User Scheduling in 4G

Reda Belhaj-Soullami, Amine Rabhi

Notations : Throughout all this report, we will denote by P the array (with length N) containing all the $(P_n)_{n\in N}$ matrices (with shape (K,M)) which entries are the $p_{k,m,n}$. We use the same notation for R.

Q1. $x_{k,m,n}$ is a binary variable, its value is 1 if channel n is used by user k with data rate $r_{k,m,n}$, and otherwise it's zero. In more mathematical terms we define

$$x_{k,m,n} = \mathbb{1}(r_{k,n} = r_{k,m,n})$$

First, for $n \in \{1,\dots,n\}$, we define P_n to be the matrix with shape (K,M) with entries $p_{k,m,n}$. Similarly we define R_n to be the matrix with shape (K,M) with entries $r_{k,m,n}$, and X_n the matrix with entries $x_{k,m,n}$.
Let's now formulate our problem and constraints.

— We have to maximize the following sum :

$$\sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{m=1}^{M} r_{k,m,n}x_{k,m,n} = \operatorname{tr}\left(\sum_{n=1}^{N} R_n^\top X_n\right)$$

— Let's look at the constraints.

— The first one is that the total transmit power budget should be less than p. Let us notice than when we choose $x_{k,m,n} = 1$, it is in our best interest to choose $p_{k,n} = p_{k,m,n}$ since we should use the least amount of power possible.
The constraint yields

$$\sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{m=1}^{M} p_{k,m,n}x_{k,m,n} \le p \iff \operatorname{tr}\left(\sum_{n=1}^{N} P_n^\top X_n\right) \le p$$

— The second one is to have exactly one user served per channel. We define the matrix J with shape (K,M) with all entries equal to one. For all $n \in \{1,\dots,n\}$ we should have

$$\sum_{n=1}^{N} x_{k,m,n} = 1$$

which one can rewrite as

$$\forall n \in \{1,\dots,n\}, \operatorname{tr}(JX_n) = 1$$

Let's denote by $\mathscr{M}_{K,M}(\{0,1\})$ the set of (K,M) matrices with binary entries.
We are left with the following problem :

$$\max_{X(1),\dots,X(n)\in\mathscr{M}_{K,M}(\{0,1\})} \operatorname{tr}\left(\sum_{n=1}^{N} R_n^\top X_n\right)$$

s.t

$$\operatorname{tr}\left(\sum_{n=1}^{N} P_n^\top X(n)\right) \le p$$

and

$$\forall n \in \{1,\dots,n\}, \operatorname{tr}(JX_n) = 1.$$

Q2. First of all we check if the instance has obviously no solutions. It happens when

$$\sum_{n=1}^{N} \min_{k\in\{1,\dots,K\}} p_{k,1,n} > p.$$

In this case, the first constraint can never be satisfied.

Then, for every (k_0, m_0, n_0), if

$$\sum_{n \neq n_0} \min_{k \in \{1,\dots,K\}} p_{k,1,n} + p_{k_0,m_0,n_0} > p$$

then (k_0, m_0, n_0) can never be chosen.

Implementation and complexity

For the first part, we simply compute for each $n \in \{1,..,N\}$ the minumum. This step has a complexity of $O(KN)$.

For the second part, for each (k_0, n_0, m_0) we compute the minimum $\min_{k \in \{1,\dots,K\}} p_{k,1,n}$. The total complexity is then $O(NKM)$.

Q3. For this question, for each $i \in \{1, \dots, N\}$ we first sort the P_i array. Then, we review the R values in the order of the sorted P array and we keep a current value r_{current} that represents the lowest $r_{k,m,i}$ for now . If a value encountered is less than r_{current} it means that we have an inversion. Therefore, it needs to be removed. Otherwise, we update the value of r_{current} and continue.

The complexity is dominated by the cost of sorting the P array which is $O(NKM\log(KM))$.

Here's the pseudo-code for the Lemma 1 algorithm.

