1 - Introduction

En 5G, une antenne transmet les données aux utilisateurs à travers une connexion wifi via différents channels. Chaque utilisateur, lié à un canal, bénéficie d'une puissance et donc d'une quantité de données qui dépend des préférences l'utilisateur ainsi que du canal (utilisateur proche ou loin de l'antenne).

Un ordonnancement (a scheduling) des données est une distribution des canaux sur les utilisateurs en divisant la puissance totale de l'antenne (budget) sur les canaux diponibles.

On cherche par la suite un ordonnancement optimal en adoptant plusieurs approches algorithmiques (chacune ayant ses avantages et ses défauts en termes de complexité en espace ou en temps). Après un pré-traitement des données du problème, on résoudra premièrement le problème LP (variables relaxées) avec un algorithme glouton, puis, le problème IP avec la programmation dynamique et un algorithme de séparation et évaluation Vers la fin, on s'intéressera à une version en temps réel du problème.

On a choisi **"Python"** comme langage de programmation. Pour lancer les codes, on n'oubliera pas de modifier le chemin d'accès aux fichiers tests dans la fonction "extraction" (chemin d'accès auquel on a enlevé le numéro de fichier).

2 - Formulation du problème

On considère un système avec :

- Un ensemble $\mathcal{K}$ de K utilisateurs
- Un ensemble $\mathcal{K}$ de N canaux
- Un ensemble $[1, .., M]$ de M correspondances.
- Les listes des puissances et des taux des données $(p_{k,m,n}, r_{k,m,n})_{k\in\mathcal{K},m\in[1,..,M],n\in\mathcal{N}}$

On veut maximiser la somme des taux de données sous les contraintes :

1. Chaque canal doit servir un et un seul utilisateur.
2. La puissance totale transmise est inférieure ou égale au budget p.

L'ensemble des triplets solution du problème sera noté $\mathcal{S}$

Question : 1 :

On définit les variables binaires $x_{k,m,n} \in \{0, 1\}$ qui valent 1 si l'utilisateur k est servi par le canal n avec la puissance $p_{k,m,n}$:

On a choisi de ne prendre que ces valeurs de la puissance allouée à chaque utilisateur, car des valeurs intermédiaires conduiraient à utiliser de la puissance sans augmenter l'utilité.

La quantité à maximiser est l'utilité totale, qui vaut $\boxed{\sum_{k\in\mathcal{K}}\sum_{n\in\mathcal{N}}\sum_{m=1}^{M} x_{k,m,n} r_{k,m,n}}$

La contrainte (1) s'écrit : pour tout $n \in \mathcal{N}$, on doit avoir, $\boxed{\sum_{k\in\mathcal{K}}\sum_{m=1}^{M} x_{k,m,n} = 1}$

La contrainte budgétaire (2) s'écrit : $\boxed{\sum_{k\in\mathcal{K}}\sum_{n\in\mathcal{N}}\sum_{m=1}^{M} x_{k,m,n} p_{k,m,n} \leq p}$.

Soit le problème d'optimisation suivant :

$$\begin{cases} \max\limits_{x_{k,m,n}} \left[\sum\limits_{k\in\mathcal{K}}\sum\limits_{n\in\mathcal{N}}\sum\limits_{m=1}^{M} x_{k,m,n} r_{k,m,n}\right] \\ s.to \\ (\forall n \in \mathcal{N}) \sum\limits_{k\in\mathcal{K}}\sum\limits_{m=1}^{M} x_{k,m,n} = 1 \\ \sum\limits_{k\in\mathcal{K}}\sum\limits_{n\in\mathcal{N}}\sum\limits_{m=1}^{M} x_{k,m,n} p_{k,m,n} \leq p \end{cases}$$

3 - Prétraitement

On souhaite éliminer tous les triplets (k, m, n) qu'on est sûr qu'ils ne peuvent appartenir à la solution.

Question 2

Il faut éliminer en premier lieu les triplets (k, m, n) tels que $p_{k,m,n} > p$. Pour ce faire, il faut appeler la fonction *"pretraitement"*. La documentation exacte de cette fonction est en commentaire dans le code.

Question 3

D'autres triplets dits *IP-dominés* ne peuvent également faire partie de la solution.

Un triplet (k, m, n) est *IP-dominé* s'il existe un autre triplet (k', m', n) avec $r_{k',m',n} \leq r_{k,m,n}$ et $p_{k',m',n} \geq p_{k,m,n}$.

