, j) for i in range(n) for j in range(n) if x[i][j].x >= 0.99]
    print('optimal route : {}'.format(arcs))
```

5.3 Lazy Constraints

Python-MIP also supports the use of cut generators to produce *lazy constraints*. Lazy constraints are dynamically generated, just as cutting planes, with the difference that lazy constraints are also applied to *integer solutions*. They should be used when the initial formulation is *incomplete*. In the case of our previous TSP example, this approach allow us to use in the initial formulation only the degree constraints and add all required sub-tour elimination constraints on demand. Auxiliary variables y would also not be necessary. The lazy constraints TSP example is exaclty as the cut generator callback example with the difference that, besides starting with a smaller formulation, we have to inform that the cut generator will be used to generate lazy constraints:

```
    ...
    m.constrs_generator = SubTourCutGenerator(F)
    m.constrs_generator.lazy_constraints = True
    model.optimize()
    ...
```

5.4 Providing initial feasible solutions

The Branch-&-Cut algorithm usually executes faster with the availability of an integer feasible solution: an upper bound for the solution cost improves its ability of pruning branches in the search tree and this solution is also used in local search MIP heuristics. MIP solvers employ several heuristics for the automatically production of these solutions but they do not always succeed.

If you have some problem specific heuristic which can produce an initial feasible solution for your application then you can inform this solution to the MIP solver using the *start* model property. Let's consider our TSP application (Section 4.2). If the graph is complete, i.e. distances are available for each pair of cities, then *any* permutation $\Pi = (\pi_1, \ldots, \pi_n)$ of the cities N can be used as an initial feasible solution. This solution has exactly $|N|$ x variables equal to one indicating the selected arcs: $((\pi_1, \pi_2), (\pi_2, \pi_3), \ldots, (\pi_{n-1}, \pi_n), (\pi_n, \pi_1))$. Even though this solution is obvious for the modeler, which knows that binary variables of this model refer to arcs in a TSP graph, this solution is not obvious for the MIP solver, which only sees variables and a constraint matrix. The following example enters an initial random permutation of cities as initial feasible solution for our TSP example, considering an instance with `n` cities, and a model `model` with references to variables stored in a matrix `x[0,...,n-1][0,..,n-1]`:

