Project INF421

User Scheduling In 5G

Subject proposed by : Marceau COUPECHOUX

February 2020

Ziyad Benomar,
Redallah Mouhoub

This subject as well as test files are available on this page:

https://marceaucoupechoux.wp.imt.fr/enseignement/english-inf421-pi/

Formulating the problem as an integer linear program

Let the binary variables $x_{k,m,n} \in \{0,1\}, k \in \mathcal{K}, n \in \mathcal{N}, m \in \mathcal{M}$ where $\mathcal{M} = \{1, ..., M\}$. We set $x_{k,m,n} = 1$ if and only if $p_{k,n} = p_{k,m,n}$; that is when the user k is served a data rate $r_{k,n}$ over channel n with power $p_{k,m,n}$.

For a given $n \in \mathcal{N}$ and under the constraint that every channel shall be used to serve exactly a single user, we can see that $p_{k_n,n} \neq 0$ for a unique $k_n \in \mathcal{K}$.

$$\begin{aligned}\text{Now, } (\forall n \in \mathcal{N},\ \exists! k_n \in \mathcal{K},\ \text{such that } p_{k_n,n} \neq 0) &\implies \forall n \in \mathcal{N},\ \exists! k_n \in \mathcal{K},\ \exists! m_n \in \mathcal{M}, x_{k_n,m_n,n} = 1 \\ &\implies \forall n \in \mathcal{N},\ \sum_{m \in \mathcal{M}} x_{k_n,m,n} = 1 \\ &\implies \forall n \in \mathcal{N}, \sum_{k \in \mathcal{K}} \sum_{m \in \mathcal{M}} x_{k,m,n} = 1\end{aligned}$$

Also, in this case, the total data rate served is given by

$$\sum_{n \in \mathcal{N}} \sum_{k \in \mathcal{K}} \sum_{m \in \mathcal{M}} r_{k,m,n} x_{k,m,n}$$

and the power budget used to deliver that data rate is

$$\sum_{n \in \mathcal{N}} \sum_{k \in \mathcal{K}} \sum_{m \in \mathcal{M}} p_{k,m,n} x_{k,m,n}.$$

Hence, our Integral Linear Problem is formulated as follows

$$\begin{aligned}\text{Maximise} \quad & \sum_{n \in \mathcal{N}} \sum_{k \in \mathcal{K}} \sum_{m \in \mathcal{M}} r_{k,m,n} x_{k,m,n} \\ \text{subject to} \quad & \sum_{n \in \mathcal{N}} \sum_{k \in \mathcal{K}} \sum_{m \in \mathcal{M}} p_{k,m,n} x_{k,m,n} \leq p \\ & \sum_{k \in \mathcal{K}} \sum_{m \in \mathcal{M}} x_{m,k,n} = 1 \quad \forall n \in \mathcal{N} \\ & x_{k,m,n} \in \{0,1\} \quad \forall k \in \mathcal{K}, n \in \mathcal{N}, m \in \mathcal{M}\end{aligned}$$

Preprocessing

A quick preprocessing

Constraints are not verified in our problem if and only if

$$\sum_{n \in \mathcal{N}} p_{k_n,m_n,n} = \sum_{n \in \mathcal{N}} \sum_{k \in \mathcal{K}} \sum_{m \in \mathcal{M}} p_{k,m,n} x_{k,m,n} > p$$

since we control the values $x_{k,m,n}$ but not $p_{k,m,n}$. We deduce that having at least one solution requires having a condition on the lower bound of the total power budget used :

$$(*) \qquad \sum_{n \in \mathcal{N}} \min_{\substack{k \in \mathcal{K} \\ m \in \mathcal{M}}} \{p_{k,m,n}\} \leq p$$

With a similar argument, a given triplet (k_0, m_0, n_0) doesn't prevent solutions to be feasible if

$$p_{k_0,m_0,n_0} \leq p - \sum_{n \neq n_0} \min_{\substack{k \in \mathcal{K} \\ m \in \mathcal{M}}} \{p_{k,m,n}\}$$

This quick preprocessing step consists of sorting the sequence $(p_{k,m,n})_{m\in\mathcal{M},k\in\mathcal{K}}$ in ascending order for each n and verifying if $(*)$ is satisfied. The problem has no solution if not. Then, eliminate all triplets $\{(k,m,n)\mid p_{k,m,n} \geq p_{k_0,m_0,n}\}$ in case (k_0,m_0,n) prevents any solution to be feasible, i.e in case it doesn't verify the previous inequality.

The sorting of $(p_{k,m,n})_{m\in\mathcal{M},k\in\mathcal{K}}$ for each n is not necessary at this step, but we chose to do it and save the triplets $(k,m,n)_{m\in\mathcal{M},k\in\mathcal{K}}$ in a linked list t_n following this order. Hence, each time we need to iterate over them we will use t_n. We used linked lists rather than other structures because they are optimal for the operations we will need.