Pour les éliminer, nous pouvons, pour chaque valeur de n, trier les listes des triplets k, m, n selon les valeurs de ($p_{k,m,n}$ et si égalité par $r_{k,m,n}$), en un temps $O(KM\log(MK))$ par un algorithme de tri idoine (tri-fusion par exemple). Le problème deviet donc similaire à la recherche de la frontière supérieure d'un nuage de points en 2D (voir figue 1).On parcourt la liste dans l'ordre strictement croissant(s'il 2 triplets consécutifs ont même valeur de $p_{k,m,n}$

on prend le deuxième). A chaque itération, on observe la valeur de $r_{k,m,n}$ et on la compare à la plus grande valeur observée jusqu'alors. Si $r_{k,m,n}$ est plus petit que cette valeur, alors le triplet (k, m, n) est IP-dominé, sinon on prend cette valeur comme nouveau maximum.

Algorithm 1: SuppIPDomines

Input : Données du problème : N, M, K et p et les listes des $p_{k,m,n}$ et $r_{k,m,n}$
Output : Liste des triplets (k, m, n) restants après suppression des IP-dominés.
$NonIPdomines \leftarrow [[] * N]$
for *n=1 to N* **do**
  $L \leftarrow [(k, m), k = 1, \ldots, K, m = 1, \ldots, M]$;
  $max \leftarrow -1$
  for $(k, m) \in L$ **do**
    if $r_{k,m,n} \leq max$ **then**
      On élimine le triplet (k, m, n) qui est IP-dominé.
    end if
    else
      $max \leftarrow r_{k,m,n}$
      ajouter (k, m, n) à $NonIPdomines[n]$
    end if
  end for
end for
return $NonIPDomines$

La complexité totale de notre fonction *SuppIPDomine* est $O(NKM \log(MK))$ (N itérations avec pour chacune un tri de liste de taille KM est un parcours de cette liste)

Question : 4 :

D'autres triplets dits *LP-dominés* ne peuvent faire partie de la solution.

Un triplet (k', m', n') est IP-dominés s'il existe 2 autre triplet (k, m, n) et (k'', m'', n'') avec $p_{k,m,n} \leq p_{k',m',n'} \leq p_{k'',m'',n''}$ et $r_{k,m,n} \leq r_{k',m',n'} \leq r_{k'',m'',n''}$ tels que :
$$\frac{r_{k'',m'',n} - r_{k',m',n}}{p_{k'',m'',n} - p_{k',m',n}} \geq \frac{r_{k',m',n} - r_{k,m,n}}{p_{k',m',n} - p_{k,m,n}}$$

Considérons la situation d'un point de vue graphique(voir figure 1).On confond point et triplet pour n fixé. Une fois que les deux traitements des questions précédentes ont été achevées, on peut, pour toute valeur de n, tracer la courbe des $r_{k,m,n}$ en fonction des $p_{k,m,n}$. Une valeur LP-dominée est située, dans cette situation, sous la corde qui joint deux autres points. Autrement dit, les valeurs LP-dominées sont celles qui empêchent la courbe obtenue d'être concave. On peut donc procéder de la façon suivante pour toute valeur de n :

- Créer une liste *NonLPDomines* des points solutions.

