Università degli Studi di Roma "Tor Vergata"

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

The Cutting-Stock Problem

Algoritmi e Modelli per l'Ottimizzazione Discreta

Studente:
Andrea Graziani
Matricola 0273395

Docente:
Andrea Pacifici

Anno Accademico 2018–2019

Contents

1 Analytical model

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**.

1.1 ILP Formulation

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 $w_i \leq W$.

As known, the first known ILP formulation of CSP was presented in 1939 by **Kantorovich**[1], which is reported below:

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

Where:

$$\begin{aligned} K &= \text{Index set of available rolls.} \\ y_k &= \begin{cases} 1, & \text{if roll } k \text{ is cut} \\ 0, & \text{otherwise} \end{cases} \\ x_k^i &= \text{Number of times item } i \text{ is cut on roll } k. \end{aligned} \tag{2}$$

Although is very easy to implement that formulation, its performances are very bad in practice due to a poor linear program relaxation of (P1).

Fortunately, an alternative and stronger ILP formulation exists and it is due to **Gilmore and Gomory**; to describe it, we had to know the following:

- 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.

The Gilmore and Gomory ILP formulation (P2) is reported below:

[1] `https://arxiv.org/ftp/arxiv/papers/1606/1606.01419.pdf`

$$
\begin{aligned}
(P2)\quad & min \quad \sum_{j=1}^{n} x_j \\
& s.t. \quad \sum_{j=1}^{n} a_{ij} x_j \geq n_i, \qquad i = 1, ..., m \quad (demand) \\
& \qquad x_j \in \mathbb{Z}_+, j = 1, ..., n
\end{aligned} \tag{3}
$$

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

$$
\begin{aligned}
& \sum_{i=1}^{m} w_i a_{ij} \leq W \\
& a_{ij} \in \mathbb{Z}_+
\end{aligned} \tag{4}
$$

Observe that each column of 3 (P2) represents a cutting patter.

Is very important to understand that *the number of existing cutting patterns* is **exponentially large**; it could be as many as $\frac{m!}{\bar{k}!(m-\bar{k})!}$ where $\bar{k}$ is the average number of items in each cutting patterns.[2]

Since, as we said, the number of possible patterns grows exponentially, it can easily run into the millions, 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**.

This is the reason according to which we have adopt and implemented another better approach to solve (P2): the **column generation method**.

[2]`http://opim.wharton.upenn.edu/~̃guignard/915_2015/slides/LR/CG/lecture_IP8.pdf`

1.2 CSP resolution algorithm

In order to properly describe the method used to solve (P2), we had to introduce the so called **Restricted Master Problem** (**RPM**), which is the LP relaxation of (P2). Its formulation is shown in 5.

Obliviously (RPM) solutions could be *fractional*; in that case is possible to *round up* the fractional solution to get a feasible solution to (P2); we will make further considerations about that later.

$$
\begin{aligned}
(RPM) \quad min \quad & \sum_{j \in P} x_j \\
s.t. \quad & \sum_{j \in P} a_{ij} x_j \geq n_i, \quad i = 1, ..., m \quad (demand) \\
& x_j \geq 0, j \in P
\end{aligned} \tag{5}
$$

The dual problem of (RPM) is:

$$
\begin{aligned}
(DRPM) \quad max \quad & \sum_{i=1}^{m} n_i \pi_i \\
s.t. \quad & \sum_{i=1}^{m} a_{ij} \pi_i \leq 1 \qquad j \in P, \\
& \pi_i \geq 0, i = 1, ..., m
\end{aligned} \tag{6}
$$

Resolution method is based on **column generation algorithm**, which main idea is to solve the CSP by starting with just a few patterns, **generating, when needed, additional columns**, that could improve current optimal solution of the linear relaxation (RPM).

To be more precise, the new columns (patterns) are found by solving an auxiliary optimization problem, called **slave problem** (SP), which, as you can see below, is a **knapsack problem**. As known, knapsack problems can be resolved efficiently in $O(mW)$ time using dynamic programming or branch and bound method. (SP) ILP formulation is:

$$
\begin{aligned}
(SP) \quad max \quad & \sum_{i=1}^{m} \pi_i y_i \\
s.t. \quad & \sum_{i=1}^{m} w_i y_i \leq W \qquad \text{(feasible cutting pattern)} \\
& y_i \in \mathbb{Z}_+, i = 1, ..., m
\end{aligned} \tag{7}
$$

where $y = (y_1, ..., y_m)$ represents a column $(a_{1j}, ..., a_{mj})^T$ (that is a cutting pattern).