Removing IP-dominated triplets

An IP-dominated triplet is simply a triplet that prevents the utility function from being increasing. In the following pseudo-code, the markAsRemoved function doesn't remove the triplet from the list; it only marks it as removed, allowing us to increment the index without changing the length of the list.

```
Algorithm 1: Remove IP-dominated terms
Input  : An instance containing all the values $(p_{k,m,n}, r_{k,m,n})$ and $p$
Output: Triplets that are not IP-dominated, sorted in increasing order of $p_{k,m,n}$ for each $n$
for $n = 0$ to $N-1$ do
    $t_n \leftarrow$ List of triplets $(k,m,n)_{k,m}$ in ascending order of $p_{k,m,n}$ ;
    $i, j \leftarrow 0, 1$ ;
    while $j < len(t_n)$ do
        if $r_{t_n[i]} \geq r_{t_n[j]}$ then
            markAsRemoved($t_n[i]$) ;
            $j \leftarrow j+1$ ;
        else
            if $p_{t_n[i]} = p_{t_n[j]}$ then
                markAsRemoved($t_n[i]$) ;
                $i \leftarrow i+1$ ;
                $j \leftarrow j+1$ ;
            end if
        end if
    end while
end for
return $\{(t_n)$ with non-marked triplets only, $n \in \mathcal{N}\}$;
```

For each n, sorting triplets $(k,m,n)_{k,m}$ can be done with a complexity $\mathcal{O}(KM\log(KM))$ using a merge-sort algorithm, and assuming that we can access the values $(p_{k,m,n})_{k,m}$ with a constant complexity.
Assume that markAsRemoved() has a constant complexity, which is easy to achieve. At each step of the while loop, we do a constant number of elemental operations, and we increment j. We get out of the loop when $j = len(t_n)$, so we need at most $len(t_n)$ iterations, the complexity of the loop is then $\mathcal{O}(len(t_n)) = \mathcal{O}(KM)$.

The time complexity of the function removeIPdominated() is $\mathcal{O}(NKM\log(KM))$

We wrote the pseudo code with indexes i, j to make it clearer, but in fact t_n is a linked list, we only need to put pointers $d1, d2$ on each its first and second element, markAsRemove($t_n[i]$) becomes $d1.remove()$, and the incrementing $d1 = d1.next$.

Removing LP-dominated triplets

An LP-dominated triplet is simply a triplet that prevents the utility function from being concave.

Algorithm 2: Remove LP-dominated terms

```
Input  : An instance containing all the values $(p_{k,m,n}, r_{k,m,n})$ and $p$
Output: Triplets that are not LP-dominated
$(t_n)_{n\in\mathcal{N}} \leftarrow$ Output of removeIPdominated() ;
for $n = 0$ to $N - 1$ do
    $i_1, i_2, i_3 \leftarrow 0, 1, 2$ ;
    while $i_3 < len(t_n)$ do
        if $\frac{r_{t_n[i_3]} - r_{t_n[i_2]}}{p_{t_n[i_3]} - p_{t_n[i_2]}} \geq \frac{r_{t_n[i_2]} - r_{t_n[i_1]}}{p_{t_n[i_2]} - p_{t_n[i_1]}}$ then
            markAsRemoved($t_n[i2]$) ;
            if $i_1 = 0$ then
                $i_2 \leftarrow i_2 + 1$ ;
                $i_3 \leftarrow i_3 + 1$ ;
            else
                $i_2 \leftarrow i_2 - 1$ ;
                $i_1 \leftarrow i_1 - 1$ ;
            end if
        else
            $i_1 \leftarrow i_1 + 1$ ;
            $i_2 \leftarrow i_2 + 1$ ;
            $i_3 \leftarrow i_3 + 1$ ;
        end if
    end while
end for
return $\{(t_n)$ with non-marked triplets only, $n \in \mathcal{N}\}$;
```

Removing IP-dominated terms takes a time complexity $\mathcal{O}(NKM\log(KM))$, and it returns the triplets already sorted in ascending order of $p_{k,m,n}$
At each step of the while loop we do a constant number of elemental operations, and either we explore a new triplet, or we remove an already explored one. Hence, each triplet is explored for the first time once, and removed at most once, which shows that the total complexity of the while loop is $\mathcal{O}(KM\log(KM))$, and since we have a loop for each n :

The time complexity of the function removeLPdominated() is $\mathcal{O}(NKM\log(KM))$

Results

The following table shows how the size of the instances are reduced after calling each of the previous pre-processing methods.

	"test1.txt"	"test2.txt"	"test3.txt"	"test4.txt"	test5.txt"
initially	24	24	24	614400	2400
quickPreprocessing()	24	0	24	614400	1954
removeIPdominated()	10	0	13	14687	300
removeLPdominated()	8	0	9	4974	179

The following figure shows graphically how triplets $(k, m, 9)_{k,m}$ in the test "test5.txt" are reduced after the pre-processing.