```
from random import shuffle
S=[i for i in range(n)]
shuffle(S)
model.start = [(x[S[k-1]][S[k]], 1.0) for k in range(n)]
```

The previous example can be integrated in our TSP example (Section 4.2) by inserting these lines before the `model.optimize()` call. Initial feasible solutions are informed in a list (line 4) of `(var, value)` pairs.

Please note that only the original non-zero problem variables need to be informed, i.e., the solver will automatically compute the values of the auxiliary y variables which are used only to eliminate sub-tours.

Chapter 6

Benchmarks

This section presents computational experiments on the creation of Integer Programming models using different mathematical modelling packages. Gurobi is the default Gurobi ® Python interface, which currently only supports the Python interpreter CPython. Pulp supports both CPython and also the just-in-time compiler Pypy. MIP also suports both. JuMP [DHL17] is the Mathematical Programming package of the Julia programming language. Both Jump and Pulp use intermediate data structures to store the mathematical programming model before flushing it to the solver, so that the selected solver does not impacts on the model creation times. MIP does not stores the model itself, directly calling problem creation/modification routines on the solver engine.

Since MIP communicates every problem modification directly to the solver engine, the engine must handle efficiently many small modification request to avoid potentially expensive resize/move operations in the constraint matrix. Gurobi automatically buffers problem modification requests and has an update method to flush these request. CBC did not had an equivalent mechanism, so we implemented an automatic buffering/flushing mechanism in the CBC C Interface. Our computational results already consider this updated CBC version.

Computational experiments executed on a ThinkPad ® X1 notebook with an Intel ® Core™ i7-7600U processor and 8 Gb of RAM using the Linux operating system. The following software releases were used: CPython 3.7.3, Pypy 7.1.1, Julia 1.1.1, JuMP 0.19 and Gurobi 8.1 and CBC svn 2563.

6.1 n-Queens

These are binary programming models. The largest model has 1 million variables and roughly 6000 constraints and 4 million of non-zero entries in the constraint matrix.

n	Gurobi	Pulp		Python-MIP				JuMP
	CPython			Gurobi		CBC		
		CPython	Pypy	CPython	Pypy	CPython	Pypy	
100	0.24	0.27	**0.18**	0.65	0.97	0.30	0.45	2.38
200	1.43	1.73	0.43	1.60	**0.18**	1.47	0.19	0.25
300	4.97	5.80	0.92	5.48	**0.37**	5.12	0.38	0.47
400	12.37	14.29	1.55	13.58	**0.72**	13.24	0.74	1.11
500	24.70	32.62	2.62	27.25	1.25	26.30	**1.23**	2.09
600	43.88	49.92	4.10	47.75	2.02	46.23	**1.99**	2.86
700	69.77	79.57	5.84	75.97	3.04	74.47	**2.94**	3.64
800	105.04	119.07	8.19	114.86	4.33	112.10	**4.26**	5.58
900	150.89	169.92	10.84	163.36	5.95	160.67	**5.83**	8.08
1000	206.63	232.32	14.26	220.56	8.02	222.09	**7.76**	10.02

Chapter 7

Classes

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

Python-MIP constants

7.1 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 *VarList*

`constrs`

list of constraints (*Constr*)

Type *ConstrList*

`add_constr(`*lin_expr*, *name=''*`)`

Creates a new constraint (row).

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

Parameters

- **lin_expr** (LinExpr) – linear expression
- **name** (*str*) – optional constraint name, used when saving model to lp or mps files

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 *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` (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` (LinExpr) – the linear constraint

`add_sos`(*sos*, *sos_type*)

Adds an Special Ordered Set (SOS) to the model

In models with binary variables it is often the case that from a list of variables only one can receive value 1 in a feasible solution. When large constraints of this type exist (packing and partitioning), branching in one variable at time usually doesn't work well: while fixing one of these variables to one leaves only one possible feasible value for the other variables in this set (zero), fixing one variable to zero keeps all other variables free. This *unbalanced* branching is highly ineffective. A Special ordered set (SOS) is a set $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ with weights $[w_1, w_2, \ldots, w_k] \in \mathbb{R}^+$. With this structure available branching on a fractional solution x^* for these variables can be performed computing:

$$\min\{u_{k'} : u_{k'} = |\sum_{j=1 \ldots k'-1} w_j . x_j^* - \sum_{j=k' \ldots,k} w_j . x_j^*|\}$$

Then, branching $\mathcal{S}_1$ would be $\sum_{j=1,\ldots,k'-1} x_j = 0$ and $\mathcal{S}_2 = \sum_{j=k',\ldots,k} x_j = 0$.

Parameters

- `sos` (*List[Tuple[Var, float]]*) – 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` (*float*) – variable lower bound, default 0.0
- `ub` (*float*) – variable upper bound, default infinity
- `obj` (*float*) – coefficient of this variable in the objective function, default 0
- `var_type` (*str*) – CONTINUOUS ("C"), BINARY ("B") or INTEGER ("I")
- `column` (Column) – constraints where this variable will appear, necessary only when constraints are already created in the model and a new variable will be created.

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 creates a vector of binary variables `x[0], ..., x[n-1]` to model `m`:

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

Return type *Var*

`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`

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

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

Return type *Constr*

Returns constraint or None if not found

`copy(`*solver_name=None*`)`
Creates a copy of the current model

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

Return type *Model*

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 float

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 *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 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

property infeas_tol
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.

Type Maximum allowed violation for constraints. Default value

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.

Return type float

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 *ConstrsGenerator*

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 +=

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

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

Return type *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 float

property objective_const
Returns the constant part of the objective function

Return type float

property objective_value
Objective function value of the solution found

Return type float

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

Return type List[float]

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=inf*, *max_solutions=inf*)
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** (*float*) – Maximum runtime in seconds (default: inf)
- **max_nodes** (*float*) – Maximum number of nodes (default: inf)
- **max_solutions** (*float*) – Maximum number of solutions (default: inf)

Return type *OptimizationStatus*

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.

`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 are not getting feasible solutions.

Return type `int`

`read(`*path*`)`
Reads 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 loaded:

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

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

`.sol` initial feasible solution

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` – can be a Var, a Constr or a list of these objects

`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 *ProgressLog*

`property sense`
The optimization sense

Return type `str`

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

`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 `List[Tuple[Var, float]]`

`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 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 *OptimizationStatus*

`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`

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

Return type *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

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

7.2 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
```

