OPERATIONAL RESEARCH
CUTTING-STOCK PROBLEM (PART 1)

Felipe Dornelas <contact@felipedornelas.com>
2012/12

---

1. PROBLEM DOMAIN

The non-relaxed formulation for the master problem domain is:

    minimize ∑ {j = 1..n} x[j]   : *minimize the number of cut bars*
    s.t.

    for i in {i..m}:

        ∑ {j = 1..n} (x[j] × a[i,j]) = b[i]

    x >=0, x in Z^n

where:

- m: the number of different widths w[i] that need to be cut.

- b[i], i in {1..m}: the given demands that need to be met for each sub-bar with width w[i].

- n: the number of distinct cutting patterns A_j. We will have one structural variable for each cutting pattern. Compared to m, n might be a much larger number.

- x[j], j in {1..n}: the number of cutting patterns A_j used to satisfy the problem constraints.

- A = a[i,j], i in {1..m}, j in {1..n}: the matrix of all cutting patterns. Each column A_j from this matrix represents a cutting pattern and is tied to a structural matrix x[j]. For a given j, a cutting pattern A_j is a vector where each element a[i,j] represents the number of sub-bars of width w[i] that are cut by this pattern j. For instance, for a given j and m = 9, the following amounts of sub-bars w[i] are produced:

```
A_j = transpose([0, 0, 3, 4, 9, 0, 4, 6, 2])

w[1] = 0
w[2] = 0
w[3] = 3
w[4] = 4
w[5] = 9
w[6] = 0
w[7] = 4
w[8] = 6
w[9] = 2
```

We will use a linear relaxation to solve this problem, thus we relax the original formulation by making:

    x >= 0, x in R

The objective function will remain unchanged.

The strategy we will use to generate the matrix A is to start with a feasible canonical set of columns, and then iteratively add columns A_j = a[i] that will be the solution for the following sub-problem:

```
maximize ∑ {i = 1..m} (p[i] × a[i])   : maximize the use of a bar's length,
                                        minimize waste.
s.t.

    ∑ {i = 1..m} (w[i] × a[i]) <= W

a >= 0, a in Z^m
```

where:

- W: the fixed width of the master bar to be cut.

- a[i], i in {1..m}: the number of cuts of width w[i] produced, i.e., the column A_j.

- p[i]: the vector of dual variables from the solution of the current linear relaxation instance using the SIMPLEX method. ***TODO:*** *I don't know how to explain p in this model. I got this information from Rosklin.*

This sub-problem can be mapped to the unbounded knapsack problem, where w[i] are the weights and p[i] are the values of the items i to be put in a knapsack with capacity W.

We will use the following algorithm:

```
# Start with a feasible matrix A = a[i,j] with m cutting-patterns
# A_j, each containing only one cut of width w[j].
n ← m
for j in {1..n}:
    a[j,j] = 1

repeat:

    # Solves the linear relaxation using SIMPLEX.
    # x[j], j in {1..n} is the vector of primal variables
    # p[i], i in {1..m} is the vector of dual variables associated
    # with this solution.
    (x, p) ← SIMPLEX(A, b)

    # Solves the unbounded knapsack problem, with capacity W, item values
    # p[i] and item weights w[i], where i in {i..m}. This solution
    # generates v, the knapsack optimal total value, and the vector
    # A_j = a[i] containing the number of items i that were put into the
    # knapsack.
    (v, A_j) ← UNBOUNDED_KNAPSACK(p, w, W)

    # The reduced cost of this new column A_j is c_j = 1 - p × A_j.
    # However, since v = p × A_k, the total value of the knapsack, we have
    # that if v <= 1 all reduced costs are positive and the problem can
    # not be further optimized.
    if v <= 1:
        STOP
    else:
        Adds A_j to the matrix A
        n ← n + 1
end.
```

2. IMPLEMENTATION ARCHITECTURE

This homework is implemented in ANSI/ISO C++ using the GLPK library and an object-oriented architecture.

The following modules are defined.

Main.cpp:

Contains the program entry point and also handles input and output.

CuttingStock.{cpp,hpp}:

This module defines the cutting-stock solver (class CuttingStockSolver), along with the data transfer classes CuttingStockInstance, representing a particular problem instance, and CuttingStockSolution, representing its solution.

The solver receives a CuttingStockInstance object, solves the linear relaxation of the problem instance using the GLPK SIMPLEX solver, and returns a CuttingStockSolution.

A problem instance is modeled as a set of orders, where each order is a tuple:

(*index*, *width*, *pieces*)

where *index* is the integer index i, *width* is the width w[i], and *pieces* is the number of pieces b[i] of w[i] that will need to be cut.

A problem solution contains the values of the variables x and the solution to the objective function. ***TODO:*** *the matrix of cutting-patterns A is not passed to to CuttingStockSolution yet.*

It's also defined the class CuttingStockSolverStatistics, solely for statistics and runtime measurement purposes.

CuttingStockInstanceBuilder.{hpp,cpp}:

This class parses a input string stream, building a CuttingStockInstance object. Used to process the standard input.

Knapsack.hpp:

This module defines generic classes to be used by any solver of any instance of the Knapsack problem, be it 01-Knapsack, Unbounded Integer Knapsack or Fractional Knapsack.

The template class KnapsackInstance represents a generic Knapsack problem instance, and it's a indexed collection of items (class KnapsackItem). Each item is a tuple:

(*value*, *weight*)

representing the value and the weight of the item.

You may have noticed that the template classes KnapsackItem and KnapsackInstance receives two generic parameters, ValueType and WeightType. They specify both the type of the value and weight properties of an item, let them be integer, floating point or boolean types.

Likewise, it's defined the output class KnapsackSolution, to store the solution of a generic Knapsack problem instance. It contains a indexed collection of the quantities of the items that will be put in the

knapsack, and also the knapsack optimal value. This generic class receives the template parameters ValueType and AmountType, to specify the type of the knapsack values and the type of the item amounts. For the Unbounded Knapsack problem, AmountType would be of integer type, but for the Fractional Knapsack it would be a floating-point type.

IntegerKnapsack.{hpp,cpp}:

This module leverages the abstract Knapsack.hpp through a concrete implementation of the Unbounded Integer Knapsack problem.

The class IntegerKnapsackInstance extends KnapsackInstance, defining double as the value type and integer as the weight type. Likewise, the class IntegerKnapsackSolution extends KnapsackSolution obviously defining integer as the amount type.

The class IntegerKnapsackSolver is a solver for the Unbounded Integer Knapsack problem using a dynamic programming algorithm, defined on [1].

FloatingPoint.{cpp,hpp}:

This module defines methods for comparing floating point numbers with a pre-defined approximation factor, namely floatingPointCompare and floatingPointEqual. For instance:

floatingPointEqual(0,0000012, 0,0000015)

will yeld true if the approximation factor is 10^(-6).

Time.{cpp,hpp}:

Defines these classes Time and Timer. These are utility classes used to measure runtime.

3. REFERENCES

[1] Lecture 10, "Solving Integer Knapsack Problems by DP", by Dr. Yun-Bin Zhao from University of Birmingham. http://web.mat.bham.ac.uk/Y.Zhao/Lect-Notes/IP-lectures

[2] GNU Linear Programming Kit Reference Manual