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 **2**

1.1 ILP Formulation 2

1.2 CSP resolution algorithm 4

2 Computational Model **6**

2.1 The CSP instance 6

2.2 ILP solver 7

2.3 CSP solver 7

2.4 Solutions visualization 11

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:

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

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`

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

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

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

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 structures, classes and variables that, collectively, are able to resolve CSP problems.

Our application, compatible with *Gurobi* ILP solver[3], is been implemented using Java programming language while GUI is rendered through OpenJFX 12[4] API; however, we preferred to omit a detailed description of all classes involved into GUI rendering because they are just too technical and so not relevant for the purposes of this report.

Application's source code is available on GitHub[5], famous web-based hosting service for version control using `git`; we remind that LaTeX source code of this report is available too.[6]

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 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] `https://www.gurobi.com/`
[4] `https://openjfx.io/`
[5] Source code available on `https://github.com/AndreaG93/AMOD-Project`
[6] See `https://github.com/AndreaG93/AMOD-Report`

```
    }

    public double getMaxItemLength() {
        return maxItemLength;
    }

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

2.2 ILP solver

Obliviously, in order to resolve any CSP problem is necessary to design a way to represent a *generic* ILP, which implementation, generally, depends by which *ILP solver* you decide to use.

To ensure low coupling between our application and the adopted ILP solver, any ILP is been represented by a Java `interface` called `LinearProblem`: that interface, as you can see in Listing 2, exports *only* necessary methods for CSP resolution.

In that way, is possible, for our application, to work with any ILP solver, as long as properly adapters, implementing all methods required by our interface, are provided. Actually, only one adapter class, called `GurobiLinearProblem`, compatible with *Gurobi* ILP solver, containing all method implementations required by `LinearProblem` interface, is provided by default.

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

```
LinearProblemSolution getSolution() throws Exception;

LinearProblemSolution getDualSolution() throws Exception;

void setObjectiveFunctionType(LinearProblemType type) throws Exception;

void changeObjectiveFunctionCoefficients(double[] newCoefficients) throws
    Exception;

void addConstraint(double[] coefficients, MathematicalSymbol symbol, double
    value) throws Exception;
void setVariables(int totalNumberOfVariables, double lowerBound, double
    upperBound, VariableType varType) throws Exception;
void setObjectiveFunction(double[] coefficients, LinearProblemType type) throws
    Exception;
void addNewColumn(double newVariableLowerBound, double newVariableUpperBound,
    double value, VariableType varType, double[] columnCoefficient) throws
    Exception;
double[] getColumnCoefficient(int index) throws Exception;
```

2.3 CSP solver

A generic CSP problem is represented by `CuttingStockProblem` type instance, which has has several very important duties like:

- Allocation, initialization and management of all `LinearProblem` type instance necessary for CSP resolution.

  In fact, every `CuttingStockProblem` type instance holds a reference to two `LinearProblem` instances called `masterProblem` and `knapsackSubProblem`, respectively used to represent the the restricted master problem (RPM) and the slave problem (SP).

  In order to properly initialize the restricted master problem according to what has been said in previous section, is enough to invoke `buildMasterProblem` method, which is shown below. Another method, called `buildKnapsackSubProblem`, is used, instead, for (SP) initialization.

Listing 3: `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());
    }
}
```

- Allocation and management of a **timer** which, according to project requirements, is been implemented in order to stop our resolver once a time limit, fixed to **20 seconds**, is reached. If timer expires, an approximate solution will be displayed.

- CSP problem resolution using the algorithm described in 1.2. That algorithm is been implemented into `executeColumnGenerationAlgorithm` method, which is shown in Listing 4:

Listing 4: `executeColumnGenerationAlgorithm()` method implementation.

```java
private void executeColumnGenerationAlgorithm() throws Exception {
    Map<Integer, CuttingStockPattern> currentSolution;
    LinearProblemSolution masterProblemDualSolution;
    LinearProblemSolution knapsackSubProblemSolution;
    int[] currentIntegerSolution;
    double[] currentMasterProblemDualSolution;
    double currentWaste;
    while (!this.timeOut) {

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

        currentIntegerSolution = this.masterProblemSolution.
            getIntegerSolutions();
        currentMasterProblemDualSolution = masterProblemDualSolution.
            getRealSolutions();

        this.cuttingStockSolution.addObjectiveFunctionRealValue(this.
            masterProblemSolution.getRealValueOfObjectiveFunction());
        this.cuttingStockSolution.addObjectiveFunctionIntegerValue(this.
            masterProblemSolution.getIntegerValueOfObjectiveFunction());

        currentSolution = buildSolutionPatterns(currentIntegerSolution);
        currentWaste = computeWasteFromPattern(currentSolution);

        this.cuttingStockSolution.setCspSolutionPatterns(currentSolution);

        if (this.cuttingStockSolution.addWasteValueCheckingForMinimumValue(
            currentWaste)) {
            this.cuttingStockSolution.setCspSolutionPatternsMinimumWaste(
                currentSolution);
            this.cuttingStockSolution.
                setObjectiveFunctionValues_MinimumWaste(this.
                masterProblemSolution.getIntegerValueOfObjectiveFunction());
        }
        this.knapsackSubProblem.changeObjectiveFunctionCoefficients(
            currentMasterProblemDualSolution);
        knapsackSubProblemSolution = this.knapsackSubProblem.getSolution();

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

            double[] newColumn = knapsackSubProblemSolution.getRealSolutions
                ();
            this.masterProblem.addNewColumn(0.0, GRB.INFINITY, 1.0,
                VariableType.REAL, newColumn);
            this.cuttingStockSolution.increaseTotalNumberOfColumnsAdded();
        } else
            break;
     }
     this.timer.cancel();
}
```

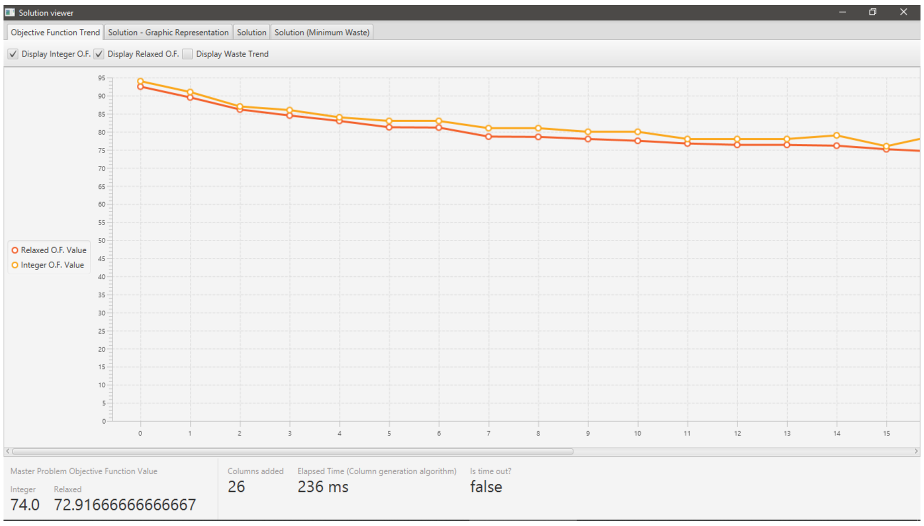
As you can see above, `executeColumnGenerationAlgorithm` method is used to collect some data which are strictly necessary for charts generation, concerning, for example, restricted master problem objective function value trend. All data collected, including final solutions, are hold by `CuttingStockSolution` type instance.

Anyway, `CuttingStockProblem` class includes other very useful method including:

`buildSolutionPatterns` which is used to build all patterns which belongs to solutions.

`computeWasteFromPattern` used, instead, to compute rolls waste according to specified patterns