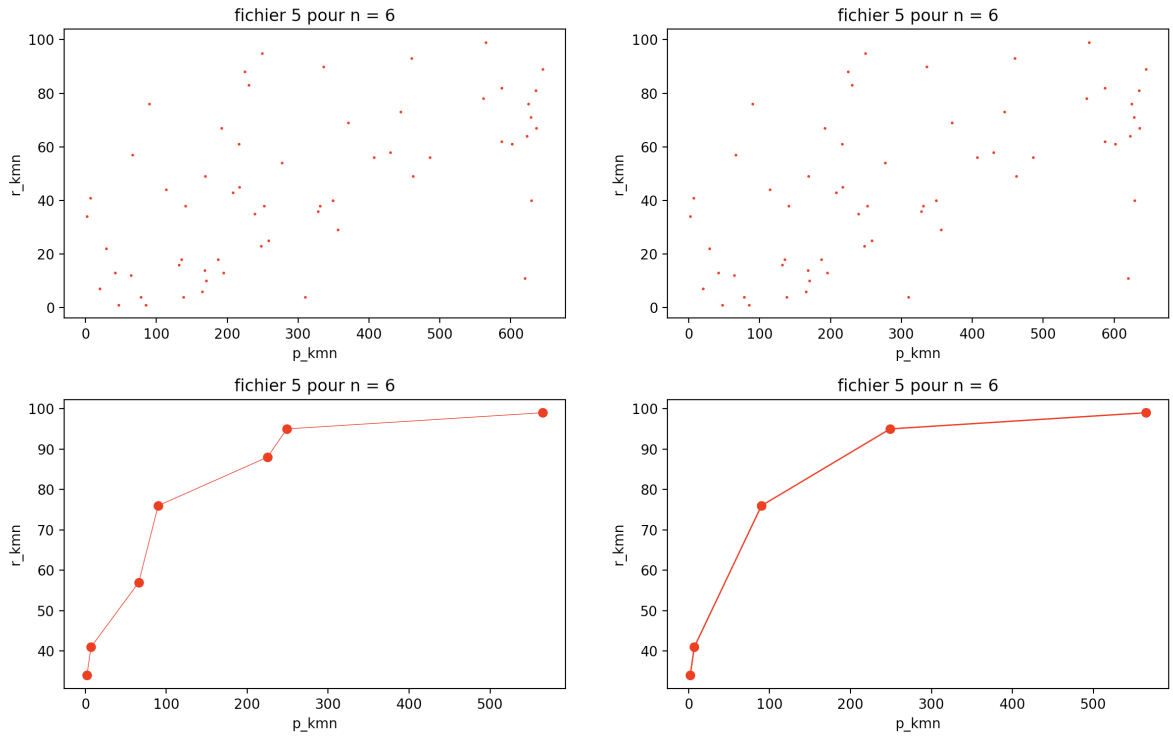

Figure 1: L'ensemble des points (k,m,5) pour $n = 6$ dans le fichier 5 et le résultat progressif des fonctions de prétraitement (de gauche à droite, du haut au bas).

- Parcourir les valeurs de $p_{k,m,n}$ dans l'ordre croissant. Avec le traitement fait à la question précédente, les valeurs de $r_{k,m,n}$ seront aussi en ordre croissant.

- On supprime tous les points de "$NonLPDomine$" tel que la pente entre ce point et son précédent dans la même liste est inférieure à celle entre ce point et le point courant de la boucle (ceci nécessite que la longueur de "$NonLPDomine$" est au moins 2) en commencant par le dernier point de la liste. Puis, on ajoute le point courant à "$NonLPDomine$".

  Le tri au début se fait en un temps $O(KM \log(KM))$ au pire des cas. Le parcours se fait en un temps $O(KMN)$ car chaque point est supprimé de "$NonLPDomine$" au plus une fois. Donc la complexité finale au pire sera bien en $O(KMN \log(KM))$. (On peut montrer même qu'elle est optimale).

  Question : 5 :

  - On tourne les algorithmes précédent sur les fichiers test et le résultat est donné (voir figure 2. Pour le fichier 2, tous les triplets sont éliminés après le premier traitement. (On a ajouté au code des compteurs pour calculer le nombre de triplets éliminés et restants pour chaque fichier).

Algorithm 2: SuppLPDomines

Input : Données du problème : N, M, K et p et les listes des $p_{k,m,n}$ et $r_{k,m,n}$
Output : Liste des triplets (k, m, n) restants après suppression des LP-dominés.
$NonLPdomines =\leftarrow [[] * N]$
$NonLPdomines = SuppIPDomines()$
for *n=1 to N* **do**
  $L \leftarrow NonIPDomine[n]$;
  for $(k'', m'') \in L$ **do**
    while *(NonLPDomines contient plus que 2 éléments)* **do**
      $(k, m) \leftarrow NonLPDomines[-2]$ $(k', m') \leftarrow NonLPDomines[-1]$
      if $\left(\frac{r_{k'',m'',n} - r_{k',m',n}}{p_{k'',m'',n} - p_{k',m',n}} \geq \frac{r_{k',m',n} - r_{k,m,n}}{p_{k',m',n} - p_{k,m,n}}\right)$ **then**
        On élimine le triplet (k', m', n') qui est LP-dominé.
      end if
    end while
    On ajoute (k",m",n") à $NonLPDomines$
  end for
end for
return $NonLPDomines$

