Optimization for Communications and Networks
Handout B12

Dr. Pratkasem Vesarach

August Semester 2016

1 Introduction to PuLP and GLPK

- PuLP is an open-source Python module that is used to model Linear Programming (LP), Integer Linear Programming (ILP) and Mixed Integer Linear Programming (MILP) optimization problems as mathematical models. It also can be used to call external LP solvers to solve the problems.
- In this class, PuLP is used as a mathematical modeller for optimization problems (e.g., LP, ILP or MILP). It works in conjunction with GNU Linear Programming Kit (GLPK) package which is an open-source solver for linear optimization problems.
- Before PuLP can be imported in Python, it must be installed first. Use the following link for the installation instructions.

  www.coin-or.org/PuLP/main/installing_pulp_at_home.html

- As for GLPK, use the following link for the installation instructions.

  https://www.gnu.org/software/glpk/

- To import PuLP module, use the code "`from pulp import *`" in the Python interactive shell or at the beginning of Python scripts/functions.
- To check whether PuLP and GLPK installations work properly, use the function "`pulp.pulpTestAll()`" in the Python interactive shell. The outputs show the information about the status of PuLP and GLPK.

  Example:

```
>>> from pulp import *
>>> pulp.pulpTestAll()
Solver pulp.solvers.CPLEX_DLL unavailable.
Solver pulp.solvers.CPLEX_CMD unavailable.
Solver pulp.solvers.COIN_CMD unavailable.
Solver pulp.solvers.COINMP_DLL unavailable.
Testing continuous LP solution
     Testing maximize continuous LP solution
     Testing unbounded continuous LP solution
     Testing unbounded continuous LP solution
     Testing MIP solution
     Testing MIP relaxation
     Testing feasibility problem (no objective)
     Testing an infeasible problem
     Testing an integer infeasible problem
     Testing column based modelling
     Testing fractional constraints
     Testing elastic constraints (no change)
     Testing elastic constraints (freebound)
     Testing elastic constraints (penalty unchanged)
```

```
     Testing elastic constraints (penalty unbounded)
* Solver pulp.solvers.GLPK_CMD passed. #GLPK is installed and available.#
Solver pulp.solvers.XPRESS unavailable.
Solver pulp.solvers.GUROBI unavailable.
```

2 PuLP Syntax and Functions

- In order to use PuLP to model an optimization problem, the first thing is to create a variable that represents the concerned optimization problem. The function for this purpose is "`LpProblem()`". The format of `LpProblem()` is

$$prob\_var = \texttt{LpProblem(}name\texttt{,}prob\_obj\texttt{)}$$

prob_var: The name of the variable that represents the optimization problem

name: Name of the problem used in the output .lp file

prob_obj: The objective of the optimization problem, either "LpMininize" (default) or "LpMaximize"

Example:
```
>>> prob = LpProblem("Minimize the Manufacturing Cost", LpMinimize)
```

- Next, the problem decision variables are created using the function "`LpVariable()`". The format of the function is

$$decision\_var = \texttt{LpVariable(}name\texttt{, lowBound = }Lower\_Value...\texttt{,upBound = }Upper\_Value\texttt{,cat = }Type\texttt{)}$$

decision_var: Name of the decision variable

name: The name of the variable used in the output .lp file

Lower_Value: The lower bound on this variables' equal to minus infinity by default

Upper_Value: The upper bound on this variables' equal to plus infinity by default

Type: The category this variable is in, Integer, Binary or Continuous(default)

Example:
```
>>> x1 = LpVariable("Prod_A" ,lowBound = 0, upBound = None, cat = "Integer")
```

- For optimization problems with many constraints and variables, it is recommended to create dictionaries for decision variables. The format is as follows.

$$decision\_var = \texttt{LpVariable.dicts(}name\texttt{, }indexes\texttt{, lowBound = }Lower\_Value...\texttt{,upBound = }Upper\_Value\texttt{,cat =}Type\texttt{)}$$

decision_var: Name of the decision variable

name: The name of the variable used in the output .lp file. The underscore will be put between *name* and strings in *indexes*

indexes: A list of strings of the keys to the dictionary of LP variables, and the main part of the variable name itself

Lower_Value: The lower bound on this variables' equal to minus infinity by default

Upper_Value: The upper bound on this variables' equal to plus infinity by default

Type: The category this variable is in, Integer, Binary or Continuous(default)

Example:

```
>>> prod = ["Bottle", "Plate", "Bowl", "Glass"]
>>> product_vars = LpVariable.dicts("Product", prod, lowBound = 0, upBound = None,
cat = "Integer")
>>> print(product_vars)
>>> 'Plate':  Product_Plate, 'Bowl':  Product_Bowl, 'Bottle':  Product_Bottle, 'Glass':
Product_Glass
```

NOTE: The underscore "_" is automatically inserted between *name* and *indexes*.

