PyomoExample

February 28, 2020

1 Pyomo package in Python for optimization

Here we illustrate the use of Pyomo for solving mathematical optimization models. There are other Python packages (such as PuLP) that can be used for (integer) linear models, but here we stick to Pyomo because it can also be used to model *nonlinear* optimization models, which will be part of the next in class assignment.

Frans de Ruiter

This example is taken from here.

More information about Pyomo can be found on the website: http://www.pyomo.org/documentation.

1.1 Installing pyomo

You can install pyomo by running

`pip install pyomo`

or easier by using conda:

`conda install -c conda-forge pyomo`.

You also need the package glpk (and ipopt for the next in class assignment):

`conda install -c conda-forge ipopt glpk`

See the Pyomo installation instructions.

We recommend using the coincbc solver instead of the glpk solver for the facility location problem, since it is much faster. On UNIX you can install the coincbc solver by:

`conda install -c conda-forge coincbc`

1.1.1 ! Installing CBC solver on Windows !

(instruction added 30/5/2018, see also solve command at the bottom of this file)

1. **Download and install** the coin cbc optimization solver (note this is an .exe file) via

   `https://www.coin-or.org/download/binary/OptimizationSuite/COIN-OR-1.8.0-win32-msvc12.exe`

   Make sure you add cbc to your path (is done automatically via an option at the end of the installation process).

2. **Restart jupyter notebook** (if it was already open open) and run the rest of this script. If you run this script without restarting jupyter notebook, the script will not run because the path to the solver cannot be found.

1.2 Knapsack model formulation

The Knapsack Problem considers the problem of selecting a set of items whose weight is not greater than a specified limit while maximizing the total value of the selected items. This problem is inspired by the challenge of filling a knapsack (or rucksack) with the most valuable items that can be carried.

A common version of this problem is the 0-1 knapsack problem, where each item is distinct and can be selected once. Suppose there are n items with positive values v1,…,vn and weights w1,…,wn. Let x1,…,xn be decision variables that can take values 0 or 1. Let W be the weight capacity of the knapsack.

The following optimization formulation represents this problem as an integer program:

$$\begin{array}{ll} \max & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0,1\} \end{array}$$

The following section illustrate how to model and solve this problem using the Pyomo package.

1.2.1 Importing packages

[1]:
```
import pyomo
from pyomo.environ import *
```

1.2.2 Input data

[2]:
```
v = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
limit = 14
items = list(sorted(v.keys()))
```

1.2.3 Model implementation

[3]:
```
# Create model
m = ConcreteModel()

# Variables
m.x = Var(items, within=Binary) # use within=NonNegativeReals for a variable␣
↪that should be nonnegative and continuous

# Objective
m.value = Objective(expr=sum(v[i]*m.x[i] for i in items), sense=maximize)

# Constraint
m.weight = Constraint(expr=sum(w[i]*m.x[i] for i in items) <= limit)
```

To add constraints in a loop, you can use something like the following:

[4]:
```
# m.constraintloop = ConstraintList()
# for i in list_of_items:
#     m.constraintloop.add(expr=sum(w[i]*m.x[i] for i in items) <= limit)
```

1.2.4 Solve and print solution

[7]:
```
# Optimize
solver = SolverFactory('cbc') # Use cbc solver
#solver = SolverFactory('glpk') # glpk solver is not recommended

# Set a time limit for 3600 seconds (1 hour). Cbc will find the optimal␣
↪solutions within a minute, but glpk does not.
solver.options['tmlim'] = 3600

status = solver.solve(m,tee=False,) # setting tee=True enables you to see the␣
↪progress of the solver
status = solver.solve(m)

# Print the status of the solved LP
print("Status = %s" % status.solver.termination_condition)

# Print the value of the variables at the optimum
for i in items:
    print("%s = %f" % (m.x[i], value(m.x[i])))

# Print the value of the objective
print("Objective = %f" % value(m.value))
```

```
Status = optimal
```

```
x[hammer] = 1.000000
x[screwdriver] = 1.000000
x[towel] = 1.000000
x[wrench] = 0.000000
Objective = 25.000000
```

[ ]:

```

```