`add_const(`*_LinExpr__const*`)`
adds a constant value to the linear expression, in the case of a constraint this correspond to the right-hand-side

`add_expr(`*_LinExpr__expr*, *coeff=1*`)`
extends a linear expression with the contents of another

`add_term(`*_LinExpr__expr*, *coeff=1*`)`
extends a linear expression with another multiplied by a constant value coefficient

`add_var(`*var*, *coeff=1*`)`
adds a variable with a coefficient to the constraint

`property const`
constant part of the linear expression

Return type `float`

`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 float.

Return type `dict`

`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`

`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.

7.3 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 *Column*

property lb
Variable lower bound.

Return type float

property name
Variable name.

Return type str

property obj
Coefficient of variable in the objective function.

Return type float

property rc
Reduced cost, only available after a linear programming model (only continuous variables) is optimized

Return type float

property ub
Variable upper bound.

Return type float

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

Type Variable type

Return type str

property x
Value of this variable in the solution.

Return type float

xi(*i*)
Value for this variable in the *i*-th solution from the solution pool.

Return type float

7.4 Constr

class Constr(*model*, *idx*)
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
contents of the constraint

Return type *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 float

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

Return type float

7.5 Column

class Column(*constrs=None*, *coeffs=None*)
A column contains all the non-zero entries of a variable in the constraint matrix. To create a variable see add_var().

7.6 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)
```

7.7 ConstrList

class ConstrList(*model*)
List of problem constraints

7.8 ConstrsGenerator

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

generate_constrs(*model*)
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 in *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` (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 or lazy constraints (`add_constrs()`), but you cannot perform other model modifications, such as add columns.

7.9 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
- `solution` (*cost*) – non-zero variables
- `the solution` (*in*) –

Return type `List[Tuple[Var, float]]`

7.10 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` (LinExpr) – a constraint

Return type `bool`

7.11 OptimizationStatus

`class OptimizationStatus`

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.

`INFEASIBLE = 1`
The model is proven infeasible

`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.

7.12 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`
Tuple in the format $(time, (lb, ub))$, where $time$ is the processing time and lb and ub are the lower and upper bounds, respectively

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

`instance`
instance name

Type str

`settings`
identification of the formulation/parameter

Type str

`settings used in the optimization (whatever is relevant to`

`identify a given computational experiment)`

`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

7.13 Useful functions

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

Parameters `expr` (`LinExpr`) – linear expression

Return type *LinExpr*

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

Parameters `expr` (`LinExpr`) – linear expression

Return type *LinExpr*

`model.xsum()`
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 *LinExpr*

Bibliography

[And73] L. G. Anderson. A simulation study of some dynamic channel assignment algorithms in a high capacity mobile telecommunications system. *IEEE Transactions on Communications*, 21:1294–1301, 1973.

[ABCC06] D.L. Applegate, R.E. Bixby, V. Chvatal, and W.J. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[DFJ54] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[DHL17] I. Dunning, J. Huchette, and M. Lubin. Jump: a modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[FGL05] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, 2005.

[Man60] A.S. Manne. On the job-shop scheduling problem. *Operations Research*, 8(2):219–223, 1960.

[PWW69] A. Pritsker, L. Watters, and P. Wolfe. Multi-project scheduling with limited resources: a zero-one programming approach. *Management Science*, 16:93–108, 1969.

Python Module Index

m

Index

P

R

S

T

U

V

W

X