Algorithm 1 Applying Lemma 1

Inputs : p, K, N, M, P, R

```
for i = 1 to N do
  p_sorted ← sort(P_i)
  r_current ← p_sorted[0]
  for l = 1 to KM do
    if R_i[p_sorted[l]] ≤ r_current then
      Remove instance (k, m, n)
    else
      r_current = R_i[p_sorted[l]]
    end if
  end for
end for
```

Proof It is obvious that Algorithm 1 is ending. Let's focus more on its correctness. We will use the fact that for each sorted array P_i, the sub-array $P_i[0, l]$ contains no inversion if are only considered the entries that were not removed. Indeed, let's suppose that this propriety is true for a certain l, then if $p_{\text{sorted}_{l+1}}$ corresponds to a rate that is lower than r_{current}, then it needs to be removed as there is an inversion between l and the index which rate is r_{current}. Otherwise, there is no need to remove l and therefore it has the highest rate in the sub-array $P_i[0 : l+1]$. In both cases, $P_i[0 : l+1]$ has no inversions. This conclude the proof of the correctness.

Complexity It is obvious that for each P_i, the **for** loop has a complexity of $O(KMlog(KM)$ that is the result of the sorting algorithms. Therefore, the algorithm has a complexity of $O(NKMlog(KM)$.

Q4. For this question, the goal is to make the function $p_{k,m,n} \mapsto r_{k,m,n}$ concave for each $n \in \{1,...,N\}$. The main idea behind our algorithm is to take the convex hull of the set $\{(p_{k,m,n}, r_{k,m,n}), k \in \{1,...,K\}, m \in \{1,...,M\}\}$ for each $n \in \{1,...,N\}$. In order to do so, we "draw" a line between the first $p_{k,m,n}$ and the last one and then we look for the first point whose rate is above the line. We keep doing this process until we reach the last point. The image below may help understand the strategy :

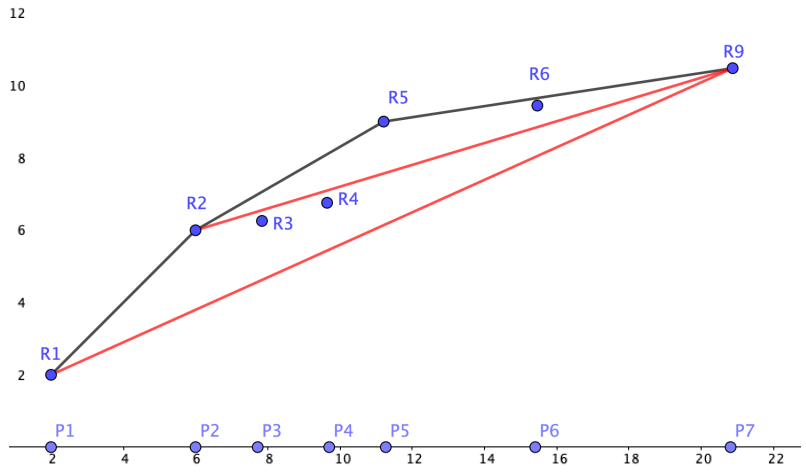

FIGURE 1 – How the Lemma 2 algorithm works : In this case, we will remove points 3,4 and 6.

Let's now provide the pseudo-code for the Lemma 2 algorithm.

Note : For the p_{sorted} array, we use a data structure that allow us to find original indexes (i.e. indexes in P_i) in time $O(1)$.

Algorithm 2 Applying Lemma 2

Inputs : p, K, N, M, P, R

for $i = 1$ to N **do**
  $p_{\text{sorted}} \leftarrow \text{sort}(P_i)$
  $q = 1$
  $t = 2$
  while $t \leq KM$ **do**
    $q' \leftarrow$ index such that $P_i[q'] = p_{\text{sorted}}[q]$
    $f \leftarrow$ index such that $P_i[f] = p_{\text{sorted}}[KM]$
    rate $= \frac{p_{\text{sorted}}[q] - p_{\text{sorted}}[KM-1]}{R_i[q'] - R_i[f]}$
    $t' \leftarrow$ index such that $P_i[t'] = p_{\text{sorted}}[t]$
    if $R_i[t'] >$ rate $\times (p_{\text{sorted}}[t] - p_{\text{sorted}}[q]) + R_i[q']$ **then**
      $q = t$
    else
      Remove instance t'
    end if
  end while
end for

Here are some figures to understand the 2 cases in the **while** loop.

Proof The algorithm starts with sorting P_i for each $i \in \{0, ..., N\}$. The application of the first lemma guaranties that each P_i is a strictly increasing list. The goal of lemma 2 is to make it a strictly concave one also. Let us show that each sub-array $P_i[0, q]$, is strictly concave, which will both prove correctness and termination. Let's consider q such as $P_i[0 : q] \wedge P_i[KM-1]$ is strictly concave.