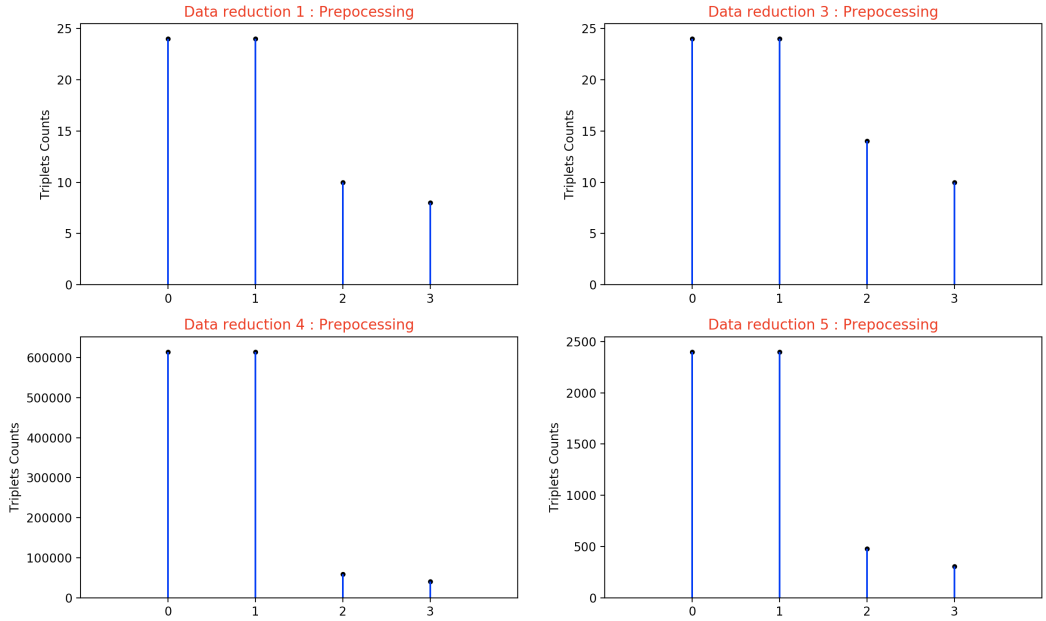

Figure 2: Réduction des nombres des triplets après chaque traitement. 0 correspond au nombre initial des triplets, 1 au nombre après le premier traitement.

4 - LP problème et algorithme glouton :

Dans cette partie, on résoud le problème en relaxant les variables binaires en variables continues sur $[0, 1]$.

Question : 6 :

L'idée intuitive consiste à investir dans le canal qui a le plus fort potentiel, c'est à dire celui qui a le meilleur rendement (e).

Pour cela, on utilisera une structure de données qui liera chacun de nos triplets à une efficacité (qui n'est que la pente entre ce point et le point qui le précède dans le résultat de la fonction "SuppLPDomines"). On note qu'on commence pour chaque n du deuxième point !

On trie après notre structure en ordre croissant de la valeur des pentes ce qui coûtera au pire $O(KMN \log(KMN))$.

On commence par une solution constituée des N premiers points des listes de la liste résultat de "SuppLPDomines" et une puissance totale égale aux sommes des puissances des triplets de la solution.

Tant que le budget p n'est pas atteint et tant que tous les utilisateurs n'ont pas l'utilité maximale : On retire le triplet de meilleur pente de notre structure et on l'insère à la place du triplet ancien du même canal si la nouvelle puissance totale est inférieure ou égale à p, sinon on réalise une combinaison des deux triplets (affectation partielle) de sorte à épuiser le budget et on insère les 2 triplets, ce qui permet d'avoir des coefficients non-entiers sur une même valeur de n. On oublie pas d'incrémenter l'utilité totale aussi.

Dans le pire des cas, le budget est suffisant pour que tous les utilisateurs atteignent l'utilité maximale. On a donc effectué $O(KMN)$ améliorations, donc la complexité totale est en $O(KMN \log(KMN))$. Voir l'algorithme 3.

Question : 7 : On tourne notre Greedy Algorithm et le solveur LP de python (sous pulp) et on obtient :

Tableau des résultats				
	puissance totale		utilité optimale	
Fichier	Greedy	LP solver	Greedy	LP solver
test1	78	78	365	365
test2	0	0	0	0
test3	100	99.99	371.15	372.15
test4	16000	16000	9870.30	9870.32
test5	1000	1000	1637	1637

- Les valeurs trouvés sont très proches.

