Universitat Politecnica de Catalunya

Master in Innovation and Research in Informatics

Algorithmic Methods For Mathematical Models (AMMM)

Final Project

Goktug Cengiz & Roberto Ruiz Wilson

Barcelona, DECEMBER 2019

Table of Contents

1. Formal Statement

A well-known company seeks our advice in order to schedule the production of its best-selling car. When customers buy a car, they can equip it with a set of *options* O: sunroof, GPS, parking sensors, etc. We say that a *class* is a set of such options. There exists a fixed set of classes C and we have a Boolean matrix $ClassOption_{c,o}$ that indicates whether class $c \in C$ has option $o \in O$. Moreover, for each class c we know the exact number ($carsOfClass_c$) of cars of that class to be produced.

All cars go through the same production line and the only thing we need to decide is the order in which they will be put in such a line. For notational purposes, let us assume that we have a set of positions $P = \{1, 2, ..., nPositions\}$ in the line to fill. Obviously, $nPositions$ is equal to the total number of cars that have to be produced.

We would like to schedule the production of the car in the following variations of the above problem:

A. In this case, we assume that not all sequences are correct. For every option o we have two integers k_o and m_o that indicate that for every window of k_o consecutive cars, at most m_o can have option o.

B. Let us now assume that we also want to avoid changing too often the class in the line. We say that a position p has a change if $p-1$ and $p+1$ exist, the classes at positions $p-1$ and p are different from each other and the classes at positions p and $p+1$ are also different from each other. The goal is to minimize the number of positions where there is a change.

C. Finally, let us consider the problem in case A but assume now that the constraint stating that for every k_o consecutive cars at most m_o can have option o is soft, i.e., it can be violated. However, we want to minimize the number of times these constraints are violated. More specifically, we say that an option o has a violation at position p if there is a window of size exactly k_o starting at p which contains more than m_o cars with option o. The goal is to minimize the number of pairs (o, p) such that option o has a violation at position p.

2. Integer Linear Model

2.1 Sets and Variables

We start with the following data:

- O = set of options o.
- C = set of classes c.
- P = set of positions p.
- k_o = size of windows for option o
- m_o = maximum amount of vehicles with option o that can appear within a window of size k_o
- $ClassOption_{c,o}$ = Boolean indicating whether class c has option o.
- $carsOfClass_c$ = the exact number of cars of class c to be produced.

Do note that O, C and P are sets and not scalar values, therefore notation such as $|P|$ refers to the amount of elements in P and **not** to an absolute value.

For decision variables we will use two matrices:

- $po_{p,o}$ = Boolean indicating whether the car at position p has option o
- $pc_{p,c}$ = Boolean indicating whether the car at position p is of class c

2.2 Problem A

2.2.1 Objective Function

The problem A only requires us to find a feasible solution, no matter which, therefore the constant 1 is enough as an objective.

2.2.2 Constraints

a) **All cars have exactly one class**
We want the pc matrix to be kept in a consistent state, this means that for every row p there must exist exactly one entry with value 1, all other entries must be 0

$$\sum_{c=1}^{|C|} pc_{p,c} = 1 \quad \forall p \in [1, |P|]$$

b) **All cars have correct options**
The following constraint will enforce consistency between the pc matrix, the po matrix and the values given by $classOption$. It ensures that for every car of class c, the correct options are selected in the matrix po using the correct options from $classOption$.

$$(pc_{p,c} = 1) \implies (po_{p,o} = classOption_{c,o}) \quad \forall p \in [1, |P|], \forall c \in [1, |C|], \forall o \in [1, |O|]$$

c) **Correct amount of each class**
This will enforce that there's a correct amount of each class in the pc matrix with respect to what's specified in $carsOfClass$.

