Previous: (5d) A Facility Location Problem

Next: (5f) Sudoku As An LP

(5e) A Cutting Stock Problem

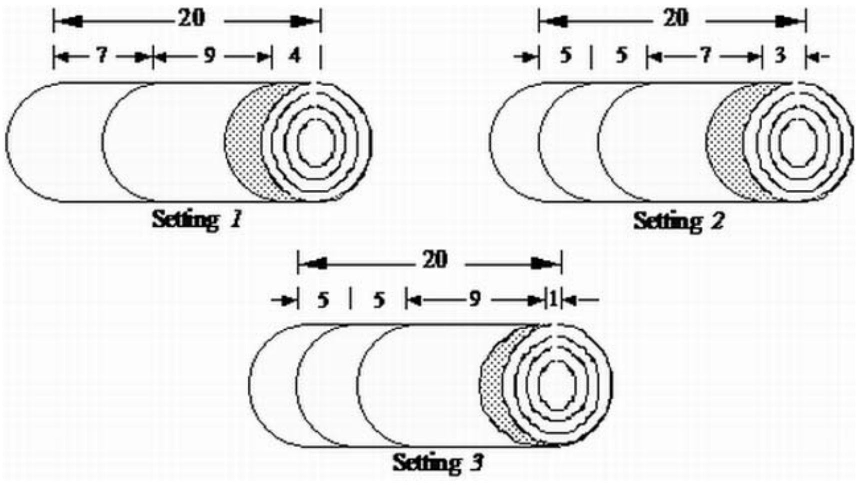
The Sponge Roll Production Problem

Problem Description

The Better Food Company produces cream-filled sponge rolls with a standard width of 20 cm each. Each 20cm roll costs the company $1.00 to produce. Special customer orders with different widths are produced by cutting (slitting) the standard rolls of sponge into shorter lengths. The fixed daily orders are summarized in the following table. These orders need to be met at least cost.

Order	Desired Width (cm)	Desired Number of Rolls
A	5	150
B	7	200
C	9	300

An order is filled by setting the cutting knives to the desired widths. Usually, there are a number of ways in which standard rolls can be slit to fill a given order. The figure below shows three possible knife settings for the 20-cm roll. Although there are other feasible settings, we limit the discussion for the moment to considering settings 1, 2 and 3 in the figure. Note that the shaded area in each diagram represents lengths of sponge that are too short to be used in meeting orders, and so these pieces must be thrown away. Such wastage is called *trim loss*.



The Better Food Company wants to know how to cut the 20cm rolls to minimise the cost of meeting their customer orders.

Formulation

The Sponge Roll Production Problem is an example of a *Cutting Stock* model. Cutting Stock models are a specialisation of *Set Covering* models. The generalised set covering model is:

$$\begin{aligned} &\min \quad c^T x \\ &\text{subject to } Ax \geq b \\ &\qquad x \geq 0, x \text{ integer} \end{aligned}$$

For cutting stock models, each column of the model represents a possible cutting pattern and each constraint represents a requirement for the patterns (in The Sponge Roll Production Problem the requirements are the customer orders). Much of the complexity in cutting stock models (and indeed set covering models in general) is removed during the generation of the columns. Each column represents a feasible pattern, but the complexity of producing feasible patterns is removed from the mathematical programme.

When formulating we will consider two alternatives:
1. The usual formulation (by row);
2. The columnwise formulation.

1. **Identify the Decision Variables**

   The decision variables are the number of times to use each sensible cutting pattern.

2. **Formulate the Objective Function**

Each cutting pattern has an associated cost, in this case the cost of the 20cm roll used (i.e., $1.00). The objective function is the cost of the total number of 20cm rolls required to be cut.

3. **Formulate the Constraints**

   The constraints ensure that the desired number of smaller rolls are produced by cutting the 20cm rolls.

Solution

The introductory commenting and import statement is entered.

```
"""
The Sponge Roll Problem for the PuLP Modeller



Authors: Antony Phillips, Dr Stuart Mitchell   2007
"""


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

A list of the roll lengths is entered and a dictionary linking the roll lengths to their demand.

```
# A list of all the roll lengths is created
LenOpts = ["5","7","9"]



# A dictionary of the demand for each roll length is created
rollDemand = {"5": 150,
              "7": 200,
              "9": 300}
```

A list of the pattern names we are analysing is entered, and then a list (set out to be viewed as a table) of the number of rolls each pattern can make, for each roll length.