If $P_i[q+1]$ has a rate strictly greater than the line that join the rate of $P_i[q]$ and the rate of $P_i[KM-1]$, then$P_i[q+1]$ isn't removed. If the sub-array $P_i[0 : q+1] \wedge P_i[KM-1]$ strictly concave, it would mean that $P_i[0 : q] \wedge P_i[KM-1]$ isn't either because $P_i[KM-1]$ has a greater rate than $P_i[q+1]$. Therefore, the property is extended to $P_i[0 : q+1]$.

If $P_i[q+1]$ has a rate smaller that the line, then the algorithm moves to $q+2$. Either way, it is obvious that the algorithm ends as t (the index that is studied) is always increasing. Moreover, as t takes the value $KM-1$ at the end, this ensures that the resulting array is strictly concave. This conclude both the proves of he correctness and the termination.

Complexity

Let's consider $i \in \{0, ..., N\}$. We first begin by sorting P_i, which complexity is $O(KM\log(KM))$. Then we apply an algorithm that removes the unwanted points according to lemma 2. We denote by $T(q)$ the complexity of applying this algorithm to an array of

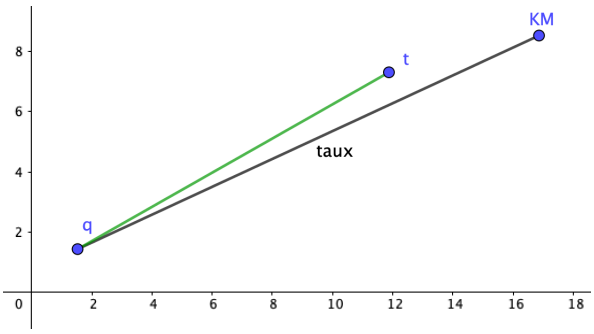

FIGURE 2 – Favorable situation

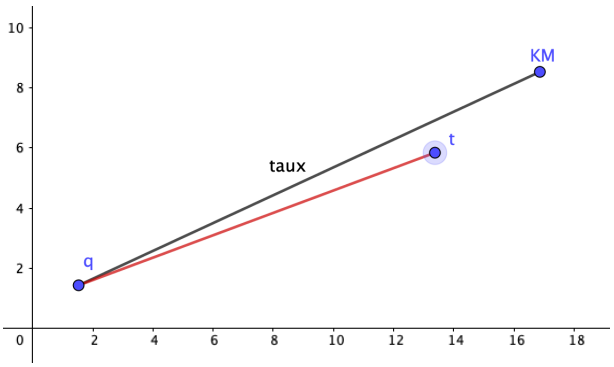

FIGURE 3 – Unfavorable situation

length q. T verifies the propriety below :

$$T(q) = T(q-r) + O(r)$$

where r is the index where q is changed in the algorithm. We deduce from that that $T(KM) = O(KM)$ Therefore, the whole algorithm has a complexity of $O(NKMlog(KM)$

Q5. This is how the test files are modified.

	File 1	File 2	File 3	File 4	File 5
After preprocessing	0 %	No solution	0 %	0 %	0 %
After Lemma 1	58 %	×	45 %	97 %	87 %
After Lemma 2	66%	×	66 %	99 %	96 %

TABLE 1 – Percentage of instances removed

Q6. Our idea is to use the instance (l,n) with the best $e_{l,n}$ possible if the channel n is not yet saturated.
Here's the basic pseudo code for our greedy algorithm. In our implementation (Python), we have to sort first the P_i arrays in order to compute E. We also sort E at the beginning to have an easy access to its maximum, and keep an array of the status of the channels (saturated or not). We also compute the powers dynamically in order to the do the constraint test in time $O(1)$

Algorithm 3 Greedy algorithm

Inputs : p, K, N, M, P, R

```
E ←[e_{l,n} for l, n in L, N]
while All channels are not saturated and E is not empty yet do
    Denote by (k, m, n) the index of the greatest value in E, and delete this value from E
    if channel n is not saturated then
        if using (k, m) in channel N violates the total power constraint then
            Channel n is saturated
        else
            Use (k, m) in channel n
        end if
    end if
end while
```

Proof
Since this is a greedy algorithm, we do not need to give a correctness proof. We only focus on proving the algorithm finishes. At every iteration of the **while** loop the cardinal of E decreases by 1. So the loop takes at most $|E|$ iterations and, thus, finishes.

Complexity As we have $|E| < KMN$ and we have to sort the P_i arrays, the cost is dominated by $O(NKM\log(KM)$.