En ce qui concerne le temps de calcul, on trouve :

Algorithm 3: GreedyAlgo

Input : Données du problème : N, M, K et p et les listes des $p_{k,m,n}$ et $r_{k,m,n}$
Output : *Solution* Liste des triplets (k, m, n) solution du LP problème(approximé).
$Solution =\leftarrow [[] * N]$
$p_{tot} =\leftarrow [[] * N]$
$U_{tot} =\leftarrow [[] * N]$
$NonLPdomines = SuppLPDomines()$
$TripPentes =$ Un dictionnaire liant triplet et sa pente
for *n=1 to N* **do**
    $Solution[n] \leftarrow NonLPDomine[n][0]$;
    $(k, m, n) = NonLPDomine[n]$;
    p_{tot} += $p_{k,m,n}$;
    U_{tot} += $r_{k,m,n}$;
end for
for *n=1 to N* **do**
    foreach *($t \in NonLPDomine$ sauf le 1^{er})* **do**
        $pente = pente(t)$; calcule la pente
        $TripPentes[t] = pente$;
    end foreach
end for
Trier *TripPentes* selon l'ordre croissant des pentes
while *($p_{tot} < p$ and TripPentes n'est pas vide)* **do**
    $(k1, m1, n1)$ = dernier élément de *TripPentes*
    $(k, m, n) = Solution[n1]$
    $p_{tempo} = p_{tot} + p_{k1,m1,n1}$ - $p_{k,m,n}$
    if *($p_{tempo} < p$)* **then**
        $Solution[n] = (k1, m1, n1)$
        $p_{tot} = p_{tempo}$
        $U_{tot} = U - tot + r_{k1,m1,n1}$
    end if
    else
        $x = \frac{p - p - tot + p_{k,m,n} - p_{k1,m1,n1}}{p_{k,m,n} - p_{k1,m1,n1}}$
        $p_{tot} = p_{tot} + xp_{k,m,n} + (1 - x)p_{k1,m1,n1}$
        $U_{tot} = U_{tot} + xr_{k,m,n} + (1 - x)r_{k1,m1,n1}$
    end if
end while
print(La fraction est : x)
return *Solution*

temps en secondes		
	Greedy	LP solver
test1	0.000138	0.03187
test2	4.84e-5	0.0097
test3	0.00014	0.012
test4	0.23	0.53
test5.	0.0008	0.02

- L'algorithme glouton calcule donc une solution approchée (qui est presque la vraie solution) en un temps raisonnablement court.

5 - Algorithme pour la résolution de l'ILP

On recherche ici une solution au ILP problème lorsque le budget p est un entier. On considère 2 approches : l'une avec la programmation dynamique, l'autre concerne la Branch and Bound méthode.

Question : 8 :

Nous nous inspirons de l'algorithme en programmation dynamique qui résout le *problème du sac à dos* sauf qu'ici le canal doit être lié toujours (à discuter dans la présentation). Définissons les variables $U[i, \pi]$ pour $i \in \mathcal{N}, \pi \in [\![0, p]\!]$, égales à l'utilité maximale que l'on peut obtenir en utilisant uniquement les canaux 1 à i et avec une puissance maximale $\pi \leq p$.

La taille de cette matrice est Np. De plus on dispose de la formule de récurrence (DP equation) :

$$U[i,\pi] = \begin{cases} 0 \text{ si i} = 0 \\ \max\limits_{p_{k,m,i} \leq \pi} U[i-1, \pi - p_{k,m,i}] + r_{k,m,i} \text{ sinon} \end{cases}$$

Il en découle l'algorithme 4.

Dans notre algorithme primal : Nous avons Np itérations, et pour chacune, on explore au pire KM possibilités d'un côté puis N itérations pour la construction de la solution à la fin. Ce qui fait un $O(NKMp)$ pour la complexité en temps.

Pour la complexité spatiale, on voit bien qu'il nous a fallu 2 matrices d'ordre Np pour trouver le résultat. D'où une complexité spatiale en $O(Np)$. On note que cette solution est plus performante que celle dans laquelle on a pu stocker chaque fois (chaque sous problème) la solution complète et non le triplet optimal.