```
# A list of all the patterns is created
PatternNames = ["A","B","C"]



# Creates a list of the number of rolls in each pattern for each different roll length
patterns = [#A B C
            [0,2,2],# 5
            [1,1,0],# 7
            [1,0,1] # 9
            ]
```

The cost of each 20cm roll to be cut up at $1 is entered.

```
# The cost of each 20cm long sponge roll used
cost = 1
```

The pattern table is made into a dictionary (as has been done in previous examples), so that 'Patterns["7"]["B"]' will return the number of rolls of length 7 when cutting according to pattern B.

```
# The pattern data is made into a dictionary
patterns = makeDict([LenOpts,PatternNames],patterns,0)
```

The problem variables are created and will be of the form `Patt_A, Patt_B, Patt_C,` due to the first two parameters. A negative number of rolls cut into the patterns cannot happen, so the lower bound is zero, and there is no upper bound. The number of 20cm rolls cut up into patterns must also be an integer.

```
# The problem variables of the number of each pattern to make are created
vars = LpVariable.dicts("Patt",PatternNames,0,None,LpInteger)
```

The variable `prob` is created.

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

The objective function is added to `prob`. This is the number of each pattern used multiplied by the the cost of each 20cm roll cut up.

```
# The objective function is entered - the total number of large rolls used * the fixed cost of each
prob += lpSum([vars[i]*cost for i in PatternNames]),"Production Cost"
```

The constraints are added to `prob`. In this case, the only constraints are ensuring that the demand for each shorter roll length is met.

```
# The demand minimum constraint is entered
for i in rollLengths:
    prob += lpSum([vars[j]*patterns[i][j] for j in PatternNames])>=rollDemand[i],"Ensuring enough %s cm rolls"%i
```

After the constraints is the usual end to the Python formulation starting with the `prob.writeLP` line. The full working file is available here.

Post-Optimal Analysis

One common extension for cutting stock problems is the sale of any extra products and/or the trim loss at discounted prices. For example, The Better Food Company can sell any excess sponge rolls for: 15c for 5cm rolls, 25c for 7cm rolls and 35c for 9cm rolls. They can also get rid of any trim loss for 4c per cm.

To incorporate the selling of excess sponge rolls, the following is added/changed:

Since each roll length now has both an associated demand and an associated surplus cost, we put these into a list in a dictionary and then use splitDict. These two sections of code replace the `rollDemand` dictionary from the simple formulation. The `surplusPrice` dictionary can now be used.

```
 # A dictionary of the demand and surplus sale price of each roll length is created
rollData = {#Length Demand SalePrice
              "5":   [150,   0.25],
              "7":   [200,   0.33],
              "9":   [300,   0.40]}



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

A new set of problem variables need to be created, and so the `vars` dictionary in the problem above needs to be more specifically renamed `pattVars`. The new additional variable dictionary is `surplusVars`, linking the the keys of the roll lengths ("5","7" and "9") to the variables `Surp_5`, `Surp_7` and `Surp_9` which are the number of rolls of surplus of each length.
The objective function has another lpSum term which is subtracted from the original. This term represents the sum of the money made from selling the surplus of each roll length. Lastly, an additional term is put into the demand constraint equation, because each roll that is counted as surplus cannot be used to fill the roll demand. [Note: the slash \ is just a line continuation operator and has only been included so the line image could fit onto the screen. Using PyDev or IDLE, this will not be needed since there is a horizontal scroll]

```
# The problem variables of the number of surplus rolls for each length are created
surplusVars = LpVariable.dicts("Surp",rollLengths,0,None,LpInteger)
```

```
# The objective function is entered - the total number of large rolls used * the fixed cost of each minus the surplus sales
prob += lpSum([pattVars[i]*cost for i in PatternNames]) - \
        lpSum([surplusVars[i]*surplusPrice[i] for i in LenOpts]),"Net Production Cost"



# The demand minimum constraint is entered
for i in LenOpts:
    prob += lpSum([pattVars[j]*patterns[i][j] for j in PatternNames]) - surplusVars[i]\
    >= rollDemand[i],"Ensuring enough %s cm rolls"%i
```

Running this successfully (before adding trim loss) will result in a objective function value of 287.5