Q7. First, Table 2 shows how the greedy algorithm compares to the optimal one, in terms of score and used power. File 2 does not have any feasible solution and File 3 is too large to allow to compute exact linear optimization.
Table 3 shows the optimality the greedy algorithm manages to achieve for each test file.

	Greedy(File 1)	Optimal(File 1)	Greedy(File 3)	Optimal(File 3)	Greedy(File 5)	Optimal(File 5)
Score	365	365	312	366,7	993	1073
Used power	78	78	63	100	863	1000
Total power available	100	100	100	100	1000	1000
Computation time	0.14 ms	12 ms	0.17 ms ,	11 ms	5.8 ms	46 s

TABLE 2 – Comparison between the greedy algorithm and the optimal algorithm in terms of score, used power, and computation time

	File 1	File 3	File 5
Score	100 %	85 %	92 %

TABLE 3 – Optimality score for the greedy algorithm

Q8. Let's use a dynamic programming algorithm. We denote by $T(q,i)$ the maximum data rate we can obtain from channels $1,\dots,i$ with the constraint of having a total power less than q. The DP paradigm relies on the fact that $T(q,i)$ can be computed with the values $T(q',i-1)$. Our algorithm has two steps : first, we compute the value of the solution (namely the T matrix). Then, we recover the actual solution X. For this second step, we are going to store additionnal information while computing T, namely the indices where the maxima occurs. From the last index and the last data rate, we can deduce what power was used to get to the last index. We repeat this operation until we have recovered all the indices.

Algorithm 4 Dynamic programming

Inputs : p, K, N, M, P, R

Initialize T an array with shape $(p+1, N+1)$ with values 0.
Initialize a Ind array of the indexes where we will find the maxima.
for $i = 2$ to $N+1$ **do**
  for $q = 2$ to $p+1$ **do**
    Compute max $T[q - P_{i-1}[k,m], i-1] + R_{i-1}[k,m]$ among all k, m with $q - P_{i-1}[k,m] \geq 0$.
    Store the value in $T[q,i]$ and its index in Ind$[q,i]$
  end for
end for
solution $\leftarrow$ [Ind$[p,N]$]
$i \leftarrow 1$
power $\leftarrow p$
while $i < N$ **do**
  puissance $\leftarrow$ puissance - P_{N-i}[solution$[i-1]$]
  $i \leftarrow i+1$
  Append to solution the element Ind[puissance -1]$[N-i-1]$
end while
Convert solution into a matrix

Proof The termination of the algorithm is obvious as it does not contain any while loop. Let's focus on the correctness. The correctness of the algorithm relies on the dynamic programming paradigm and the following equation :

$$T(q,i) = \max_{(k,m)\in I_{q,i}} T(q - P_i(k,m), i-1) + R_i(k,m)$$

where $I_{q,i} = \{(k,m) \in [\![1,K]\!] \times [\![1,M]\!] | q - P_i(k,m) \geq 0\}$

Using this equation in the algorithm, we compute the values $T(q,i)$. Now we prove that from the two arrays (T and Ind) we can recover the solution. The solution relies also on the equation above. In fact, having $T[p+1, N+1]$ and the index k, m) that reached the maximum in the equation above, we can find both the index of P_N as well as the data rate that was optimal with a power lower than$p - P_N[k,m]$. Therefore, we have the same problem than before with a reduced size which ensures that the final solution will exactly be the list of (k,m) that appear in the final solution.

Complexity The main issue in DP is to fill the tables. All the other parts of the algorithm have a complexity of $O(N)$ as their aim is just to fill a couple of arrays which length are exactly N. For filling the main tables, each entry needs to find the maximum of KM

others at most. Therefore, the complexity is $O(pNKM)$.

Finally we conlcude with a general comparison of our three algorithms (Table 4).

	Greedy(1)	Optimal(1)	Dynamic(1)	Greedy(3)	Optimal(3)	Dynamic(3)	Greedy(5)	Optimal(File 5)	Dynamic(5)
Score	365	365	365	312	366,7	324	993	1073	1006
Used power	78	78	78	63	100	81	863	1000	924
Total Power	100	100	100	100	100	100	1000	1000	1000
Time	0.14 ms	12 ms	5.9 ms	0.17 ms	11 ms	5.7 ms	5.8 ms	46 s	1.4 s

TABLE 4 – Comparison between the three algorithms