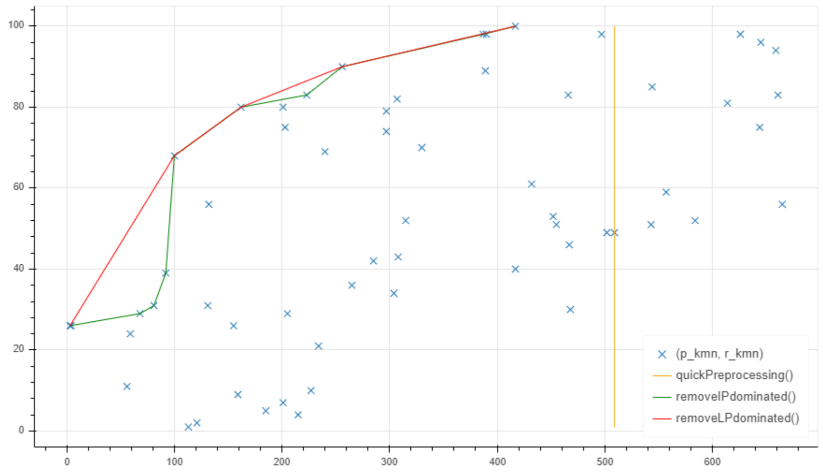

Figure 1: reduction of triplets (k,m,n) for n=9 in test5.txt. Points $(p_{k,m,n}, r_{k,m,n})$ on the right of the vertical orange line are the ones removed after the initial pre-processing. the green curve connects non-IP-dominated points, the red curve connects non-LP-dominated ones.

Note : before presenting the algorithms answering the remaining questions, we will explain the main structures that we will use in our program.
An instance will be modeled by an object of the class **Instance**. Such object has the following attributes:

- Integers N, K, M, P giving the dimensions of our problem and the total budget power;
- two matrices p, r containing the values $p_{k,m,n}, r_{k,m,n}$;
- a table t of linked lists of triplets : for each n, $t_n = t[n]$ is a linked list containing triplets (k, m, n) sorted in ascending order of $p_{k,m,n}$.

The solution is given by the values $x_{k,m,n}$, but since for each $n, \exists! k_n \in \mathcal{K}$, $\exists! m_n \in \mathcal{M}, x_{k_n,m_n,n} = 1$, we can simply model the solution by an array $x[0, ..., N-1]$ of couples (k, m) such that for each n, $x[n] = (k_n, m_n)$. When solving the LP-problem in Q6, we will need one convex combination between two values $x_{k,m,m}$ and $x_{k',m',n}$ having the same n, that's why will need to add some attributes.
A solution will be modeled by an object of the class **Solution**, having the following attributes

- an instance *ins* of the problem for which we will find a solution
- an array x of couples (k, m)
- integers k, m, n and a float number λ

For a given solution *sol* having the attributes x, k, m, n, λ, we have $\forall n' \neq n : x_{k',m',n'} = 1$ when $(k', m') = x[n']$, and $x_{k,m,n} = \lambda$, $x_{k_n,m_n,n} = 1 - \lambda$ when $(k_n, m_n) = x[n]$.

In the pseudo-code we will respect the problem notations, but in our code, $x_{k,m,m} = 1$ is replaced by $x[n] = (k, m)$. An object of the class Instance occupies a space $\mathcal{O}(KMN)$ and its construction requires a $\mathcal{O}(NKM \log(KM))$ time complexity (because we construct t). And an object of the class Solution occupies a space $\mathcal{O}(N)$ if we don't count the instance.

All the solving functions we expose in the following assume that we have already pre-processed the instance. In particular, we won't ask ourselves when treating an instance if the problem has a solution or not.

Linear Program and Greedy Algorithm

A greedy algorithm

After the pre-processing, the points $(p_{l,n}, r_{l,n})$ are sorted in ascending order of $p_{l,n}$ and also the utility function is increasing, therefore it is always better to choose $(p_{l,n}, r_{l,n})$ over $(p_{l-1,n}, r_{l-1,n})$ for $l > 1$ since $r_{l,n} > r_{l-1,n}$. The greater $e_{l,n}$, the better the transition from $(l-1, n)$ to (l, n), hence the idea for our algorithm.

