Contents

1 The Cutting-stock problem

The **Cutting-stock problem** (**CSP**) is the problem concerning cutting standard-sized pieces of stock material, called *rolls*, into pieces of specified sizes, **minimizing material wasted**.

In order to formalize that problem as an **integer linear programming** (**ILP**), suppose that stock material width is equal to W, while m customers want n_i rolls, each of which is wide w_i, where $i = 1, ..., m$. Obliviously $w_i \leq W$.

1.1 *Kantorovich* ILP Formulation

$$\begin{array}{llll} (P1) & min & \sum_{k \in K} y_k & \\ & s.t. & \sum_{k \in K} x_i^k \geq n_i, \quad i = 1, ..., m & (demand) \\ & & \sum_{i=1}^{m} w_i x_i^k \leq W y_k, \quad \forall k \in K & (widthlimitation) \\ & & x_i^k \in \mathbb{Z}_+, y_k \in \{0, 1\} & \end{array} \tag{1}$$

1.2 *Gilmore* and *Gomory* ILP formulation

- An alternative, stronger formulation, is due to Gilmore and Gomory, which main idea is **to enumerate all possible raw cutting patterns**.
- A pattern $j \in J$ is described by the vector $(a_{1j}, a_{2j}, ..., a_{mj})$, where a_{ij} represents the number of final rolls of width w_i obtained from cutting a raw roll according to pattern j.
- In this model we have an integer variable x_j for each pattern $j \in J$, indicating how many times pattern j is used; in other words it represents how many raw rolls are cut according to pattern j.

$$\begin{array}{llll} (P2) & min & \sum_{j=1}^{n} x_j & \\ & s.t. & \sum_{j=1}^{n} a_{ij} x_j \geq n_i, \quad i = 1, ..., m & (demand) \\ & & x_j \in \mathbb{Z}_+, j = 1, ..., n & \end{array} \tag{2}$$

$$\begin{array}{llll} (LPMP) & min & \sum_{j=1}^{n} x_j & \\ & s.t. & \sum_{j=1}^{n} a_{ij} x_j \geq n_i, \quad i = 1, ..., m & (demand) \\ & & x_j \in \mathbb{R}_+, j = 1, ..., n & \end{array} \tag{3}$$

Recall that the dual problem of (LPM) is

$$
\begin{aligned}
(DLPMP) \quad & max \quad \sum_{i=1}^{m} n_i \pi_i \\
& s.t. \quad \sum_{i=1}^{m} a_{ij}\pi_i \leq 1 \qquad j \in P, \\
& \qquad \pi_i \geq 0, i = 1, ..., m
\end{aligned}
\tag{4}
$$

4

The which is called Linear Programming Master Problem. The solution to (LPM) could be fractional, It is possible to round up the fractional solution to get a feasible solution to

Where n represents the total number of cutting patterns satisfying following relations:

$$
\begin{gathered}
\sum_{i=1}^{m} w_i a_{ij} \leq W \\
a_{ij} \in \mathbb{Z}_+
\end{gathered}
\tag{5}
$$

- **How many cutting patterns exist?**

  In general, the number of possible patterns **grows exponentially** as a function of m and it can easily run into the millions. So, it may therefore become impractical to generate and enumerate all possible cutting patterns.

  Even if we had a way of generating all existing cutting pattern, that is all columns, the **standard simplex algorithm** will need to calculate the reduced cost for each non-basic variable, which is, from a computational point of view, impossible when n is huge because is very easy for any computer to go **out of memory**.

- Fortunately, another better approach, which we have used in our project, exists: **column generation method**.

This method solves the cutting-stock problem by starting with just a few patterns and it generates additional patterns when they are needed. To be more precise, the new patterns are found by solving an auxiliary optimization problem, which, as we will see, is a knapsack problem, using dual variable information from the linear problem.

Auxiliary knapsack problems can be resolved efficiently in $O(mW)$ time using dynamic programming or branch and bound method.

Algorithm 1

```
function COLUMN GENERATION ALGORITHM (arrivalJob)
    LPMP
    Start with initial columns A of (LPM). For instance, use the simple pattern to cut a roll into bW/wic rolls of width wi, A is a diagonal matrix.
    if ($n_1 + n_2 = N$) then
        Send arrivalJob on the cloud.
    else
        Send arrivalJob on the cloudlet.
```

2 Computational Model

Let's start now the description of our system *computational model* in which, as known, system's states exist as a collection of data structure, classes and variables that collectively characterize the system and are systematically updated as simulated time evolves.

