 Previous: (5g) Column Generation

(5h) Column Generation 2

All LP's we have solved so far have been using **row-wise** modelling: the variables have been created and given bounds, the complete objective function has been entered on one line, and each of the constraints has been entered one line at a time (sometimes in a *for* loop) stating how the pre-created variables must relate to each other.
This means that it is easy to add a new constraint to the existing `prob` object, as was done in the Sudoku Problem. However, when a new variable (a pattern in the Cutting Stock Problem) is added, a new `prob` object must be created so that the different objective function and different constraints can be added. This was done on the first Column Generation page, but the entire process can be made much more efficient by using **column-wise modelling**.
In **column-wise modelling**, the problem data is added as columns (one column would be all the coefficients on any particular variable, across the objective function and all the constraints). This makes it much easier to add more variables.

A simple example of the familiar row-wise modelling:

```
from pulp import *
prob = LpProblem("test", LpMinimize)
x = LpVariable("x", 0, 4)
y = LpVariable("y", -1, 1)
z = LpVariable("z", 0)
prob += x + 4*y + 9*z, "obj"
prob += x+y <= 5, "c1"
prob += x+z >= 10, "c2"
prob += -y+z == 7, "c3"
prob.solve()
```

This same problem can be modelled using column-wise modelling:

```
from pulp import *
prob = LpProblem("test", LpMinimize)




obj = LpConstraintVar("obj")
prob.setObjective(obj)      # the obj variable is initialised in this standard way


a = LpConstraintVar("Ca", LpConstraintLE, 5)    # each constraint is created, but only the sign and R.H.S are specified
b = LpConstraintVar("Cb", LpConstraintGE, 10)
c = LpConstraintVar("Cc", LpConstraintEQ, 7)


prob += a   # each constraint is added to prob
prob += b
prob += c


x = LpVariable("x", 0, 4, LpContinuous, obj + a + b)      # each variable is added with it's bounds, category/type,
y = LpVariable("y", -1, 1, LpContinuous, 4*obj + a - c)   # and it's coefficient in the objective function and each
z = LpVariable("z", 0, None, LpContinuous, 9*obj + b + c) # of the constraints


prob.solve()
```

The main tricky part to columnwise modelling is entering the last parameter of the variable definitions correctly. Note that for constraint a, it has a coefficient of "+1" in the x and y variable definitions and is earlier specified to be less than or equal to 5. Therefore, all the information is specified to create "x+y <= 5" as was entered in full in row-wise modelling. Also note that a, b & c are all added to prob before the variables are created, and no more statements relating to prob need to made before the LP is solved. This is because obj, a, b & c are already added to prob.

Using this column-wise modelling, the Sponge Roll Problem can be solved more efficiently again. It is still in two files: A main file and a function file.

Main File

```
"""
The Sponge Roll Problem with Columnwise Column Generation for the PuLP Modeller



Authors: Antony Phillips,  Dr Stuart Mitchell  2008
"""


# Import Column Generation functions
from CGcolumnwise import *
```

After the standard introduction, a function called createMaster() is called. This will set up all aspects of the problem except for the pattern variables. i.e. it creates obj and assigns it to prob, it creates the constraints and adds them to prob (as above, when the constraints are created the only required specification is the R.H.S), it creates the surplus variables since they will not change. (It may be worthwhile scrolling down to see the createMaster function in the functions file)

```
# The Master Problem is created
prob, obj, constraints = createMaster()
```

A set of initial patterns which will make the problem solvable (not optimal) are required.

```
# A list of starting patterns is created
newPatterns = [[1,0,0],[0,1,0],[0,0,1]]
```

The newPatterns list will continue to be altered as more patterns are required to be added. Eventually the newPatterns list will be empty which will end the *while* loop

```
# New patterns will be added until newPatterns is an empty list
while newPatterns:
```

The addPatterns function will create the patterns in newPatterns as LpVariables.

```
    # The new patterns are added to the problem
    addPatterns(obj,constraints,newPatterns)
```

The masterSolve function is passed `prob` which is ready to solve, and the duals are passed out.

```
    # The master problem is solved, and the dual variables are returned
    duals = masterSolve(prob)
```