- The next step is to enter the problem data into the problem variable with the "`+=`" operator. The objective function is entered first, with a comma "`,`" at the end of the statement and a short string that explain the objective function.

  Example:

  ```
  >>> prob += 10*x1 + 5*x2 + 7*x3, "Total Manufacturing Cost"
  ```

- The constraints are entered next using the same procedure as before.

  Example:

  ```
  >>> prob += x1 + x2 +3*x3 == 10, "Equality_1"
  >>> prob += x1 + x2 <= 4, "Inequality_1"
  >>> prob += 7*x1 + 6*x2 + 3*x3 >= 10, "Inequality_2"
  ```

- For more complicated problems, using dictionary data type, *list comprehension*, and "`lpSum()`" function for entering the objective function and constraints make the codes more compact and easier to modify.

- List comprehension is the syntax that allows lists to be built from other lists. The basic format of a list comprehension is

  newlist_var = [*out_express* `for` *input_var* `in` *input_list*]

  newlist_var: New list to be created

  out_express: An output expression producing elements of the output list from members of the input list

  input_var: A variable representing members of the input sequence.

  input_list: An input list

  Example:

  ```
  >>> prod = ["Bottle", "Plate", "Bowl", "Glass"]
  >>> new_prod = ["Prod_" + i for i in prod]
  >>> New_Ingre
  ['Prod_Bottle', 'Prod_Plate', 'Prod_Bowl', 'Prod_Glass']
  ```

- The "`lpSum(`*in_list*`)`" function is for summing a list of linear expressions where *in_list* is an input list of linear expressions.

  Example:

  ```
  >>> prod = ["Bottle", "Plate", "Bowl"]
  >>> product_vars = LpVariable.dicts("Product", prod, lowBound = 0, upBound = None,
  cat = "Integer")
  >>> costs = {"Bottle":1, "Plate":3, "Bowl":2}
  >>> prob += lpSum([costs[i]*product_vars[i] for i in prod]), "Total Cost"
  ```

- Once all information of the problem are entered, the function "`writeLP(`*filename.lp*`)`" is used to create a text file with extension "`.lp`" which represents the problem.

  Example:

  ```
  >>> prob.writeLP("ManufacProb.lp")
  ```

- To solve the optimization problem, use the function "`solve()`". The problem is solved using the solver that PuLP chooses. If the default solver is used, the input brackets after `solve()` are left empty, however the choice of solver can also be specified. In our class, we use GLPK.

  Example:

  ```
  >>> prob.solve(GLPK())
  ```

- The results of the solver call can be displayed as output. First, we request the status of the solution using the command "`prob.status`". The value of the status is returned as an integer, which must be converted to its text meaning using the `LpStatus` dictionary. Since `LpStatus` is a dictionary(dict), its input must be in square brackets. The possible statuses of the results are: *Not Solved, Infeasible, Unbounded, Undefined* or *Optimal*.

  Example:

  ```
  >>> print "Status:", LpStatus[prob.status]
  ```

- The variables and their resolved optimum values can be printed to the screen. The for `loop` makes variable cycle through all the problem variable names. Then, it prints each variable name, followed by an equal sign, followed by its optimum value, name and varValue are properties of the object variable.

  Example:

  ```
  >>> for v in prob.variables():
  >>>     print v.name, "=", v.varValue
  ```

- The optimal cost is printed to the screen using the command "`prob.objective`" where `objective` is an attribute of the object `prob`. The function `value` is used to ensure that the number is in the right format to be displayed.