Our model is been implemented using Java programming languages, which source code is fully available on GitHub[1], famous web-based hosting service for version control using `git`; we remind that LaTeX source code of this report is available too.[2]

In order to properly describe how our implementation works we need to understand how every mathematical elements, seen in previous section, are represented and managed.

2.1 The CSP instance

A *generic* CSP instance has been implemented and represented using a Java class called `CuttingStockInstance`, shown in Listing 1. In addition to several getter and setter methods, that class has two very important features:

- It has got a `final double` type field, called `maxItemLength`, which is used to hold the stock material width value that is, in other word, the raw roll width W.
- Through a list data structure, it holds a reference to all items required by customers through an `ArrayList<>` type field, called `items`.

  As you can see from Listing 1, that list contains m `CuttingStockItem` type instances which, obliviously, represent all specific items request by each customer. To be more precise, every `CuttingStockItem` type instances i, where $i = 1, ..., m$, contains two `double` type fields, called `length` and `amount`, which are used respectively to hold a reference to w_i and n_i, that is the length and the amount of rolls required by m-th customer.

Listing 1: `CuttingStockInstance` class implementation.

```java
public class CuttingStockInstance {

    private final double maxItemLength;
    private ArrayList<CuttingStockItem> items;

    public CuttingStockInstance(double maxItemLength) {
        this.maxItemLength = maxItemLength;
        this.items = new ArrayList<>();
    }

    public void addItems(double amount, double length) {
        this.items.add(new CuttingStockItem(amount, length))
            ;
    }
```

[1]Source code available on `https://github.com/AndreaG93/PMCSN-Project`

[2]See `https://github.com/AndreaG93/PMCSN-Project-Report`

```java

    double getMaxItemLength() {
        return maxItemLength;
    }

    ArrayList<CuttingStockItem> getItems() {
        return items;
    }
}
```

Listing 2: `solve()` method implementation.

```java
public void solve() throws Exception {

   buildMasterProblem();
   buildKnapsackSubProblem();

   timer.schedule( task, 15000 );
   long start = System.currentTimeMillis();

   executeColumnGenerationAlgorithm();

   long finish = System.currentTimeMillis();
   buildSolution();

   this.cuttingStockSolution.setTimeElapsed(finish - start);
}
```

Listing 3: `executeColumnGenerationAlgorithm()` method implementation.

```java
private void executeColumnGenerationAlgorithm() throws
    Exception {

    LinearProblemSolution masterProblemDualSolution;
    LinearProblemSolution knapsackSubProblemSolution;

    while (!this.timeOut) {

       this.masterProblemSolution = this.masterProblem.
           getSolution();
       masterProblemDualSolution = this.masterProblem.
           getDualSolution();

       this.cuttingStockSolution.addObjectiveFunctionValue(
           this.masterProblemSolution.
           getValueObjectiveFunction());

       this.knapsackSubProblem.
           changeObjectiveFunctionCoefficients(
           masterProblemDualSolution.getSolutions());
       knapsackSubProblemSolution = this.knapsackSubProblem.
           getSolution();

       if (1 - knapsackSubProblemSolution.
           getValueObjectiveFunction() < 0) {

            double[] newColumn = knapsackSubProblemSolution.getSolutions();

            this.masterProblem.addNewColumn(0.0, GRB.INFINITY, 1.0, VariableType.REAL, newColumn);
            this.cuttingStockSolution.increaseTotalNumberOfColumnsAdded();

        } else
            break;
        }

    this.timer.cancel();
}
```

Listing 4: `buildMasterProblem()` method implementation.

```java
private void buildMasterProblem() throws Exception {

    ArrayList<CuttingStockItem> cuttingStockItems = instance.getItems();
    double maxItemLength = instance.getMaxItemLength();
    int numberOfVariables = cuttingStockItems.size();

    double[] coefficientObjectiveFunction = new double[numberOfVariables];

    this.masterProblem.setVariables(numberOfVariables, 0, GRB.INFINITY, VariableType.REAL);

    Arrays.fill(coefficientObjectiveFunction, 1);

    this.masterProblem.setObjectiveFunction(coefficientObjectiveFunction, LinearProblemType.min);

    for (int index = 0; index < numberOfVariables; index++) {

        CuttingStockItem currentItem = cuttingStockItems.get(index);

        double[] constraintCoefficients = new double[numberOfVariables];

        constraintCoefficients[index] = ((int) (maxItemLength / currentItem.getLength()));
        this.masterProblem.addConstraint(constraintCoefficients, MathematicalSymbol.GREATER_EQUAL, currentItem.getAmount());
    }
}
```