Question : 9 :

On cherche maintenant la solution en utilisant l'approche duale qui consiste à transformer le problème de maximation de l'utilité en un problème de minimisation de la

Algorithm 4: DynProSolutionPrimal

Input : Données du problème : N, M, K et p et les listes des $p_{k,m,n}$ et $r_{k,m,n}$
Output : U Matrice des utilités maximales des sous-problèmes et $TRiplets - Sol$ la solution.
$U =\leftarrow [[0] * (p+1)]^*(N+1)$ $TripSolSub =\leftarrow [[0] * (p+1)]^*(N+1)$ contient le triplet optimal pour le sous problème correspondant
$NonLPdomines = SuppLPDomines()$
for π *= 1 to N* **do**
  $U[0][\pi] =\leftarrow 0$;
end for
for *n=1 to N* **do**
  for π *= 0 to N* **do**
    foreach $(t \in NonLPDomine)$ **do**
      $(k, m, n) \leftarrow t$;
      if $p_{k,m,n} \leq \pi$ *et* $r_{k,m,n} + U[n-1, \pi - p_{k,m,n}] \geq U[n, \pi]$ **then**
        $U[n, \pi] \leftarrow r_{k,m,n} + U[n-1, \pi - p_{k,m,n}]$; $TripSolSub[n] \leftarrow (k, m, n)$;
      end if
    end foreach
  end for
end for
On construit maintenant la solution
$TripletsSol \leftarrow [[] * N]$
$c \leftarrow p$
$p_{tot} \leftarrow 0$
for *n=N to 1, -1* **do**
  $t \leftarrow TripSolSub[n][c]$;
  $k, m, n \leftarrow t$;
  $c \leftarrow c - p_{k,m,n}$;
  $p_{tot} \leftarrow p_{tot} + p_{k,m,n}$;
  $TripletsSol[n] = t$;
end for
print(L'utilité maximale est : U[N][p])
print(La puissance totale est : p_{tot}
return $TripletsSol$

puissance totale. Notons que pour trouver une solution, il faut se donner une borne supérieure pour l'utilité u ce qui évitera de chercher la solution d'une infinité de sous problèmes.

Le problème de minimisation est le suivant :

$$\begin{cases} \min\limits_{x_{k,m,n}} \left[\sum\limits_{k\in\mathcal{K}} \sum\limits_{n\in\mathcal{N}} \sum\limits_{m=1}^{M} x_{k,m,n} p_{k,m,n} \right] \\ s.to \\ (\forall n \in \mathcal{N}) \sum\limits_{k\in\mathcal{K}} \sum\limits_{m=1}^{M} x_{k,m,n} = 1 \\ \sum\limits_{k\in\mathcal{K}} \sum\limits_{n\in\mathcal{N}} \sum\limits_{m=1}^{M} x_{k,m,n} r_{k,m,n} \geq \rho \end{cases}$$

On suppose d'abord que l'utilité maximale est majorée par $U > 0$. Nous reviendrons sur cette valeur plus tard. Soit donc $C[n+1, \rho]$ le coût minimal en puissance à dépenser pour obtenir une utilité d'au moins ρ. On a la relation de récurrence :

$$C[i,\rho] = \begin{cases} \sum\limits_{n\in\mathcal{N}} \max\limits_{k,m}(r_{k,m,n}) \text{ si i} = 0 \\ \min(\min\limits_{r_{k,m,i}\leq\rho} C[i-1, \rho - r_{k,m,i}] + p_{k,m,i}, \min\limits_{r_{k,m,i}>\rho} p_{k,m,i}) \text{ sinon} \end{cases}$$

Il en découle l'algorithme 5.

Comme pour l'algorithme primal : Nous avons NU itérations, et pour chacune, on explore au pire KM possibilités d'un côté puis N itérations pour la construction de la solution à la fin. Ce qui fait un $O(NKMU)$ pour la complexité en temps.

Pour la complexité spatiale, on voit bien qu'il nous a fallu 2 matrices d'ordre NU pour trouver le résultat. D'où une complexité spatiale en $O(NU)$.

Question : 10 :

On adopte la démarche Branch and Bound pour résoudre le LP probème maintenant. On commence par définir nos noeus.

On prend comme racine un objet(noeud) qui représente le problème initial avec les contraintes initiales.

On imagine ensuite un arbre de hauteur $N+1$ dans lequel chaque noeud est un sous-problème (problème initial auquel on ajoute des contraintes supplémentaires). Ainsi, le noeud situé à un niveau n pour $n = 0, \ldots, N$ a ses n premiers canaux déjà utilisés et le chemin de la racine à ce noeud nous donne les n premiers triplets. Ce noeud donne comme KM fils (si cela est bien envisageable) dans lequels le canal $n+1$ est utilisé.

Les noeuds seront construits progressivement !