  Example:

  ```
  >>> print "The optimal cost is = ", value(prob.objective)
  ```

3 Examples

Note: The following examples are taken from the document "*pulp Documentation, Release 1.4.6.*

Example 1 A cat food manufacturer would like to produce their products as cheaply as possible while maintaining the required amount of nutrients. In each 100g can of cat food, the contained nutrients must be as specified in Table 1. In this example, there are only two ingredients (i.e., chicken and beef), their cost and nutrients are listed in Table 2.

Table 1: Nutrient Requirements.

Nutrient	**Amount (gram)**
Protein	≥ 8.0
Fat	≥ 6.0
Fibre	≤ 2.0
Salt	≤ 0.4

Table 2: Ingredient's cost and nutrients (per gram of ingredient). The unit of cost is dollar while the unit of nutrients is gram.

Ingredient	**Cost**	**Protein**	**Fat**	**Fibre**	**Salt**
Chicken	0.013	0.100	0.080	0.001	0.002
Beef	0.008	0.200	0.100	0.005	0.005

Using the provided information, the mathematical representation of the problem is as follows.

$x1$: percentage of chicken meat in a can of cat food
$x2$: percentage of beef in a can of cat food

$$\begin{aligned} \text{minimize} \quad & 0.013x_1 + 0.008x_2 \\ \text{subject to} \quad & 1.000x_1 + 1.000x_2 = 100.0 \\ & 0.100x_1 + 0.200x_2 \geq 8.0 \\ & 0.080x_1 + 0.100x_2 \geq 6.0 \\ & 0.001x_1 + 0.005x_2 \leq 2.0 \\ & 0.002x_1 + 0.005x_2 \leq 4.0 \\ & x_1, x_2 >= 0 \end{aligned}$$

The Python scripts for solving the above problem is as follows.

```
#Import PuLP modeler functions
from pulp import *

#Create the "prob" variable to contain the problem data
prob = LpProblem("The Whiskas Problem", LpMinimize)

#The two variables Beef and Chicken are created with a lower limit of zero
x1 = LpVariable("ChickenPercent", lowBound = 0, upBound = None, cat = "LpInteger")
x2 = LpVariable("BeefPercent", lowBound = 0, upBound = None, cat = "LpInteger")

#The objective function is added to "prob" first
prob += 0.013*x1 + 0.008*x2, "Total Cost of Ingredients per can"

#The five constraints are entered
prob += x1 + x2 == 100, "Percentages Sum"
```

```
prob += 0.100*x1 + 0.200*x2 >= 8.0, "Protein Requirement"
prob += 0.080*x1 + 0.100*x2 >= 6.0, "Fat Requirement"
prob += 0.001*x1 + 0.005*x2 <= 2.0, "Fibre Requirement"
prob += 0.002*x1 + 0.005*x2 <= 0.4, "Salt Requirement"

#The problem data is written to an .lp file
prob.writeLP("FoodModel1.lp")

#The problem is solved using GLPK
prob.solve(GLPK())

#The status of the solution is printed to the screen
print "Status:", LpStatus[prob.status]

#Each of the variables is printed with it's resolved optimum value
for v in prob.variables():
   print v.name, "=", v.varValue

#The optimised objective function value is printed to the screen
print "Total Cost of Ingredients per can = ", value(prob.objective)
```

Example 2 In the following example, the nature of the problem is as same as the previous problem. However, there are more decision variables. Accordingly, it is better to utilize the dictionary data type and the list comprehension in the codes.

Table 3: Nutrient requirements per can.

Nutrient	**Amount (gram)**
Protein	≥ 8.0
Fat	≥ 6.0
Fibre	≤ 2.0
Salt	≤ 0.4

Table 4: Ingredients' cost and nutrients (per gram of each ingredient). The unit of cost is dollar while the unit of nutrients is gram.

Ingredient	**Cost**	**Protein**	**Fat**	**Fibre**	**Salt**
Chicken	0.013	0.100	0.080	0.001	0.002
Beef	0.008	0.200	0.100	0.005	0.005
Mutton	0.010	0.150	0.110	0.003	0.007
Rice	0.002	0.000	0.010	0.100	0.002
Wheat Bran	0.005	0.040	0.010	0.150	0.008
Gel	0.001	0.000	0.000	0.000	0.000

Using the provided information, the mathematical representation of the problem is as follows.