In order to find a new column capable to improve current (RPM) optimal solution, we have to resolve (SP) in the dual variables, finding the column with the **most negative reduced cost**.

Given the optimal dual solution $\bar{\pi}$ of (RPM), the reduced cost of column $j \in \{1, ..., n\} \setminus P$ is:

$$\sum_{i=1}^{m} a_{ij}\bar{\pi} \tag{8}$$

A naive way of finding the new column:

$$min\{1 - \sum_{i=1}^{m} a_{ij}\bar{\pi} \quad | \quad j \in \{1, ..., n\} \setminus P\} \tag{9}$$

which is impractical because we are not able to list all cutting patterns in real applications. Therefore we can look for a column (cut pattern) such that:

$$\kappa = min 1 - \sum_{i=1}^{m} a_{ij} = 1 - max \sum_{i=1}^{m} \pi_i y_i \tag{10}$$

1. Heuristically initialize columns of (RPM) with a subset of pattern $P' \subset P$. In our computational model, we have used very simple patterns according to which a stock roll is cut into $\lfloor \frac{W}{w_i} \rfloor$ rolls of width w_i (where $i = 1, ..., m$).
2. Repeat until $\kappa \geq 0$:
   (a) Solve (RPM) and let $\bar{\pi}$ the optimal dual solution of (RPM).
   (b) Identify a column $p \in P$ that could reduce the objective function value solving (SP) in dual variables; if $\kappa \geq 0$, add the new column $y = (y_1, ..., y_m)$ to (RPM).

2 Computational Model

Let's start now the description of our resolver implementation in which, as known, we will turn all mathematical variables described above into a collection of data structure, classes and variables that, collectively, are able to resolve CSP problem.

Our resolver is been implemented using Java programming languages, which source code is fully available on GitHub[3], famous web-based hosting service for version control using `git`; we remind that LaTeX source code of this report is available too.[4]

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))
            ;
    }
```

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

[4]See `https://github.com/AndreaG93/AMOD-Report`

```

    double getMaxItemLength() {
        return maxItemLength;
    }

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

3 ILP implementation

Since an ILP Java implementation depends strictly on which ILP solver you decide to use, in order to ensure low coupling between our computational model and adopted ILP solver, a *generic* ILP is been represented using a Java `interface` exporting only necessary methods for CSP resolution; that interface, shown in 2, is called `LinearProblem`.

In that way, is possible for our computational model to work with several ILP solvers, as long as properly adapters, implementing all methods required by our interface, are provided.

Our computational model is compatible with *Gurobi* ILP solver through the concrete class `GurobiLinearProblem`, containing all method implementation of 2 interface using all.

Listing 2: Some methods exported by `LinearProblem` interface.

```
public abstract LinearProblemSolution getSolution() throws
    Exception;
public abstract LinearProblemSolution getDualSolution()
    throws Exception;
public abstract void setObjectiveFunctionType(
    LinearProblemType type) throws Exception;
...
public abstract void addConstraint(double[] coefficients,
    MathematicalSymbol symbol, double value) throws Exception
    ;
...
public abstract void addNewColumn(double
    newVariableLowerBound, double newVariableUpperBound,
    double value, VariableType
```

4 Column Generation Algorithm implementation

In our computational model,

The most important code is included, instead, into `CuttingStockProblem` class because it contains next-event simulation logic. Technically, `ComputationalModel` provides `perform` method implementation, which contains the algorithm[5] used to perform a simulation based on next-event approach, whose Java implementation is reported in Listing ??.

CuttingStockProblem type instance has several very important responsibilities:

- Allocation, initialization and management of all `LinearProblem` type instance necessary for CSP resolution, including the linear programming master problem and the auxiliary knapsack problem.

  The initialization phase of necessary `LinearProblem` type instance, performed according to what has been said previously from the analytical point of view, is carried out during `buildMasterProblem` and `buildKnapsackSubProblem` methods execution.

- Allocation and management of a **timer**, which, according to requirements, is been implemented in order to stop our resolver once a time limit, fixed to 15 seconds, is reached. If timer expires, an approximate solution of specified CSP problem will be displayed.

- CSP problem Resolution trhought Column Generation Algorithm

- Solution building.

Listing 3: `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 4: `executeColumnGenerationAlgorithm()` method implementation.

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

    LinearProblemSolution masterProblemDualSolution;
    LinearProblemSolution knapsackSubProblemSolution;

    while (!this.timeOut) {

```

[5]Lawrence M. Leemis, Stephen K. Park, *Discrete-Event Simulation: A First Course* (Pearson; 1 edition January 6, 2006), Algorithm 5.1.1, page 189

```java
        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 5: `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());
    }
}
```

4.1 Solutions display