L'idée est donc représenter une classe "Sub-Problem" qui contient le sous problème de manière à accéder à tous les données à partir de ce noeud (le chemin qui emmène de

Algorithm 5: DynProSolutionDual

Input : Données du problème : N, M, K et p et les listes des $p_{k,m,n}$ et $r_{k,m,n}$ et U
Output : U Matrice des coûts minimales des sous-problèmes et $TRiplets - Sol$ la solution.
$U =\leftarrow [[0] * (U+1)]^*(N+1)$
$TripSolSub =\leftarrow [[0] * (U+1)]^*(N+1)$ contient le triplet optimal pour le sous problème correspondant
$NonLPdomines = SuppLPDomines()$
$romax =\leftarrow$la somme des utilités maximales des $NonLPDomines[n]$ pour $n \in \mathcal{N}$
for ρ = *1 to* U **do**
  $C[0][\rho] =\leftarrow romax$;
end for
for n=*1 to* N **do**
  for ρ = *0 to* N **do**
    foreach $(t \in NonLPDomine)$ **do**
      $(k, m, n) \leftarrow t$;
      if $r_{k,m,n} \leq \rho$ **then**
        if $r_{k,m,n} + C[n-1, \rho - p_{k,m,n}] \geq C[n, \pi]$ **then**
          $C[n, \rho] \leftarrow r_{k,m,n} + C[n-1, \rho - p_{k,m,n}]$; $TripSolSub[n] \leftarrow (k, m, n)$;
        end if
      end if
      else
        $C[n, \rho]p_{k,m,n}]$; $TripSolSub[n] \leftarrow (k, m, n)$;
      end if
    end foreach
  end for
end for
On construit maintenant la solution
$TripletsSol \leftarrow [[] * N]$
$c \leftarrow U$
$U_{tot} \leftarrow 0$
for n=N *to 1, -1* **do**
  $t \leftarrow TripSolSub[n][c]$;
  $k, m, n \leftarrow t$;
  $c \leftarrow c - r_{k,m,n}$;
  $U_{tot} \leftarrow U_{tot} + r_{k,m,n}$;
  $TripletsSol[n] = t$;
end for
print(L'utilité maximale est : C[N][p])
print(La puissance totale est : U_{tot}
return $TripletsSol$

la racine à ce noeud - budget total) puis de parcourir les noeuds de l'arbre en partant de la racine avec un parcours en profondeur (en utilisant donc une pile des noeuds) et pour chaque noeud v :

calculer une approximation de la valeur optimale de l'utilité pour les canaux non appariés restants avec la méthode de classe "ComputeUpperBound", en déduire l'utilité maximale z_v :

Si z_v est inférieure à une solution faisable déjà trouvée, on abondonne cette branche.

Si z_v est liée à une solution entière, on la prend comme nouvelle solution faisable selon qu'elle est supérieure ou inférieure à la solution calculée jusqu'à l'instant, et on ajoute à la pile les nouveaux noeuds fils avec la méthode de classe "neighbors()".

Sinon, on abondonne la branche.

Ainsi, le pseudo-code de la classe "SubProblem" est 6 :

Algorithm 6: Class : Sub-Problem

Input : Données du problème : N, M, K et p et les listes des $p_{k,m,n}$ et $r_{k,m,n}$ et U
Class Sub-Problem{
triplet; le triplet en question
level; le niveau coreespondant
path; le chemin de la racine au noeud
p; budget total
puissances; liste des $p_{k,m,n}$
utilites; liste des $r_{k,m,n}$
neighbors(); elle renvoie les noeuds fils qui ne sont pour lesquels le triplet n'est pas LP-dominé
ComputeUpperBound(); elle indique s'il existe une solution faisable (LP) et renvoie l'utilité optimale pour un noeud (z_v) ainsi que la liste des triplets solution si la valeur optimale est lié à une solution entière.

et l'algorithme du Branch and Bound associé est 7.

Dans le pire des cas , on exploitera tous les noeuds de l'arbre ce produit une complexité temporelle exponentielle en N : $O((KM)^N$ (chaque noeud donne naissance à KM fils).

Dans le meilleur des cas, la solution optimale est vitement trouvée dès le premier noeud et on se retrouve avec un $O(KM)$.

