

## Table of Contents

Problem definition and assumptions.....	1
Approach to solution.....	1
Constants.....	1
Variables.....	2
Goal function.....	2
Build, language and some implementation comments.....	3

## Problem definition and assumptions

Please refer to PDF document provided for the problem definition. On top of that, the following assumptions have been made:

- 1) A row has only two windows sit, one at each side of the row
- 2) Sitting “together” means beside each other. Breaking a group on the same row will cause passengers to be unsatisfied.
- 3) As a consequence unless a group occupies a full row, it can only get one window sit.

The problem is rather close to a multiple knapsack (with a twist). If assumption 2 is removed the problem becomes simpler.

## Approach to solution

The problem has been structured as an integer linear programming (ILP) problem and solved using an ILP open source solver. This type of approach can look over complicated at a glance but offers better flexibility than an ad-hoc heuristic, it takes advantage of existing libraries and is more open to changes in business logic (e.g. we can introduce rows of different size, we don't care if we exceed the plane capacity, we can consider more unsatisfied the users flying than the ones left behind in case of overbooking etc.).

The overall idea is to assign as many groups as possible according to the business goals and after that fill the flight with unsatisfied users. If the number of users is greater than the sits on the flight the problem does not change, the linear solver will try to maximize the score.

The formulation of the problem is as follows:

### Constants

$i$  --> row index

$j$  --> group index

$S_j$  --> size of group  $j$

$M_{ij}$  --> 1 if the size of group  $j$  is the same as row  $i$ , 0 otherwise

$R_i$  --> size of row  $i$

$W_j$  --> number of window preferences for group  $j$

$E_j$  --> number of window preferences for group  $j$  minus 1 if positive, zero otherwise

$V_j$  --> 1 if the group  $j$  has at least one window preference, 0 otherwise

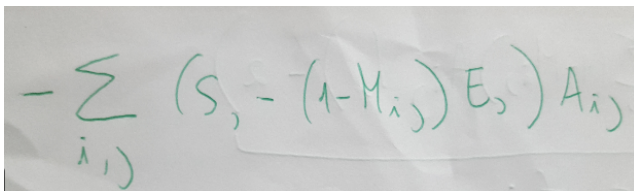
## Variables

$A_{ij}$  --> 1 if the group  $j$  is assigned to row  $i$ , 0 otherwise

## Goal function

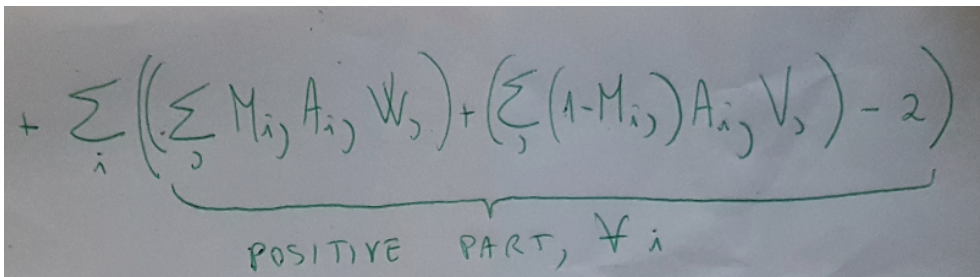
The problem is in form of minimum.

The first term of the goal function is self-explanatory.


$$- \sum_{i,j} (S_j - (1 - M_{i,j}) E_j) A_{i,j}$$

Given that it is not possible to allocate two window sits for a group that does not match the row size we incur automatically a penalty if that is required. In case of overbooking the algorithm is likely to leave such a group on the ground.

the second one accounts for two cases.


$$+ \sum_i \left( \underbrace{\left( \sum_j M_{i,j} A_{i,j} W_j \right) + \left( \sum_j (1 - M_{i,j}) A_{i,j} V_j \right) - 2}_{\text{POSITIVE PART, } \forall i} \right)$$

1. a group is taking the full row, so they can occupy two window sits without splitting
2. groups are smaller than the row, so one group can only take one window sit without splitting.
3. Note that for each row the above two situations are mutually exclusive. If a group takes the full row the other constraints will force all the other assignment variables to zero.

We are only interested in the case where the availability of window sits is not sufficient, so the positive part of the second term.

That introduces a non-linear constraint that we can turn into linear by the well known variable split trick- namely  $q_i^+$  and  $q_i^-$ , both integer. The  $\delta_i$  is the related a binary variable.

$$\forall_i \left( \sum_j (M_{ij} W_j + (1 - M_{ij}) V_j) A_{ij} \right) - 2 = q_i^+ - q_i^-$$

$$\forall_i \quad q_i^+ \geq 0 \quad q_i^- \geq 0$$

$$\forall_i \quad q_i^+ \leq \delta_i (R_i - 2)$$

$$\forall_i \quad q_i^- \leq (1 - \delta_i) \cdot 2$$

## Build, language and some implementation comments

1. The project is a simple maven project written in Java. The interface between the native linear solver implementation and java is not available in Maven, so the libraries have been included in the project. This is obviously not recommended, and would be better to import them in a local repository. *mvn install* (or eclipse maven integration) will build and run the tests. Obvious dependencies apply (Java 8 JVM).
2. It would be worth to look at sparse arrays as input, it would probably make the code less verbose and more memory efficient.
3. There are no performance requirements in the test but on a real size flight (180 sats) the solver runs in a fraction of a second (see test included).
4. The project has been built and tested on a linux 64 bit distribution, due to the presence of native code I would not recommend running it on a Windows environment.
5. I have experienced a couple of JVM crashes from the native libraries invoked by the solver. Not time to dig this out, however since I opened my first bug on the matlab linear solver over 20 years ago I would hammer this before putting it into any production environment.