Introduction to the use of solvers with Python

Integer linear programming: formulations, techniques and applications.

Santiago Valdés Ravelo
santiago.ravelo@inf.ufrgs.br

October, 2020
“First, solve the problem. Then, write the code.”

Attributed to John Johnson.
1. Introduction

2. PuLP
   - Coding formulations
   - Solving formulations

3. DipPy
Introduction
Commercial solvers
Non-commercial solvers

GLPK (GNU Linear Programming Kit)

Ipsolve
Mixed Integer Linear Programming (MILP) solver
Brought to you by: keikland, peno64
Why python?

- Very intuitive and easy-to-program language.
- Growing community and popularity:
  - 3rd in TIOBE index.
  - 2nd in GitHub.
- Common interfaces for different solvers (e.g., PuLP/DipPy):
  - Simplifies the formulations coding.
  - Facilitates to embed solvers calls in algorithm solutions (e.g., matheuristics) or in software applications.
PuLP
PuLP is a linear programming modeler written in Python and able to integrate with different solvers.

The main classes for coding formulations are \texttt{LpProblem}, \texttt{LpVariable}, \texttt{LpConstraint} and \texttt{LpConstraintVar}.

The solvers interfaces are given by the classes \texttt{LpSolver} and \texttt{LpSolver\_CMD}.

Access documentation here  

PuLP
Import **PuLP** classes and functions: `from pulp import *`

Define the variables:
- `myVariable = LpVariable(name, lowBound = None, upBound = None, cat = 'Continuous', e = None)`
- `myVariables = LpVariable.dicts(name, indexs, lowBound = None, upBound = None, cat = 0)`

Define the problem:
- `myProblem = LpProblem("name", LpMinimize)`
- `myProblem = LpProblem("name", LpMaximize)`

Define the objective function: `myProblem += expression, name`

Define the constraints:
- `myProblem += expression <= value`
- `myProblem += expression == value`
- `myProblem += expression >= value`
Consider the following problem:

\[
\begin{align*}
\text{max} & \quad 3 \times x + 2 \times y \\
\text{s.t. :} & \quad x - y + z = 1 \\
& \quad x + 2 \times y \leq 14 \\
& \quad 4 \times x + y \leq 20 \\
& \quad x, y \in \mathbb{Z}_+, z \in \mathbb{R}_+ 
\end{align*}
\]
Coding formulations. Simple example

\[
\begin{align*}
\text{max} & \quad 3 \times x + 2 \times y \\
\text{s.t.:} & \quad x - y + z = 1 \\
& \quad x + 2 \times y \leq 14 \\
& \quad 4 \times x + y \leq 20 \\
& \quad x, y \in \mathbb{Z}_+, z \in \mathbb{R}_+ 
\end{align*}
\]

```python
from pulp import *

x = LpVariable("x", lowBound = 0, cat = 'Integer')
y = LpVariable("y", lowBound = 0, cat = 'Integer')
z = LpVariable("z", lowBound = 0)

problem = LpProblem("myProblem", LpMaximize)
problem += 3 * x + 2 * y, "myObjective"
problem += x - y + z == 1
problem += x + 2 * y <= 14
problem += 4 * x + y <= 20
```
Coding formulations. Knapsack

Instance: $O, \nu: O \rightarrow \mathbb{R}_+, \omega: O \rightarrow \mathbb{R}_+, W \in \mathbb{R}_+$.

\[ \begin{align*}
\text{max} & \quad \sum_{o \in O} \nu(o) \times x_o \\
\text{s.t.} & \quad \sum_{o \in O} \omega(o) \times x_o \leq W \\
& \quad x_o \in \{0, 1\}, \forall o \in O
\end{align*} \]
from pulp import *

def knapsack(values, weights, W):
    x = LpVariable.dicts("x", [o for o in range(len(values))], lowBound = 0, upBound = 1, cat='Integer')

    problem = LpProblem("BinaryKnapsack", LpMaximize)

    problem += lpSum(values[o] * x[o] for o in range(len(values))), "profit"

    problem += lpSum(weights[o] * x[o] for o in range(len(values))) <= W
Instance: $O, \nu: O \rightarrow \mathbb{R}_+, \omega: O \rightarrow \mathbb{R}_+, W \in \mathbb{R}^m$.