Question : 11 : Les résultats sont comme suit :

Algorithm 7: BBAlgo

Input : Données du problème : N, M, K et p et les listes des $p_{k,m,n}$ et $r_{k,m,n}$ et U
Output : *Solution* la solution et P_{tot} la puissance correspondante et U_{tot} l'utilité optimale.
$NonLPdomines = SuppLPDomines(...)$
$Solution = []$
$U_{tot} = 0$
$p_{tot} = 0$
sub-Prob = new Sub-Problem$((), 0, [], p, (p_{k,m,n}), (r_{k,m,n}))$
pile = () vide
while *pile n'est pas vide* **do**
    retirer un élément sub-prob de la *pile*
    subSol = sub-prob.ComputeUpperBound()
    if *(subSol existe = il y'a une solution (LP))* **then**
        if *(l'utilité optimale de SubSol (z_v) est supérieure strictement à U_{tot})* **then**
            if *(la solution est entière* **then**
                $U_{tot} = z_v$ p_{tot} = la puissance totale correspondante *Solution* = la solution du sous problème avec le chemin("path")
            end if
            else
                pile.empiler(sub-prob.neighbors())
            end if
        end if
    end if
end while
print(La puissance totale est p_{tot})
print(L'utilité maximale est U_{tot})
return *Solution*

Tableau des résultats						
	puissance totale		utilité optimale		temps en secondes	
Fichier	DP	BB	DP	BB	DP	BB
test1	78	78	365	365	0.0018	0.115
test2	0	0	0	0	0.001	0.008
test3	68	68	350	350	0.002	0.138
test4	16000	–	9870	–	19min30	–
test5	1000	–	1637	–	0.48	–

- On voit bien que le DP renvoie une bonne solution en un temps raisonnable pour les petites tailles. Mais le temps d'exécution devient très grand pour de grandes tailles d'où l'utilité de l'algorithme du Branch and Bound qui dans ses meilleurs cas doit produire une solution ayant une complexité inférieure à ceux du DP.

Question : 12 :

Pour cette question, on s'inspire de l'algorithme glouton défini à la question 6. La différence est que l'on ne connaît pas en avance les valeurs des $p_{k,m,n}$ et de $r_{k,m,n}$ pour les utilisateurs qui ne sont pas encore arrivés. Cependant, on peut tout à fait utiliser l'algorithme glouton pour déterminer la puissance à attribuer à l'utilisateur courant, sans modifier les puissances attribuées aux utilisateurs précédents. Ensuite, faisons des suppositions pour les utilisateurs dont on ne connaît pas encore les valeurs : pour cela, on effectue des tirages aléatoires qui donnent des valeurs aux triplets inconnus.

L'algorithme résultant est décrit en 8.

Question : 13 :

Les résultats de la question (13) sont dans le code python. Il faut appeler la fonction "grapheTempsReel(iterations)" avec "iterations" le nombre de problèmes sur lesquelles on veut comparer l'Online scheduling et le Greedy algorithm.

Algorithm 8: AlgoTempsReel

K_{max} est une donnée du problème qui correspond à la valeur de l'utilisateur courant.
Les valeurs de $p_{k,m,n}$ et $r_{k,m,n}$ sont randomisées avant prétraitement.
foreach n **do**
    $L[n] \leftarrow$ Liste des triplets (k, m, n) triés selon la valeur de $p_{k,m,n}$
end foreach
Cette opération se fait en temps $O(KM \log(KM))$; on obtient un tableau à 2 dimensions.
$U \leftarrow$ Liste des $e_{1,n}$
$V \leftarrow$ Liste des valeurs de n triées selon les valeurs de U, telles que dans l'affectation actuelle, le canal n sert un utilistaeur $k \leq K_{max}$.
$W \leftarrow$ Tableau de longueur n initialisé avec des 0 indiquant les valeurs de l.
$X \leftarrow$ Tableau des $x_{k,m,n}$
$pr \leftarrow p$
while $U[V[-1]] > 0$ *et* $pr >= 1$ **do**
    $n \leftarrow V[-1]$
    Comment : Pour des raisons pratiques d'encodage des nombres flottants, on ne teste pas si le reliquat de puissance est nul.
    if $pr \geq p[L[n][W[n]+1]] - p[L[n][W[n]]]$ **then**
        $pr \leftarrow pr - (p[L[n][W[n]+1]] - p[L[n][W[n]]])$
        $X[L[n][W[n]]] \leftarrow 0$
        $X[L[n][W[n]+1]] \leftarrow 1$
        $W[n] \leftarrow W[n] + 1$
        On retire la valeur de U[V[-1]].
        $U[n] \leftarrow e_{W[n],n}$
        Retrait puis insertion à nouveau de n à sa place dans le tableau V si on a $W[n] \leq KM - 1$.
    end if
    else
        Quitter la boucle.
    end if
    Comment :Afin d'obtenir des coefficients entiers, on n'épuise plus la puissance restante.
end while