Algorithm 3: Greedy Algorithm for LP problem

```
Input  : Preprocessed instance containing all the values $(p_{l,n}, r_{l,n})$ in ascending order and $p$ for each n,
         and the budget power p
Output: Solution to the LP problem
Sort all the values $(e_{l,n})_{l,n}$ in decreasing order ;
Let $x$ be a new three dimensional matrix of shape $K \times M \times N$ s.t. $x_{k,m,n} = 0, \forall k, m, n$ ;
for $n = 0$ to $N-1$ do
    $x_{1,n} \leftarrow 1$ ;
end for
while $\sum_{l,n} x_{l,n}p_{l,n} < p$ and all the couples $(l, n)$ have not been visited yet do
    find the greatest $e_{l_0,n_0}$ between two last visited elements among all channels ;
    if $\sum_{l \neq l_0, n} x_{l,n}p_{l,n} + p_{l_0,n_0} \leq p$ then
        $x_{l_0-1,n_0} \leftarrow 0$ ;
        $x_{l_0,n_0} \leftarrow 1$ ;
    else
        $x_{l_0-1,n_0}, x_{l_0,n_0} \leftarrow$ fractional convex combination s.t.
          $\sum_{l \neq l_0, n} x_{l,n}p_{l,n} + x_{l_0-1,n_0}p_{l_0-1,n_0} + x_{l_0,n_0}p_{l_0,n_0} = p$ ;
    end if
end while
return $(x_{l,n})_{l \in \mathcal{L}, n \in \mathcal{N}}$;
```

Sorting all the values $(e_{l,n})_{l,n}$ in decreasing order can be done with a complexity $\mathcal{O}(NKM \log(NKM))$. The for loop has a $\mathcal{O}(N)$ complexity, and the while loop $\mathcal{O}(NKM)$ because at each step we make a constant number of elemental operations and a new couple (l, n) is visited until there are no couples left or the budget constraint is saturated. In space, we need to store the instance, an object of the class Solution, and a sorted list of $(e_{l,n})_{l,n}$. Hence

The time complexity of the function solveLP() is $\mathcal{O}(NKM \log(NKM))$
and its space complexity is $\mathcal{O}(NKM)$

For the implementation of our algorithms, we also computed a method solveSubLP() that solves the sub-problem where values of $x_{k,m,n}$ are fixed for $n < n_0$. This method will be used later in the Branch-And-Bound algorithm.

Results

Results for the greedy algorithm

$solveLP()$	"test1.txt"	"test3.txt"	"test4.txt"	test5.txt"
Budget power	100	100	16000	1000
Used power	78	100	16000	1000
Data rate	365	372.15384	9870.322	1637
Run time (ms)	12	5	13	326
IP-solution	true	false	false	true

Note that for the tests 1 and 5, we directly get a solution for the IP-problem.

Algorithms for solving the ILP

A DP algorithm

Here, we assign a channel to a user after computing the maximum data rate we could obtain from a sub-problem. Let $R(n,p)$ be the maximum data rate provided by channels $n, ..., N-1$ under a power budget less than p. When we choose a triplet (k,m,n), our budget decreases by $p_{k,m,n}$ and our data rate increases by $r_{k,m,n}$. The formula that expresses the link between a problem and its sub-problems is the following:

$$\begin{aligned} &\forall\, 0 \le n < N,\ \forall\, 0 \le p < P \\ &\qquad R(n,p) = max\{R(n+1, p - p_{k,m,n}) + r_{k,m,n} \mid (k,m) \in \mathcal{K} \times \mathcal{M}\} \\ &\qquad R(N,p) = R(n,0) = 0 \end{aligned}$$

The optimal data rate is then given by $R(0,P)$.

Algorithm 4: Dynamic Programming Algorithm for IP problem

```
Input  : An instance containing all the values $(p_{k,m,n}, r_{k,m,n})$ and $p_{max}$
Output: Solution to the IP problem
Initialize $R$ a null matrix of shape $(N+1) \times (P+1)$
for $n = N-1$ to $0$ do
    for $p = 0$ to $P$ do
        $R(n,p) \leftarrow max\{R(n+1, p - p_{k,m,n}) + r_{k,m,n} \mid (k,m) \in \mathcal{K} \times \mathcal{M}\}$ ;
        $(k,m) \leftarrow argmax\{R(n+1, p - p_{k,m,n}) + r_{k,m,n} \mid (k,m) \in \mathcal{K} \times \mathcal{M}\}$ ;
        $x_{k,m,n} \leftarrow 1$ ;
    end for
end for
return $(x_{k,m,n})_{k \in \mathcal{K}, m \in \mathcal{M}, n \in \mathcal{N}}$ ;
```

Finding the maximum data rate for each n, p can be done with a time complexity $\mathcal{O}(KM)$. Hence

The time complexity of solveDP() is $\mathcal{O}(PKMN)$

However, the space complexity depends on whether we choose to store the solutions while computing the optimal data rate or by computing the solutions from the stored optima of all sub-problems.

In fact, the first approach requires to fill an additional table stroring the chosen triplets in the solution ($\mathcal{O}(N)$ space) while the second one doesn't require any further storing.