The trim loss is incorporated into the model differently to the surplus since the trim sold in each pattern is exactly known before the model runs, and the value of trim sold is simply based on how rolls were cut into each pattern.
Firstly, the sale value for each centremetre of trim is entered and the number of centremetres of trim in each pattern is added to a dictionary. This could actually be calculated using loops, but we have explicitly entered it instead, for this simple model with only 3 pattern choices.

```
# A dictionary of the number of cms of trim in each pattern is created
trim = {"A": 4,
        "B": 2,
        "C": 1}
```

```
# The sale value of each cm of trim
trimValue = 0.04
```

Lastly, a trim term is put into the objective function after the surplus term. It represents the value of the trim sold and has the meaning: the number of each pattern created multiplied by the number of centremetres of trim caused by using that pattern multiplied by the value of each centremetre of trim.

```
# The objective function is entered - the total number of large rolls used * the fixed cost of each minus the surplus sales, and the trim sales
prob += lpSum([pattVars[i]*cost for i in PatternNames]) - lpSum([surplusVars[i]*surplusPrice[i] for i in LenOpts]) \
- lpSum([pattVars[i]*trim[i]*trimValue for i in PatternNames]),"Net Production Cost"
```

This should give you an objective function value of $251.5

The full working file with surplus and trim included is available here.

Validation

We need to make sure the number of each pattern cut is integer. We should check to make sure we have met our demand correctly, i.e., ensure the patterns are correctly defined. We should make sure that the number of rolls cut in a pattern will fit into a 20cm roll, i.e., the pattern is valid.

Presentation of Solution and Analysis

Your management summary for The Sponge Roll Production should contain a brief problem description, some discussion on your method for generating cutting patterns and a description of the patterns used. It should then present a summary of the optimal use of the cutting patterns.

Implementation and Ongoing Monitoring

To implement your solution you should ensure that the cutting patterns you are using can be set up of the cutting machines. You could also check for hidden set-up costs, i.e., costs incurred by switching from one cutting pattern to the next. Ongoing monitoring should take the form of consistent checking of the product line. Introducing new products means that you will need to regenerate your cutting patterns. As usual, monitoring the costs and demands for any beneficial opportunities should take place.

Extra for Experts

In the above problem we only analysed 3 different patterns for cutting the sponge rolls, whereas there are in fact many different cutting patterns that are worth considering, as you will remember from STATS 255. All the patterns should be considered and the rest for this case can be calculated simply by hand. However, supposing the sponge roll was longer, or many different lengths were required instead of just three, it would take too long to solve by hand. The calculation of these other patterns can be done in Python at the start of your code so you do not need to explicitly enter them.

This is done by using 2 logical functions:

The first is the function that calculates the patterns themselves using a series of for loops. This function will only work for length options lists of size 3. The code works quite simply:
`j` represents the number of length 5 rolls to be cut, `k` represents the number of length 7 rolls to be cut and `l` represents the number of length 9 rolls to be cut. The range of each of these loops is the logical range that `j`,`k` and `l` can take. That is, a minimum value of zero, and a maximum value of how many of that length could be cut if the full length of the roll was cut into pieces that size. e.g. for `j`, the maximum possible value is 4 which is computed with 'int(totalRollLength/lenOpts[1])'. The `int` function rounds the result of the inner calculation down to the nearest integer (i.e. it truncates off the decimal part). The `+1` ensures that the top value is reached when iterating through the for loop.

Inside the for loops, the sum of the lengths for that combination is calculated. If this sum is less than or equal to the total roll length then the cutting pattern is possible. In STATS255, the patterns with trim of 5 or more would not have been used, however there is no harm in having such patterns at [0,0,0] whereby there is 20cm trim and no rolls. Therefore they are calculated along with the other patterns. Supposing the trimValue was greater than 0.05, and the cost of each roll was still 1.00, this would actually make the problem have a negative infinity cost, since the [0,0,0] pattern would make money.

```
def calculatePatterns(totalRollLength,lenOpts):
    """
     The patterns are created using for loops

     The inputs are:
     totalRollLength - the length of the roll
     lenOpts - a list of the sizes of cutting options


     It returns the list of patterns


     Authors: Antony Phillips, Dr Stuart Mitchell    2007
    """


    patterns =[]


    # All possible combinations of cutting patterns are trialed, and the feasible/sensible patterns are added to the list
    for j in range (0,int(totalRollLength/lenOpts[0])+1):
        for k in range (0, int(totalRollLength/lenOpts[1])+1):
            for l in range (0, int(totalRollLength/lenOpts[2])+1):
                lengthSum = j*lenOpts[0]+k*lenOpts[1]+l*lenOpts[2]
                if lengthSum <= totalRollLength:
                    patterns += [[j,k,l]]


    return patterns
```

Since the patterns are now calculated, they need to be assigned names and a trim value automatically too. This is done using another function which calls the calculatePatterns function. Whilst both functions could have been combined into one (or we could have even used no functions), it is much simpler for the reader to interpret it this way.