The subSolve function is still formulated as before (row-wise), receiving only the input of `duals` and returning the newPatterns list.

```
    # The sub problem is solved and a new pattern will be returned if there is one
    # which can reduce the master objective function
    newPatterns = subSolve(duals)
```

Once the newPatterns list is returned empty, the loop will end and one final solve of the problem is required which has non-relaxed integer constraints. The masterSolve function is designed to take the relax input and relax the variables, whilst still being able to use the same prob variable.

```
# The master problem is solved with Integer Constraints not relaxed
solution, varsdict = masterSolve(prob,relax = False)
```

The solution is printed.

```
# Display Solution
for i,j in varsdict.items():
    print i, "=", j



print "objective = ", solution
```

The main file is avialable here.

Function File CGcolumnwise.py

The start to the function file is the same as before except there is a class variable called numPatterns which is incremented each time the init function runs.

```
"""
Columnwise Column Generation Functions



Authors: Antony Phillips,  Dr Stuart Mitchell  2008
"""


# Import PuLP modeler functions
from pulp import *


class Pattern:
    """
    Information on a specific pattern in the SpongeRoll Problem
    """
    cost = 1
    trimValue = 0.04
    totalRollLength = 20
    lenOpts = ["5", "7", "9"]
    numPatterns = 0

    def __init__(self, name, lengths = None):
        self.name = name
        self.lengthsdict = dict(zip(self.lenOpts,lengths))
        Pattern.numPatterns += 1


    def __str__(self):
        return self.name


    def trim(self):
        return Pattern.totalRollLength - sum([int(i)*int(self.lengthsdict[i]) for i in self.lengthsdict])
```

createMaster

The createMaster function sets up the constraints and surplusVars so the rollData dictionary is required. The prob object is initialised.

```
def createMaster():

    rollData = {#Length Demand SalePrice
              "5":   [150,   0.25],
              "7":   [200,   0.33],
              "9":   [300,   0.40]}


    (rollDemand,surplusPrice) = splitDict(rollData)


    # The variable prob is created
    prob = LpProblem("MasterSpongeRollProblem",LpMinimize)
```

```
    # The variable obj is created and set as the LP's objective function
    obj = LpConstraintVar("Obj")
    prob.setObjective(obj)
```

Each constraint is logically named, given a sign and a R.H.S value. Each constraint is then added to prob and the constraints remain saved in a dictionary for use later when defining variables.

```
    # The constraints are initialised and added to prob
    constraints = {}
    for l in Pattern.lenOpts:
        constraints[l]= LpConstraintVar("Min" + str(l), LpConstraintGE, rollDemand[l])
        prob += constraints[l]
```

The surplus variables are created. The last parameter means that: the negative of the surplus price * surplus variable for each of the lenOpts, occurs in the objective function. It also means that the negative of the surplus variable for each of the length options is in the constraint for that length option. This column-wise variable definition is important to understand. The coefficient in front of the `obj` or the constraint is multiplied by the surplus variable and occurs in that respective constraint or the objective function. Note that this loop makes all three surplusVars in the objective function but only one in each constraint.

```
    # The surplus variables are created
    surplusVars = []
    for i in Pattern.lenOpts:
        surplusVars += [LpVariable("Surplus "+ i,0,None,LpContinuous, -surplusPrice[i] * obj - constraints[i])]

    return prob,obj,constraints
```

addPatterns

The addPatterns function is the first call inside the *while* loop. It's task is to create the patterns as LpVariable instances.

A *for* loop is used so that `i` becomes each of the pattern lists in newPatterns. (Since newPatterns is a list of lists)

```
def addPatterns(obj,constraints,newPatterns):

    # A list called Patterns is created to contain all the Pattern class
    # objects created in this function call
    Patterns = []
    for i in newPatterns:
```

Each pattern is checked that it does not use more cms of roll than are available

```
        # The new patterns are checked to see that their length does not exceed
        # the total roll length
        lsum = 0
        for j,k in zip(i,Pattern.lenOpts):
            lsum += j * int(k)
        if lsum > Pattern.totalRollLength:
            raise "Length Options too large for Roll"
```