An alternative DP algorithm

For this alternative DP algorithm we assume that we have an LP solver that provides us with U an upper bound for the data rate we could get. If we denote the minimum power budget allocated to channels $n, ..., N-1$ providing data rate r by $P(n,r)$, then the optimal allocated power budget is $P(0,U)$ and the formula becomes:

$$\begin{aligned} &\forall\, 0 \le n < N,\ \forall\, 0 \le r < U \\ &\qquad P(n,r) = min\{P(n+1, r - r_{k,m,n}) + p_{k,m,n} \mid (k,m) \in \mathcal{K} \times \mathcal{M}\} \\ &\qquad P(n,0) = 0;\ P(N-1,r) = p_{k,m,n} \text{ if } r_{k,m,n} = r, \text{ and } +\infty \text{ otherwise} \end{aligned}$$

We will construct an optimal solution using the minimum possible power using the functions. The first one computes the table $(P(n,r))_{n,r}$, and the second solves the IP problem using this table.

Algorithm 5: computeTableP()

```
Input  : An instance containing all the values $(p_{k,m,n}, r_{k,m,n})$ and $U$ obtained by some LP solver
Output: A table P containing the values $(P(n,r))_{n\geq N, r\geq U}$
Let $P(n,r)$ be a null matrix of shape N $\times(U+1)$
Initialize $P(n,0)$ and $P(N-1,r)$ as in the formula
for  $n = N-2$ to 0 do
    for $r = 0$ to $U$ do
        $P(n,r) \leftarrow min\{P(n+1, r - r_{k,m,n}) + p_{k,m,n} \mid (k,m) \in \mathcal{K} \times \mathcal{M}\}$ ;
    end for
end for
return  $P$;
```

Algorithm 6: Alternative Dynamic Programming Algorithm for IP problem

```
Input  : An instance containing all the values $(p_{k,m,n}, r_{k,m,n})$ and $U$ obtained by some LP solver
Output: Optimal solution to the IP problem using the less power possible
Compute $sol$ a solution to the LP-problem
U $\leftarrow$ $(int) dataRate(sol)$   //An upper bound for the data rate
P = computeTableP(U);
// decrement U untill we get a feasible data rate
while $P(0,U) = +\infty$ do
    $U \leftarrow U-1$
end while
Initialize a three dimensional matrix $x$ of shape $N \times K \times M$
for  $n = N$ to 0 do
    for $r = 0$ to $U$ do
        $(k,m) \leftarrow argmin\{P(n+1, r - r_{k,m,n}) + p_{k,m,n} \mid (k,m) \in \mathcal{K} \times \mathcal{M}\}$ ;
        $x_{k,m,n} \leftarrow 1$ ;
    end for
end for
return  $(x_{k,m,n})_{k\in\mathcal{K}, m\in\mathcal{M}, n\in\mathcal{N}}$ ;
```

In the function computeTableP(), finding the minimal power for each n, r can be done with a time complexity $\mathcal{O}(KM)$, thus the time complexity needed for constructing P is $\mathcal{O}(KMNU)$.

In the second function, we first compute a solution for the LP problem in order to have an upper bound U. If we do this using our greedy algorithm, it will require a $\mathcal{O}(KMN\log(KMN))$. Calling computeTableP() requires $\mathcal{O}(KMNU)$ time. The while loop has a $\wr(U)$ complexity, we are sure of its termination because the instance has been pre-processed (look at the note page 4). And for each n, r we need $\mathcal{O}(KM)$ time to find the good couple (k,m), that yields a $\mathcal{O}(KMNU)$ time complexity.

For the space requirement, solveDPAlternative() needs $\mathcal{O}(NU)$ to store the table P, this is why it can be beneficial to use the function solveLP() to have a reduced upper bound U. We could have thought of other values of U such as the sum on n of the maximal data rate over $(r_{k,m,n})_{k,m}$, but the upper bound would have been too large and so will be the space requirement.
However, solving the LP-problem requires $\mathcal{O}(NKM\log(NKM))$ for storing the sorted values $(e_{l,n})_{l,n}$.

Deducing an upper bound from the solution of the LP-problem or looking for another method to calculate it depends on the parameters K, M, N and on the values $r_{k,m,n}$.

The time complexity of solveDPAlternative() is $\mathcal{O}(KMN(U + \log(KMN))$
The space requirement of solveDPAlternative() is $\mathcal{O}(KMN(U + \log(KMN))$

A branch and bound algorithm

We start by defining an intermediate recursive function that solves the sub-IP-problem starting from n_0. We pass over all possible values of $(x_{k,m,n})_{k,m,n \geq n_0}$ susceptible of giving a solution with a better data rate than dataRate(x^*), and whenever we find a solution to the ILP problem, we compare it to x^*. Hence, at the the end of the recursive calls, x^* is the optimal solution.