$$\sum_{p=1}^{|P|} pc_{p,c} = carsOfClass_c \quad \forall c \in [1, |C|]$$

d) **Window constraints**
This constraint will guarantee that for every option o, there doesn't exist a window of k_o consecutive cars that have more than m_o times that option. Every window starting at position p will end at position $p + k_o - 1$. For each window the options are summed and they must be less than or equal to the required amount in m_o.

$$\sum_{i=p}^{p+k_o-1} po_{i,o} \leq m_o \quad \forall o \in [1, |O|], \forall p \in [1, |P| - k_o + 1]$$

2.3 Problem B

2.3.1 Objective Function

We need to minimize the amount of changes from one class to another in the line of positions. Another way of looking at this is to maximize the amount of consecutive cars of the same class that will be created. We define a change as a difference in the class of the car at position p and the car at position $p + 1$ since this is equivalent to the formal definition of the problem.

To achieve this we only need to look at the matrix pc since that's the one that contains the information on class changes. A class will change if within a class column one of the positions is set to 1 and either the entry above it or below is 0. By subtracting each entry from the one below, we'll obtain 0 when it doesn't change and either 1 or -1 if there is a change. So we square this difference to make changes appear only as a positive 1.

One last thing to consider is that each change will be accounted for exactly twice. This because when going from position p to position $p+1$ there's a change from c to c', the change will be accounted for in both the column for class c and the column for class c'. If we were interested in knowing the exact count of changes we would need to divide the given objective function by 2. But given that the optimal arrangement of positions is exactly the same whether or not we divide the objective function by a constant, we have chosen to leave it as is.

Minimize:

$$\sum_{c=1}^{|C|} \sum_{p=1}^{|P|-1} (pc_{p,c} - pc_{p+1,c})^2$$

2.3.2 Constraints

The constraints for Problem B are exactly the same constraints as for Problem A, so we include them here for the sake of completeness but don't explain them again.

a) **All cars have exactly one class**

$$\sum_{c=1}^{|C|} pc_{p,c} = 1 \quad \forall p \in [1, |P|]$$

b) **All cars have correct options**

$$(pc_{p,c} = 1) \implies (po_{p,o} = classOption_{c,o}) \quad \forall p \in [1, |P|], \forall c \in [1, |C|], \forall o \in [1, |O|]$$

c) **Correct amount of each class**

$$\sum_{p=1}^{|P|} pc_{p,c} = carsOfClass_c \quad \forall c \in [1, |C|]$$

d) **Window constraints**

$$\sum_{i=p}^{p+k_o-1} po_{i,o} \leq m_o \quad \forall o \in [1, |O|], \forall p \in [1, |P| - k_o + 1]$$

2.4 Problem C

2.4.1 Objective Function

For Problem C, instead of minimizing the amount of incorrect windows we chose the inverse problem of maximizing the valid windows. Both are the exact same problem, the only difference is in the direction of the inequality, and we take advantage of the fact that the maximization problem uses a "less than or equal" instead of a strict inequality.

For this objective function we're adding up Boolean values, when the sum of the elements within a windows is less than or equal to the maximum allowed allowed for that window, we add a 1, otherwise we add a zero.

Maximize:

$$\sum_{o=1}^{|O|} \sum_{p=1}^{|P|-k_o+1} \left(\left(\sum_{i=p}^{p+k_o-1} po_{i,o} \right) <= m_o \right)$$

2.4.2 Constraints

Again, use the same constraints as in problems A and B, but this time we eliminate constraint c.

a) **All cars have exactly one class**

$$\sum_{c=1}^{|C|} pc_{p,c} = 1 \quad \forall p \in [1, |P|]$$

b) **All cars have correct options**

$$(pc_{p,c} = 1) \implies (po_{p,o} = classOption_{c,o}) \quad \forall p \in [1, |P|], \forall c \in [1, |C|], \forall o \in [1, |O|]$$

c) **Correct amount of each class**

$$\sum_{p=1}^{|P|} pc_{p,c} = carsOfClass_c \quad \forall c \in [1, |C|]$$

3. Heuristics

For each algorithm:

- define MAX_VALUE = $carsOfClass_c$;

3.1 GREEDY

Algorithm 1 GREEDY Problem

```
1: function ALGORITHM(array{Class} C, array{int} carsOfClass, int nPositions )
2:     # P is the solution. In each position of P there is the class assigned to that position.
3:     array{Class} P ← new Array(nPositions)
4:     array copyCarsOfClass ← copy(carsOfClass)
5:     for (i ← 0; i < nPositions; ++i) do
6:         # q has for each class the q value
7:         Queue{Class,int} q ← evaluate_q(C, copyCarsOfClass, i, P)
8:         bestClass ← q.first()
9:         P.add(bestClass.c)
10:    end for
11:    return P
12: end function
```

The tactic will be: if the class of the actual position is equal to the previous one, for sure we will not have changes, so this will be a good solution for sure. Otherwise no matter the class it is, it will not be such good. In that case the q value will be number of cars that the class have, this means we prefer a class with more cars because it is more possible that in the next positions it does not produce changes, because the algorithm tends to place the same classes in adjacent positions. It is important to mention that the structure of carsOfClass will be updating during the execution while a car of class c is assigned.

Algorithm 2 Evaluate q

```
1: function EVALUATE_Q(array{Class} C, array{int} carsOfClass, int nPosition, array{Class}
   P))
2:     queue q{Class,int} ← new Queue(nPositions)
3:     for (c in Class) do
4:         # We always compare with the back position, so in the first position any class can be
   part of the solution
5:         if ( thenposition == 0)
6:             q.add(c, MAX_VALUE)
7:         else
8:             if ( thenc == P[position - 1])
9:                 q.add(c, MAX_VALUE)
10:            else
11:                q.add(c, carsOfClass[c])
12:            end if
13:        end if
14:        −− carsOfClass[c]
15:        # Remove the class if all the cars available are already in the solution
16:        if ( thencarsOfClass[c] == 0)
17:            delete(carsOfClass[c])
18:        end if
19:    end for
20: end function
```

3.2 GRASP

Algorithm 3 GRASP Problem

```
1: function GRASP(array{Class} C, array{int} carsOfClass, int nPositions)
2:     array{Class} P ← new Array(nPositions)
3:     while S is not a Solution do
4:         for (i ← 0; i < nPositions; ++i) do
5:             Queue{Class,int} q ← evaluate_q(C, copyCarsOfClass, i, P)
6:             # q has for each class the q value
7:             qmin ← q.last()
8:             qmax ← q.first()
9:             RCLmax ← [classValue in q || classValue.qvalue ≥ qmax - alpha * (qmax - qmin)]
10:            # We are chosing the pair of {Class,int} (class and its q value) that has a q value
    greater or equals than qmax - alpha * (qmax - qmin)
11:            i ← rand(size(RCLmax))
12:            P.add(RCL[i].c)
13:        end for
14:    end while
15: end function
```

3.3 Greedy Function $q(\sim)$

The q value of a class C to be assigned to a position i is:

$$q(c,i) = \begin{cases} +\infty, & i == 0 \text{ or } c == p[i-1] \\ carsOfCars[c], & i \neq 0 \text{ and } c \neq p[i-1] \end{cases}$$

3.4 Restricted Candidate List (RCL)

RCL = CV.qvalue ≥ qmax - (qmax - qmin) ∀ cv ∈ q where q is an array of a pair class, qvalue previously computed.

Position	1	2	3	4	5
Class	1	1	2	**2**	**6**

As you can see in the table above, we run the algorithms proposed in the previous item to decide the next two cars in the sequence. They are selected as 2 and 6 respectively.

In position 4 it will goes 4 because with a car of class 2 it will not produce any change. And in position 5 as all the cars in class 2 are already assigned, the class must be different, the chosen will be 6 because our algorithm choose the class with the biggest number of cars, so that it is more difficult to produce changes because in the following positions will be cars of class 6.