Previous: (5f) Sudoku As An LP

Next: (5h) Column Generation 2

(5g) Column Generation

Column Generation is an efficient way of solving large LPs. In the Cutting Stock Problem we created every possible pattern for lengths that could be cut from the 20cm rolls, and made each pattern a problem variable. However, if the roll length had been 100m long (for example), this would have required over 700 patterns to be created (with the same cutting options of 5cm, 7cm and 9cm). To overcome this problem we can initially solve the problem by explicitly defining just a few patterns to be considered. Then, given the dual variable values from this first solution, we can formulate a sub-problem to find if any other patterns have a negative reduced cost. This means that the new pattern would further reduce the objective function value. The main problem is re-solved with this new pattern and the dual values from this solve are used to find another pattern. When the reduced cost is greater than or equal to zero, the optimal solution is found since the objective function cannot be reduced any further.

A good explanation of the above process is under Delayed Column Generation at Wikipedia.

To implement the Column Generation process, 3 blocks of code are required to be written: A main file, a masterSolve function and a subSolve function. It is a good idea to create a separate file for the functions and the Pattern class definition.

Main File

A file header explains the purpose of the program and identifies the Author/s. The main file only needs to import the CG.py function library, and not PuLP. This is because the main file itself does not use any PuLP functions.

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



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


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

Since a loop is required later on to ensure more patterns are searched for until there are none that would reduce the objective function, the morePatterns variable must be initially set to True. The list of starting patterns must also be entered. Any starting patterns are adequate as long as the problem is feasible/solvable.

```
# The roll data is created
rollData = {#Length Demand SalePrice
            "5":   [150,   0.25],
            "7":   [200,   0.33],
            "9":   [300,   0.40]}



# The boolean variable morePatterns is set to True to test for more patterns
morePatterns = True


# A list of starting patterns is created
patternslist = [[4,0,0],[0,2,0],[0,0,2] ]
```

Each of the starting patterns are used to create an object of the class Pattern, in the list Patterns. This class is defined in the CG.py function file.

```
# The starting patterns are instantiated with the Pattern class
Patterns = []
for i in patternslist:
    Patterns += [Pattern("P" + str(len(Patterns)), i)]
```

The dual variable values from the master problem are passed as an input into the sub problem which passes out the amended Patterns list of Pattern class objects.

```
# This loop will be repeated until morePatterns is set to False
while morePatterns == True:

    # Solve the problem as a Relaxed LP
    duals = masterSolve(Patterns, rollData)


    # Find another pattern
    Patterns, morePatterns = subSolve(Patterns, duals)
```

The problem is solved for a final time with all added patterns, and the solution and variable values are passed as output. The problem is solved as a non-relaxed Integer Problem since having any particular pattern cut a non integer number of times is impossible.

```
# Re-solve as an Integer Problem
solution, varsdict = masterSolve(Patterns, rollData, relax = False)
```

Displays the solution variables and objective value.

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




print "objective = ", solution
```

The main file is available here.

Function File CG.py

The second file contains the Pattern class definition, and the functions of masterSolve & subSolve. Before these it is important to import the PuLP functions:

```
"""
Column Generation Functions



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


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

Class Definition

The Pattern class has 4 pattern constants (cost, trimValue, totalRollLength & lenOpts) defined inside it as class variables. There are 3 functions inside the Pattern class:
__init__: creates a Pattern object and assigns the **name** and **lengthsdict** attributes out of the inputs of the name and a list of the number of rolls of each length in that pattern.
__str__: returns the name of a pattern.
trim: returns the trim of a pattern, calculated using the class variables and attributes of that pattern.

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

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


    def __str__(self):
        return self.name


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

masterSolve Function

This function simply solves the Sponge Roll Problem LP and returns the relevant result depending on if this is the final call to this function (when the LP is not relaxed).