Algorithm 7: solveRecBB() : an intermediate function for Branch and Bound algorithm

```
Input : An instance containing all the values $(p_{k,m,n}, r_{k,m,n})$ ;
        Two three dimensional matrices $x^*$ and $x$ of shape $K$x$M$x$N$ ;
        // $x^*$ is the best solution found so far, and $x$ is a test matrix
        $n_0 \in \mathcal{N}, p_{left}$
Output: No output, instead it modifies $x$ and solves the sub-problem where $(x_{k,m,n})$ for $n < n_0$ are
        fixed, and the remaining power budget is $p_{left}$
// STOP CONDITIONS ——————————————————————————
if $n_0 = N$ then
    if $dataRate(x) > dataRate(x^*)$ then
        $(x^*_{k,m,n})_{k \in \mathcal{K}, m \in \mathcal{M}, n \in \mathcal{N}} \leftarrow (x_{k,m,n})_{k \in \mathcal{K}, m \in \mathcal{M}, n \in \mathcal{N}}$ ;
        return;
    end if
else if the relaxed sub-problem has no solution then
    return;
$x \leftarrow$ the solution for the relaxed sub-problem // solveSubProblem() ;
else if dataRate(x) < (dataRate($x^*$) + 1) then
    return;
else if all $x_{k,m,n}$ are integers then
    $(x^*_{k,m,n})_{k \in \mathcal{K}, m \in \mathcal{M}, n \in \mathcal{N}} \leftarrow (x_{k,m,n})_{k \in \mathcal{K}, m \in \mathcal{M}, n \in \mathcal{N}}$ ;
    return;
// RECURSION ——————————————————————————
else
    for $k = 0$ to $K - 1$ do
        for $m = 1$ to $M - 1$ do
            $x_{k,m,n_0} \leftarrow 1$ ;
            $x_{k',m',n_0} \leftarrow 0, \forall (k', m') \neq (k, m)$ ;
            solveRecBB(instance, $x^*$, $x$, $n_0 + 1$, $p_{left} - p_{k,,m,n}$) ;
        end for
    end for
    return;
```

Note. The solution to the LP-sub-problem provides an upper bound for the IP-sub-problem, if we have $dataRate(x) < dataRate(x^*) + 1$, then any IP-solution will a data rate no larger than $dataRate(x^*)$ because they are both integers.

We move then to solving the main problem. We start with two matrices x and x^*, and with $n_0 = 0$. Instead of initializing x^* as a null matrix, we initialize it as the solution for the LP-problem and delete the last iteration of solveLP() that makes the solution non-integer, therefore we have a non-optimal but good initial upper bound. If the solution given by solveLP() happens to be be a solution for the IP problem, then the stop condition on dataRate(x^*) will prevent from exploring the tree.

Algorithm 8: Branch and Bound Algorithm for ILP problem

```
Input  : An instance containing all the values $(p_{k,m,n}, r_{k,m,n})$ and $p$
Output: Solution to the ILP problem
Sort all the triplets $(k, m, n)_{k,m,n}$ in descending order of $e_{l,n}$ in a global variable
// To avoid sorting them each time we need to solve a sub-LP-problem
Let $x^*$ and $x$ two three dimensional matrices of same shape $K \times M \times N$ ;
$x^* \leftarrow solveLP$(instance) ;
if there are two values $x^*_{k_1,m_1,n}$ and $x^*_{k_2,m_2,n}$ that are fractional then
    Set $x^*_{k_i,m_i,n} = 0$ and $x^*_{k_j,m_j,n} = 1$, where $p_{k_j,m_j,n} \leq p_{k_j,m_j,n}$ and $i, j \in \{1, 2\}$ ;
end if
solveRecBB(instance,$x^*, x, 0, p$) ;
return $x^*$;
```

Note. When solving the sub-LP-problem and during the recursion phase, we don't duplicate $x_{k,m,n}$, we work on the same matrix given in argument, and there is no output, hence during all the recursive calls of solveRecBB(), we never create new local variables. Unfortunately we are obliged to give n_0 in argument, so a copy of its value at each depth level is automatically generated. The space complexity is thus proportional to the time complexity.

When calling solveRecBB() for $n = n_0$, the size of the problem is $N - n_0$. We do a constant number of comparisons and affectations, we solve the sub-LP- problem in a $\mathcal{O}(KM(N - n_0))$ because we already have sorted $(e_{l,n})$, and we call KM times solveRecBB() for a smaller sub-Problem of size $N - n_0 - 1$. Hence, the complexity $T(n)$ for a problem of size n satisfies the recursion

$$\begin{aligned} T(n) &= KM \times n + KM \times T(n-1) \\ &= KM(n + T(n-1)) \\ T(n) &\geq KM \times T(n-1) \end{aligned}$$

And since solving a problem o size 0 is done in constant time, we can deduce that $T(n) \geq (KM)^N$, and therefore the term $KM \times n$ is negligible in the recursion equation, and we have $T(n) = \mathcal{O}((KM)^N)$.
In solveRecBB(), sorting $(e_{l,n})_{l,n}$ has a $\mathcal{O}(KMN \log(KMN))$ complexity, and calling solveRecBB() for $n = 0$ has $\mathcal{O}((KM)^N)$ complexity. Hence

The time complexity of the function solveBB() is $\mathcal{O}((KM)^N)$

Note that this complexity is theoretical. When testing our function on the test files, we obtain very satisfying results !