Each pattern that is about to be added, is printed along with it's name. `Pattern.numPatterns` is a class variable that will increment each time the Pattern.__init__ function is run (i.e. when a new pattern is created as an instance of the Pattern class). Each of the patterns in newPatterns is created as a Pattern instance and added to the `Patterns` list.

```
        # The number of rolls of each length in each new pattern is printed
        print "P"+str(Pattern.numPatterns),"=",i


        # The patterns are instantiated as Pattern objects
        Patterns += [Pattern("P" + str(Pattern.numPatterns),i)]
```

The pattern variables are created. The lpSum term here just saves us from writing 'i.lengthsdict["5"]*constraints["5"] + i.lengthsdict["7"]*constraints["7"] + i.lengthsdict["9"]*constraints["9"]'. The last parameter of the LpVariable init function works the same as before, it just has more terms in this case. This is the end of the function - there is no output since the creation of LpVariable instances, referenced to `obj` and the `constraints`, adds the variables.

```
    # The pattern variables are created
    pattVars = []
    for i in Patterns:
        pattVars += [LpVariable("Pattern "+i.name,0,None,LpContinuous, (i.cost - Pattern.trimValue*i.trim())  obj\
         + lpSum([constraints[l]*i.lengthsdict[l] for l in Pattern.lenOpts]))]
```

masterSolve

The masterSolve works the same as in Column Generation, except it is already passed prob and so does not have to do so many steps.

If the relax parameter is passed as False, the variables will be set to Integer. This will only occur on the last run.

```
def masterSolve(prob,relax=True):

    # Unrelaxes the Integer Constraint
    if not relax:
        for v in prob.variables():
            v.cat = LpInteger
```

```
    # The problem is solved using CPLEX, with no message output and rounded
    prob.solve(CPLEX(msg=0))
    prob.roundSolution()
```

If the problem is relaxed then the call is from within the *while* loop and a duals dictionary is returned.

```
    if relax:
        # A dictionary of dual variable values is returned
        duals = {}
        for i,name in zip(Pattern.lenOpts,["Min5","Min7","Min9"]):
            duals[i] = prob.constraints[name].pi
        return duals
```

If the problem is not relaxed then the variable names, and their values (in varsdict), and the objective function value are returned.

```
    else:
        # A dictionary of variable values and the objective value are returned
        varsdict = {}
        for v in prob.variables():
            varsdict[v.name] = v.varValue

        return value(prob.objective), varsdict
```

subSolve

The subSolve function searches for more patterns that would reduce the objective function of the masterSolve further.

Most of this function is the same as it was in Column Generation.

```
def subSolve(duals):

    # The variable prob is created
    prob = LpProblem("SubProb",LpMinimize)


    # The problem variables are created
    vars = LpVariable.dicts("Roll Length", Pattern.lenOpts, 0, None, LpInteger)


    trim = LpVariable("Trim", 0 ,None,LpInteger)


    # The objective function is entered: the reduced cost of a new pattern
    prob += (Pattern.cost - Pattern.trimValue*trim) - lpSum([vars[i]*duals[i] for i in Pattern.lenOpts]), "Objective"


    # The conservation of length constraint is entered
    prob += lpSum([vars[i]*int(i) for i in Pattern.lenOpts]) + trim == Pattern.totalRollLength, "lengthEquate"


    # The problem is solved using CPLEX
    prob.solve(CPLEX(msg=0))


    # The variable values are rounded
    prob.roundSolution()
```

This is again similar to before, except the output is a list containing another list. The *if* statement is set to `< -10**-5` since otherwise some very small negative values which are meant to be zero (except became slightly negative due to floating point error) will cause a new pattern to be found.

```
    newPatterns = []
    # Check if there are more patterns which would reduce the master LP objective function further
    if value(prob.objective) < -10**-5:
        varsdict = {}
        for v in prob.variables():
            varsdict[v.name] = v.varValue
        # Adds the new pattern to the newPatterns list
        newPatterns += [[int(varsdict["Roll_Length_5"]),int(varsdict["Roll_Length_7"]),int(varsdict["Roll_Length_9"])]]

    return newPatterns
```

The full function file is available here.

 Previous: (5g) Column Generation