2.4 Solutions visualization



Master Problem Objective Function Value

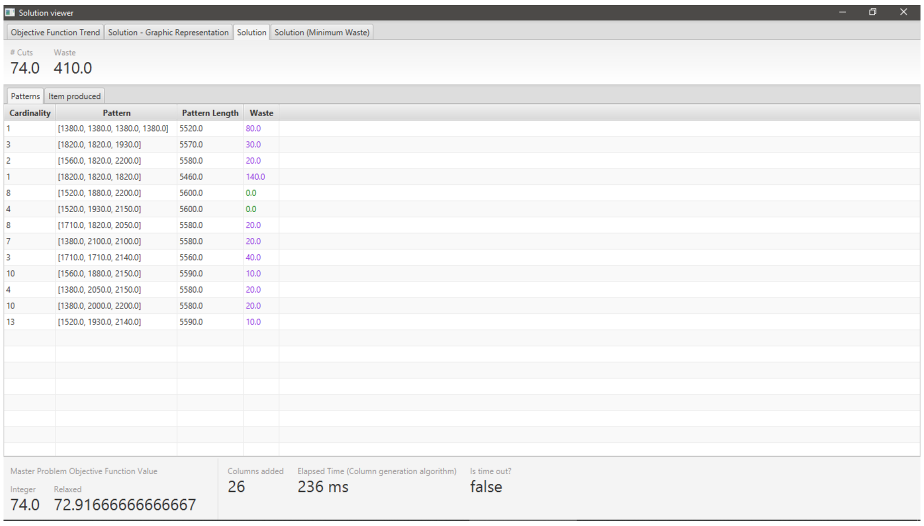
Integer: 74.0 Relaxed: 72.91666666666667

Columns added: 26

Elapsed Time (Column generation algorithm): 236 ms

Is time out?: false

Figure 1: Class diagram of our model (main classes and methods only are showed)



Cuts: 74.0 Waste: 410.0

Cardinality	Pattern	Pattern Length	Waste
1	[1380.0, 1380.0, 1380.0, 1380.0]	5520.0	80.0
3	[1820.0, 1820.0, 1930.0]	5570.0	30.0
2	[1560.0, 1820.0, 2200.0]	5580.0	20.0
1	[1820.0, 1820.0, 1820.0]	5460.0	140.0
8	[1520.0, 1880.0, 2200.0]	5600.0	0.0
4	[1520.0, 1930.0, 2150.0]	5600.0	0.0
8	[1710.0, 1820.0, 2050.0]	5580.0	20.0
7	[1380.0, 2100.0, 2100.0]	5580.0	20.0
3	[1710.0, 1710.0, 2140.0]	5560.0	40.0
10	[1560.0, 1880.0, 2150.0]	5590.0	10.0
4	[1380.0, 2050.0, 2150.0]	5580.0	20.0
10	[1380.0, 2000.0, 2200.0]	5580.0	20.0
13	[1520.0, 1930.0, 2140.0]	5590.0	10.0

Master Problem Objective Function Value

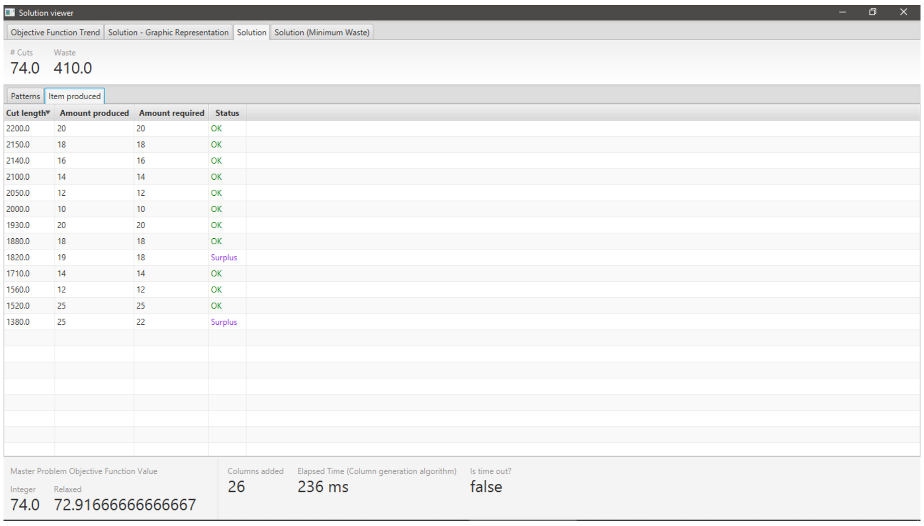
Integer: 74.0 Relaxed: 72.91666666666667

Columns added: 26

Elapsed Time (Column generation algorithm): 236 ms

Is time out?: false

Figure 2: Class diagram of our model (main classes and methods only are showed)


Solution viewer

Objective Function Trend | Solution - Graphic Representation | Solution | Solution (Minimum Waste)