Comparing results

Since we have multiple stop conditions and a good initial upper bound, we cut an important number of nodes. Here is table of results for solveBB() showing also the number of explored nodes for each test :

solveBB()	"test1.txt"	"test3.txt"	"test4.txt"	test5.txt"
Budget power	100	100	16000	1000
Used power	78	68	16000	1000
Data rate	365	350	9870	1637
Run time (ms)	22.0	1.2	949.0	1.0
Explored nodes	1	10	4536	1

And here is the table of results for the dynamic programming algorithm

solveDPalt()	"test1.txt"	"test3.txt"	"test4.txt"	test5.txt"
Budget power	100	100	16000	1000
Used power	78	68	15999	1000
Data rate	365	350	9870	1637
Run time (ms)	10.0	0.0	4173.0	20.0

The two functions find exactly the same optimal solution for the IP-problem. The difference of the run time for the tests 1,2, and 5 is not significant because the instances have a small size. But for the test4, we can see that the Branch-And-Bound algorithm is almost 10 times faster than the dynamic programming algorithm. Still the BB algorithm occupies $\mathcal{O}((KM)^N)$ space when the DP only occupies $\mathcal{O}(Np)$ space. For very large problems, we may not be able to to use the BB algorithm because of of the space constraint.

Stochastic Online Scheduling

Before attempting to solve the problem, let's try first to understand it well. If we consider a given channel n, at each time $t = k$, a new user will arrive and the n will see M new points $(p_{k,m,n}, r_{k,m,n})_m$, n chooses to pick one of them or wait for the next users. For this, we need first to identify the best point among the current ones, and decide if it is a good point, in which case we pick it, or not.
Our pseudo-code is based on this simple idea :

Algorithm 9: Heuristic algorithm for the Online problem

```
Input  : $K$, $M$, $N$, $p$, $p^{max}$, $r^{max}$
Output: Solution to the Online problem
$k \leftarrow 0$ ;
Let $\mathcal{N}_k$ be the set of unused channels at time $k$ ;
while $\mathcal{N}_k \neq \emptyset$ and $k < K$ and $p > 0$ do
    Generate random values $(p_{k,m,n})_{n,m}$ in $\{1, ..., p^{max}\}$ and $(r_{k,m,n})_{n,m}$ in $\{1, ..., r^{max}\}$
    for $n = 0$ to $N_k$ do
        Choose the "best" triplet $(k, m^*, n)$ among $\{(k, m, n)\}_{0 \leq m < M}$ ;
        if $(k, m^*, n)$ is "good enough" then
            $x_{k,m^*,n} \leftarrow 1$ ;
            remove $n$ from $\mathcal{N}_k$ ;
            $p \leftarrow p - p_{k,m^*,n}$ ;
        end if
    end for
    $k \leftarrow k + 1$ ;
end while
return $x^*$;
```

If we want to make this pseudo-code efficient, we need to define precisely a criteria for comparing two points (p_1, r_1) and (p_2, r_2) in order to choose the best one, and we need to know when a point can be said good enough. Let (p_1, r_1) and (p_2, r_2) be two points such that $p_1 < p_2$ and let $e = \frac{r_2 - r_1}{p_2 - p_1}$ be slope between them. It is logical that if e is too small then (p_1, r_1) is better, and if it's too large then (p_2, r_2) is better. Therefore, we need a threshold value e^* from which (p_2, r_2) is better. The natural value we thought of, since we know the distribution of p and r is the average expected slope :

$$e^* =< e >=< \frac{r}{p} >= \frac{1}{p^{max} r^{max}} \sum_{p=1}^{p^{max}} \sum_{r=1}^{r^{max}} \frac{r}{p}$$

$$\begin{aligned}
(p_2, r_2)\text{is better than } (p_1, r_1) &\iff \frac{r_2 - r_1}{p_2 - p_1} \geq e^* && (1)\\
&\iff r_2 - r_1 \geq (p_2 - p_1)e^* && (2)\\
&\iff r_2 - e^* p_2 \geq r_1 - e^* p_1 && (3)\\
&\iff \text{quality}(p_2, r_2) \geq \text{quality}(p_1, r_1) && (4)
\end{aligned}$$

While quality() is the function $(p, r) \longrightarrow r - e^* p$.
Now that we have a tool for deciding the best point, we still need a criteria for saying that a point is good enough.A simple way to make this decision is to estimate the percentage of points that are no better than the current one and compare it to an exigency parameter α :

Algorithm 10: goodEnough(p*, r^*)

```
Input  : point $(p^*, r^*)$
Output: boolean value : true if $(p^*, r^*)$ is good enough, and false otherwise
if $\frac{1}{p^{max} r^{max}} |\{(p, r),\ quality(p^*, r^*) \geq quality(p, r)\}| > \alpha$ then
    return true;
else
    return false;
end if
```

OPTIMIZATIONS.