Apart from calling calculatePatterns, the makePatterns function does 3 key things:
- Creates a name list to correspond to each of the patterns (P0,P1...)
- Calculates the trim for each pattern and puts it in a dictionary, with the pattern name as the reference key
- Prints the name of the patterns next to the list of the number of rolls of each length in that pattern

```
def makePatterns(totalRollLength,lenOpts):
    """
     Makes the different cutting patterns for a cutting stock problem.

     The inputs are:
     totalRollLength : the length of the roll
     lenOpts: a list of the sizes of cutting options as strings


     Authors: Antony Phillips, Dr Stuart Mitchell    2007
    """


    # calculatePatterns is called to create a list of the feasible cutting options in tlist
    patterns = calculatePatterns(totalRollLength,lenOpts)


    # The list PatternNames is created
    PatternNames = []
    for i in range(len(patterns)):
        PatternNames += ["P"+str(i)]


    # The amount of trim (unused material) for each pattern is calculated and added to the dictionary
    # trim, with the reference key of the pattern name.
    trim = {}
    for name,pattern in zip(PatternNames,patterns):
        ssum = 0
        for rep,l in zip(pattern,lenOpts):
            ssum += rep*l
        trim[name] = totalRollLength - ssum
    # The different cutting lengths are printed, and the number of each roll of that length in each
    # pattern is printed below. This is so the user can see what each pattern contains.
    print "Lens: %s" %lenOpts
    for name,pattern in zip(PatternNames,patterns):
        print name + "  = %s"%pattern


    return (PatternNames,patterns,trim)
```

The final changes must be made to the main code to use these functions. PatternNames, patterns and trim no longer need to be created explicitly in the main function and instead this line is put in:

```
(PatternNames,patterns,trim) = makePatterns(totalRollLength,[int(l) for l in LenOpts])
```

The second argument is the length options list, but it is altered so that all the values are integers instead of strings.

The last alteration that must be made is the argument order in the makeDict function. This is because when our patterns list is created using a function, it is actually transposed from the way we entered it explicitly in the first example. The makeDict line should be:

```
patterns = makeDict([PatternNames,LenOpts],patterns,0)
```

Since the code above has been written using functions, any function can be altered to be more efficient or versatile. In the above formulation, the only factor limiting LenOpts lists to length 3 is the calculatePatterns function. This function can be made recursive so it can calculate the pattern list for an input LenOpts of any size. An example of how this can be done is shown below: (when this is called, pass an empty list '[]' as the starting head variable)

The full working code using this recursive function is available here. The objective function value is now $246.5

```
def calculatePatterns(totalRollLength,lenOpts, head):
    """
     Recursively calculates the list of options lists for a cutting stock problem. The input
     tlist is a pointer, and will be the output of the function call.

     The inputs are:
     totalRollLength - the length of the roll
     lenOpts - a list of the sizes of remaining cutting options
     head - the current list that has been passed down though the recusion


     Returns the list of patterns


     Authors: Bojan Blazevic, Dr Stuart Mitchell    2007
    """
    if lenOpts:
        patterns =[]
        #take the first option off lenOpts
        opt = lenOpts[0]
        for rep in range(int(totalRollLength/opt)+1):
            #reduce the length
            l = totalRollLength - rep*opt
            h = head[:]
            h.append(rep)

            patterns.extend(calculatePatterns(l, lenOpts[1:], h))
    else:
        #end of the recursion
        patterns = [head]
    return patterns
```

Lastly, for people who prefer using an Object-Oriented approach, it is possible to define a Pattern class. This object-oriented method is used later in the Column Generation sections, and it is a good idea to understand how it works. This code can be found here.

Previous: (5d) A Facility Location Problem

Next: (5f) Sudoku As An LP