# Cuts 74.0 Waste 410.0

Patterns | Item produced

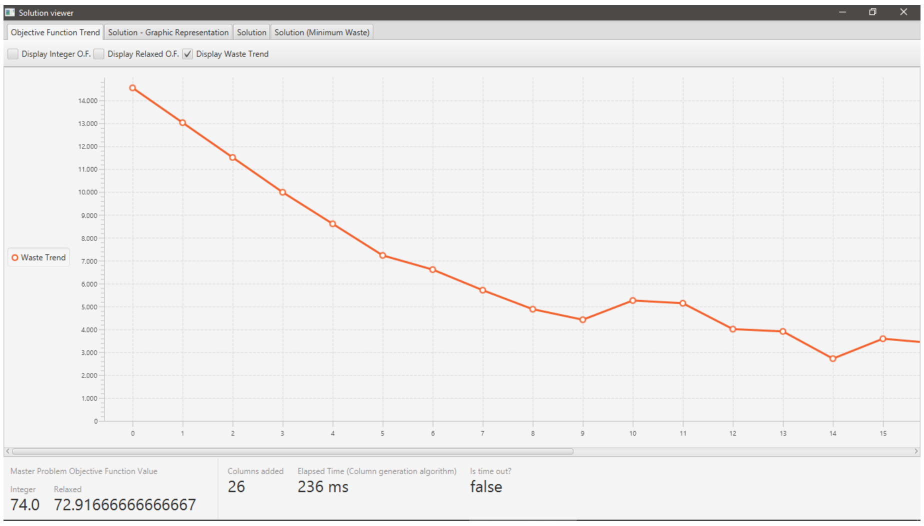
Cut length▾	Amount produced	Amount required	Status
2200.0	20	20	OK
2150.0	18	18	OK
2140.0	16	16	OK
2100.0	14	14	OK
2050.0	12	12	OK
2000.0	10	10	OK
1930.0	20	20	OK
1880.0	18	18	OK
1820.0	19	18	Surplus
1710.0	14	14	OK
1560.0	12	12	OK
1520.0	25	25	OK
1380.0	25	22	Surplus

Master Problem Objective Function Value

Integer 74.0 Relaxed 72.91666666666667

Columns added 26 Elapsed Time (Column generation algorithm) 236 ms Is time out? false

Figure 3: Class diagram of our model (main classes and methods only are showed)



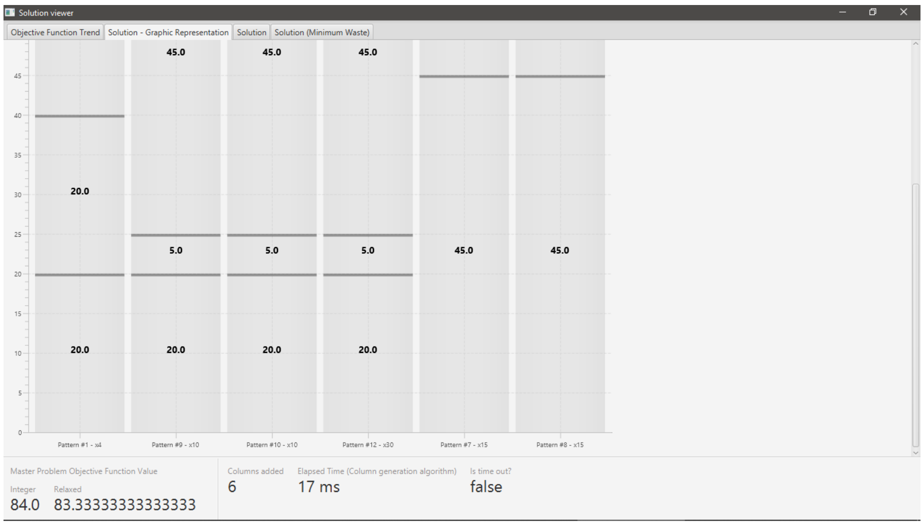
Display Integer O.F. Display Relaxed O.F. [✓ Display Waste Trend]

Master Problem Objective Function Value

Integer 74.0 Relaxed 72.91666666666667

Columns added 26 Elapsed Time (Column generation algorithm) 236 ms Is time out? false

Figure 4: Class diagram of our model (main classes and methods only are showed)



Master Problem Objective Function Value

Integer 84.0

Relaxed 83.33333333333333

Columns added 6

Elapsed Time (Column generation algorithm) 17 ms

Is time out? false

Figure 5: Class diagram of our model (main classes and methods only are showed)