The function quality() does not take the budget power p in consideration, if we want it to do so, we can think that at time k, the allowed budget power for each channel is $\frac{p}{|\mathcal{N}_k|}$, so the quality of the point $(p_{k,m,n}, r_{k,m,n})$ should be reduced the more $p_{k,m,n}$ is larger than $\frac{p}{|\mathcal{N}_k|}$.

$$\text{If } p_{k,m,n} > p : \text{ quality}(p_{k,m,n}, r_{k,m,n}) = (r_{k,m,n} - e^* p_{k,m,n}) \frac{p}{p_{k,m,n} |\mathcal{N}_k|}$$

But we should also set a limit budget for each channel in order to let enough budget for the others, we choose to take limitAllowedBudget $= p - (|\mathcal{N}_k| - 1)(1 + \frac{p^{max}}{M})$
Hence, the quality is $(r_{k,m,n} - e^* p_{k,m,n})$ when $p_{k,m,n} \leq p/|\mathcal{N}_k|$, it is multiplied by $\frac{p}{p_{k,m,n}|\mathcal{N}_k|}$ if $p_{k,m,n} > p/|\mathcal{N}_k|$, and it is $-\infty$ if p is larger than the limit allowed budget.
We can also optimize the function goodEnough() by allowing the exigency parameter α to change over time : we can imagine that the exigency at the beginning is higher then the exigency at the end. At the time k, we will consider the exigency

$$\alpha_k = \alpha_0 + \frac{k}{K - 2}(\alpha_{K-1} - \alpha_0)$$

Appliying the stochastic online scheduling

Since we make online choice, there is a risk of having a non complete solution, in which all the channels didn't serve a user. For such a solution, the data rate will be considered null when computing the competitive ratio.
When testing our algorithm, we tried different values of α_0 and α_{K-1}, we noticed that the ratio is remarkably high for some of them, so we ran an auxiliary algorithm computing the ratio over 100 experiences for α_0, α_{K-1} taking all the values of [0,1] with a step of 0.01, and then we did the same with a bigger precision on smaller intervals. The best results were provided by $\alpha_0 = 0.945$ and $\alpha_{K-2} = 0.855$. They are the values used by default in our program.
Over 500000 experiences, we get the following results :

- Competitive ratio : 0.7905965
- 0.00273 % of the solutions are not complete
- Run time : 5 minutes 2 seconds

Here is a graph showing the heuristic distribution of the competitive ratio

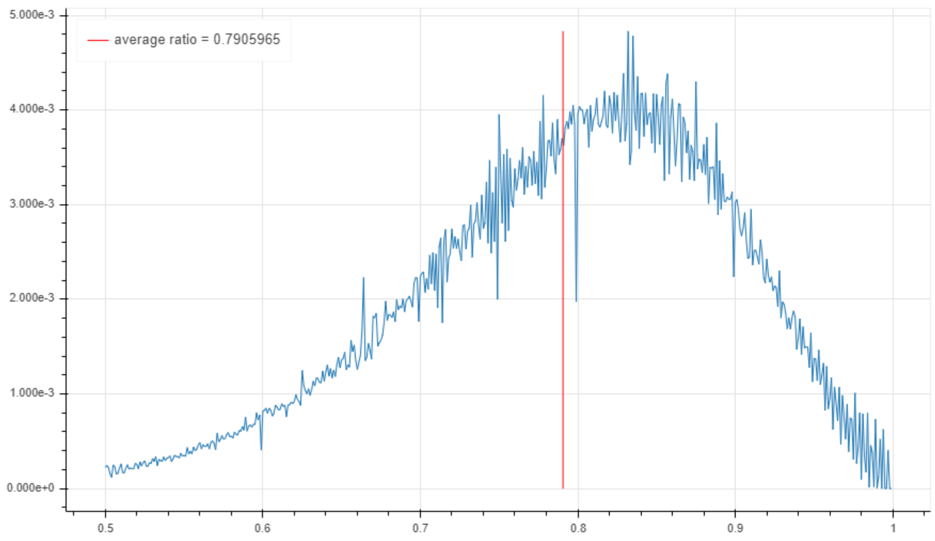

Figure 2: Heuristic distribution of the competitive ratio over 500000 tests with $p = 100, p^{max} = 50, r^{max} = 100, M = 2, N = 4, K = 10$