\[
\max \sum_{o \in O} \sum_{i=1}^{m} \nu(o) \times x_{oi}
\]

s.t.:

\[
\sum_{o \in O} \omega(o) \times x_{oi} \leq W_i \quad \forall 1 \leq i \leq m
\]

\[
\sum_{i=1}^{m} x_{oi} \leq 1 \quad \forall o \in O
\]

\[
x_{oi} \in \{0, 1\} \quad \forall o \in O, 1 \leq i \leq m
\]
from pulp import *

def multipleKnapsack(values, weights, W):
    x = LpVariable.dicts("x", [(o, i) for o in range(len(values)) for i in range(len(W))],
                lowBound = 0, upBound = 1, cat='Integer')

    problem = LpProblem("MultipleKnapsack", LpMaximize)

    problem += lpSum(values[o] * x[o, i] for o in range(len(values)) for i in range(len(W))),
                "profit"

    for i in range(len(W)):
        problem += lpSum(weights[o] * x[o, i] for o in range(len(values))) <= W[i]

    for o in range(len(W)):
        problem += lpSum(x[o, i] for i in range(len(W))) <= 1
The function `list_solvers()` returns the available solvers:

- e.g., `['GLPK_CMD', 'PYGLPK', 'CPLEX_CMD', 'CPLEX_PY', 'GUROBI', 'GUROBI_CMD', 'XPRESS', 'COIN_CMD', 'SCIP_CMD']`

Get the solver:

- e.g., `solver = GUROBI(parameters)`.
- Some of the parameters may include: `timeLimit` in seconds, `Cuts`, `Heuristics` and `Presolve` to indicate if there will be used, respectively, standard cut generation, embedded heuristics and preprocessing.

Solve the problem:

- `myProblem.solve(solver)`
- `solver.buildSolverModel(myProblem), solver.callSolver(myProblem) and solver.findSolutionValues(myProblem)`

Get solution value: `value(myProblem.objective)`.

The variables are elements in `myProblem.variables()` each one with attributes `name` and `varValue`. 
from pulp import *

x = LpVariable("x", lowBound = 0, cat = 'Integer')
y = LpVariable("y", lowBound = 0, cat = 'Integer')
z = LpVariable("z", lowBound = 0)

problem = LpProblem("myProblem", LpMaximize)

problem += 3 * x + 2 * y, "myObjective"

problem += x - y + z == 1
problem += x + 2 * y <= 14
problem += 4 * x + y <= 20

problem.solve(GUROBI_CMD())

print('Optimal value: ' + str(value(problem.objective)))
print('Optimal solution: ')
for variable in problem.variables():
    print('    ' + variable.name + " = " + str(variable.varValue))
from pulp import *

def knapsack(values, weights, W):
    x = LpVariable.dicts("x", [o for o in range(len(values))], lowBound = 0, upBound = 1, cat='Integer')

    problem = LpProblem("BinaryKnapsack", LpMaximize)

    problem += lpSum(values[o] * x[o] for o in range(len(values))), "profit"

    problem += lpSum(weights[o] * x[o] for o in range(len(values))) <= W

    solver = GUROBI(timeLimit = 3600)

    solver.buildSolverModel(problem)
    solver.callSolver(problem)
    solver.findSolutionValues(problem)

    print('Optimal value: ' + str(value(problem.objective)))
    print('Optimal solution: ')
    for variable in problem.variables():
        print('  ' + variable.name + ' = ' + str(variable.varValue))
DipPy
Advanced branching. Includes `branch_method`.

Customized cuts. Includes `generate_cuts` and `is_solution_feasible`.

Customized columns. Includes `relaxed_solver` and `init_vars`.

Heuristics. Includes `heuristics`.

Introduction to the use of solvers with Python

Integer linear programming: formulations, techniques and applications.

Santiago Valdés Ravelo
santiago.ravelo@inf.ufrgs.br

October, 2020