$x1$: percentage of chicken meat in a can of cat food
$x2$: percentage of beef in a can of cat food
$x3$: percentage of mutton in a can of cat food
$x4$: percentage of rice in a can of cat food
$x5$: percentage of wheat bran in a can of cat food
$x6$: percentage of gel in a can of cat food

$$
\begin{aligned}
\text{minimize} \quad & 0.013x_1 + 0.008x_2 + 0.010x_3 + 0.002x_4 + 0.005x_5 + 0.001x_6 \\
\text{subject to} \quad & x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 100.0 \\
& 0.100x_1 + 0.200x_2 + 0.150x_3 + 0.040x_5 \geq 8.0 \\
& 0.080x_1 + 0.100x_2 + 0.110x_3 + 0.010x_4 + 0.010x_5 \geq 6.0 \\
& 0.001x_1 + 0.005x_2 + 0.003x_3 + 0.100x_4 + 0.150x_5 \leq 2.0 \\
& 0.002x_1 + 0.005x_2 + 0.007x_3 + 0.002x_4 + 0.008x_5 \leq 0.4 \\
& x_1, x_2, x_3, x_4, x_5, x_6 >= 0
\end{aligned}
$$

The Python scripts for solving the problem is as follows.

```
#Import PuLP modeler functions
from pulp import *

#Creates a list of the Ingredients
Ingre = ["CHICKEN", "BEEF", "MUTTON", "RICE", "WHEAT", "GEL"]

#A dictionary of the costs of each of the Ingredients is created
costs = {"CHICKEN":  0.013,
        "BEEF":  0.008,
        "MUTTON":  0.010,
        "RICE":  0.002,
        "WHEAT":  0.005,
        "GEL":  0.001}

#A dictionary of the protein percent in each of the Ingredients is created
proteinPer = {"CHICKEN":  0.100,
        "BEEF":  0.200,
        "MUTTON":  0.150,
        "RICE":  0.000,
        "WHEAT":  0.040,
        "GEL":  0.000}

#A dictionary of the fat percent in each of the Ingredients is created
fatPer = {"CHICKEN":  0.080,
        "BEEF":  0.100,
        "MUTTON":  0.110,
        "RICE":  0.010,
        "WHEAT":  0.010,
        "GEL":  0.000}

#A dictionary of the fibre percent in each of the Ingredients is created
fibrePer = {"CHICKEN":  0.001,
        "BEEF":  0.005,
        "MUTTON":  0.003,
        "RICE":  0.100,
        "WHEAT":  0.150,
        "GEL":  0.000}

#A dictionary of the salt percent in each of the Ingredients is created
saltPer = {"CHICKEN":  0.002,
        "BEEF":  0.005,
        "MUTTON":  0.007,
        "RICE":  0.002,
        "WHEAT":  0.008,
        "GEL":  0.000}
```

```
#Create the "prob" variable to contain the problem data
prob = LpProblem("The Whiskas Problem", LpMinimize)

#A dictionary called "ingre_vars" is created to contain the referenced Variables
ingre_vars = LpVariable.dicts("Ingr",Ingre,0)

#The objective function is added to "prob" first
prob += lpSum([costs[i]*ingre_vars[i] for i in Ingre]), "Total Cost"

#The five constraints are added to "prob"
prob += lpSum([ingre_vars[i] for i in Ingre]) == 100, "PercentagesSum"
prob += lpSum([proteinPer[i] * ingre_vars[i] for i in Ingre]) >= 8.0, "ProteinRequirement"
prob += lpSum([fatPer[i] * ingre_vars[i] for i in Ingre]) >= 6.0, "FatRequirement"
prob += lpSum([fibrePer[i] * ingre_vars[i] for i in Ingre]) <= 2.0, "FibreRequirement"
prob += lpSum([saltPer[i] * ingre_vars[i] for i in Ingre]) <= 0.4, "SaltRequirement"

#The problem data is written to an .lp file
prob.writeLP("FoodModel2.lp")

#The problem is solved using GLPK
prob.solve(GLPK())

#The status of the solution is printed to the screen
print "Status:", LpStatus[prob.status]

#Each of the variables is printed with it's resolved optimum value
for v in prob.variables():
   print v.name, "=", v.varValue

#The optimised objective function value is printed to the screen
print "Total Cost of Ingredients per can = ", value(prob.objective)
```