The only required inputs are the list of Pattern objects (Patterns), the rollData and whether or not the LP is relaxed. The rollData is split normally and prob is created.

```
def masterSolve(Patterns, rollData, relax = True):




    # The rollData is made into separate dictionaries
    (rollDemand,surplusPrice) = splitDict(rollData)


    # The variable prob is created
    prob = LpProblem("Cutting Stock Problem",LpMinimize)
```

Depending on the value of the input variable **relax**, the problem variables are made to be either LpInteger or LpContinuous.

```
    # vartype represents whether or not the variables are relaxed
    if relax:
        vartype = LpContinuous
    else:
        vartype = LpInteger


    # The problem variables are created
    pattVars = LpVariable.dicts("Pattern", Patterns, 0, None, vartype)
    surplusVars = LpVariable.dicts("Surplus", Pattern.lenOpts, 0, None, vartype)
```

The objective function and constraints are logically added to the prob variable. The class variables are used and the Pattern object list **Patterns** is used in several list comprehensions.

```
    # The objective function is entered: (the total number of large rolls used * the cost of each) -
    # (the value of the surplus stock) - (the value of the trim)
    prob += lpSum([pattVars[i]*Pattern.cost for i in Patterns]) - lpSum([surplusVars[i]*surplusPrice[i]\
            for i in Pattern.lenOpts]) - lpSum([pattVars[i]*i.trim()*Pattern.trimValue for i in Patterns])


    # The demand minimum constraint is entered
    for j in Pattern.lenOpts:
        prob += lpSum([pattVars[i]*i.lengthsdict[j] for i in Patterns]) - surplusVars[j]>=rollDemand[j],"Min%s"%j
```

The problem is solved using CPLEX and with no output messages. The solution variables are then rounded, otherwise values that were meant to be exact values may be slightly off due to floating point representation error.

```
    # The problem is solved using CPLEX with no output
    prob.solve(CPLEX(msg=0))



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

Since the state of relaxation of the LP determines what output is necessary, an *if* statement is used. When the LP is relaxed, it is not the final run through masterSolve and we are still looking for more patterns. Therefore the required output is the dictionary of dual variables values

```
    if relax:
            # Creates a dual variables list
            duals = {}
            for name,i in zip([Min5,Min7,'Min9'],Pattern.lenOpts):
                duals[i] = prob.constraints[name].pi

            return duals
```

Alternatively, if the problem is not relaxed then the function needs to return the value of the objective function, and the optimum variable values. The function also prints the number of rolls of each length in each pattern name, so the solution can be fully interpreted.

```
    else:
            # Creates a dictionary of the variables and their values
            varsdict = {}
            for v in prob.variables():
                varsdict[v.name] = v.varValue

            # The number of rolls of each length in each pattern is printed
            for i in Patterns:
                print i, " = %s"%[i.lengthsdict[j] for j in Pattern.lenOpts]


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

subSolve Function

This function searches for another pattern to add to the master LP which would reduce the objective function. This is done by minimising the reduced cost of any potential new pattern. If the minimum value is less than zero, it is worth adding. The reduced cost of a pattern is calculated by the cost of that pattern minus the sum of the dual variables on each of the length constraints multiplied by the number of rolls of that length in the new pattern.

The inputs to the subSolve function are the list of Pattern objects, and the duals dictionary. The prob variable is created first.

```
def subSolve(Patterns, duals):

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

A variable for the number of each length roll in the pattern is created, along with a variable for the trim length.

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

The reduced cost of a new pattern is entered. Note that the cost of adding the pattern is it's cost ($1 in this case) minus the money gained back from sale of trim. This is because the amount of trim is a fixed attribute for any given pattern.

```
    # 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 only constraint to this problem is that the total length of the trim and the smaller cut rolls must be equal to the length of the initial roll.

```
# 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 and the results are rounded.

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


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

The new pattern is written to a dictionary so that there is no problem with the values being out of order.

```
# The new pattern is written to a dictionary
varsdict = {}
newPattern = {}
for v in prob.variables():
    varsdict[v.name] = v.varValue
for i,j in zip(Pattern.lenOpts,["Roll_Length_5","Roll_Length_7","Roll_Length_9"]):
    newPattern[i] = int(varsdict[j])
```

If the value of the objective function is negative then this new pattern will reduce the value of master LP objective function and so it is added to the Patterns list, as a new instance of the Pattern class. Otherwise, morePatterns is set to False, so that the loop in the main function will end. Some values of prob.objective would be *very* small negatives that were meant to be zero, but took a non-zero value due to floating point representation error. To stop these values being interpreted as negatives (when they should be zero), value(prob.objective) is tested to be less than -10^-5.

```
# Check if there are more patterns which would reduce the master LP objective function further
if value(prob.objective) < -10**-5:
    morePatterns = True # continue adding patterns
    Patterns += [Pattern("P" + str(len(Patterns)), [newPattern[i] for i in ["5","7","9"]])]
else:
    morePatterns = False # all patterns have been added


return Patterns, morePatterns
```

The function file is available here.

 Previous: (5f) Sudoku As An LP

 Next: (5h) Column Generation 2