INF421 - 5G scheduling

Calot–Plaetevoet Paul

3 Février 2020

1 Introduction

2 Formulation du problème

2.1 Question 1).

Le problème se met sous la forme ILP classique suivante :

Maximiser $R^\top X$ avec : $\left\{\begin{array}{l} R = (r_{k,m,n})_{m\in(1,...,M),k\in(1,...,K),n\in(1,...,N)} \\ X = (x_{k,m,n})_{m\in(1,...,M),k\in(1,...,K),n\in(1,...,N)} \end{array}\right.$ tels que :

- $AX < (1,...,1)^\top$, de taille KNM et où A est telle que : $\forall i \in (1,...,N), \forall j \in (1,...,KNM)$, $A_{i,j} = \left\{\begin{array}{l} 1si(j-1)MK \leq i \leq jMK \\ 0sinon \end{array}\right.$. Cela pour assurer qu'on aie qu'une seul $x_{k,m,n} = 1$ par canal.
- $X \geq 0$.
- $X \in \mathbf{Z}$ et même : $\forall i, X_i \in 0,1$.

3 Preprocessing

3.1 Question 2).

Les triplets (k_0, m_0, n_0) n'étant pas pertinents pour la recherche d'une solution sont tels que :

- On a $p_{(k_0,m_0,n_0)} + \sum_{n=1,n\neq n_0}^{N} \underset{k,m}{min}(p_{(k,m,n)}) > P$ où P est la puissance maximale que peut fournir l'antenne.
- Il existe un autre triplé (k_1, m_1, n_1) tel que : $\left\{\begin{array}{l} p_{k_0,m_0,n_0} = p_{k_1,m_1,n_1} \\ r_{k_0,m_0,n_0} < r_{(k_1,m_1,n_1} \end{array}\right.$.
- Il existe un autre triplé (k_1, m_1, n_1) tel que : $\left\{\begin{array}{l} {}_{k_0,m_0,n_0} = r_{k_1,m_1,n_1} \\ p_{k_0,m_0,n_0} > p_{(k_1,m_1,n_1} \end{array}\right.$.

3.2 Question 3).

Algorithme 1 Éliminer les termes IP-dominés

```
procédure removeIPDominated
    pour n = 0 à N faire
        tant que il reste des triplets qui sont supprimés (on s'arrête lorsque
                plus aucun triplet n'est supprimé) faire
            Trier les triplets du channel n pour le comparateur Comparator (tri
                    fusion).
            Enlever les triplets sélectionnés.
        fin tant que
    fin pour
fin procédure

procédure Comparator(Triplet t1, Triplet t2)
    si r1 > r2 alors
        si p1 ≤ p2 alors
            x2 = 0
        fin si
        retourne 1
    sinon
        si p2 ≤ p1 alors
            x1 = 0
        fin si
        retourne −1
    fin si
fin procédure
```

On réalise un tri fusion avec ce comparateur qui en plus de ranger les triplets selon l'ordre indiqué les élimine si ils sont IP-dominé. Ce tri est réalisé d'abord sur r, puis sur p.

Du point de vue de la complexité, on réalise N tris fusion sur moins de KM éléments. Si on oublie la boucle *while* de la première procédure, on obtient donc une complexité en $O(NKMlog(KM))$. J'ai remarqué en pratique l'utilité de la boucle *while* qui permettait d'éliminer quelques triplets additionnels.

3.3 Question 4).

Le formule (2) permet de voir qu'on cherche l'enveloppe concave supérieure des triplets représentés dans un repère ayant pour abscisse p et ordonnée r. Calcul de la complexité :

- N tours de boucle.
- Tri des triplets (tri fusion) : $O(KMlog(KM)$. Le parcours des triplets et

---

Algorithme 2 Éliminer les termes LP-dominés

Input : la liste des triplets de chaque channel.

procédure *removeLPDominated*

  pour $n = 0$ à N **faire**

    Tri des triplets du channel n à p croissant.

    initialiser L comme un tableau de triplets de taille le nombre de triplets dans le channel. L contient le triplets constituant l'enveloppe concave supérieure.

    entier k qui correspondont au nombre de triplets pertinents dans L.

    tant que il y a des triplets dans le channel **faire**

      t : un nouveau triplet

      tant que L contient deux points et la séquence constituée des deux derniers points de L et de t ne tourne pas dans le sens trigonométrique. **faire**

        Enlever le dernier point de L

      fin tant que

      Ajouter t à L

    fin tant que

    Enlever les triplets présents dans L à la liste des triplets du channel n.

  fin pour

fin procédure

procédure CROSS(Triplet a, Triplet b, Triplet c)

  retourne $(b_p - a_p)(c_r - a_r) - (b_r - a_r)(c_p - a_p)$ // Cela correspond à (2).

fin procédure

---

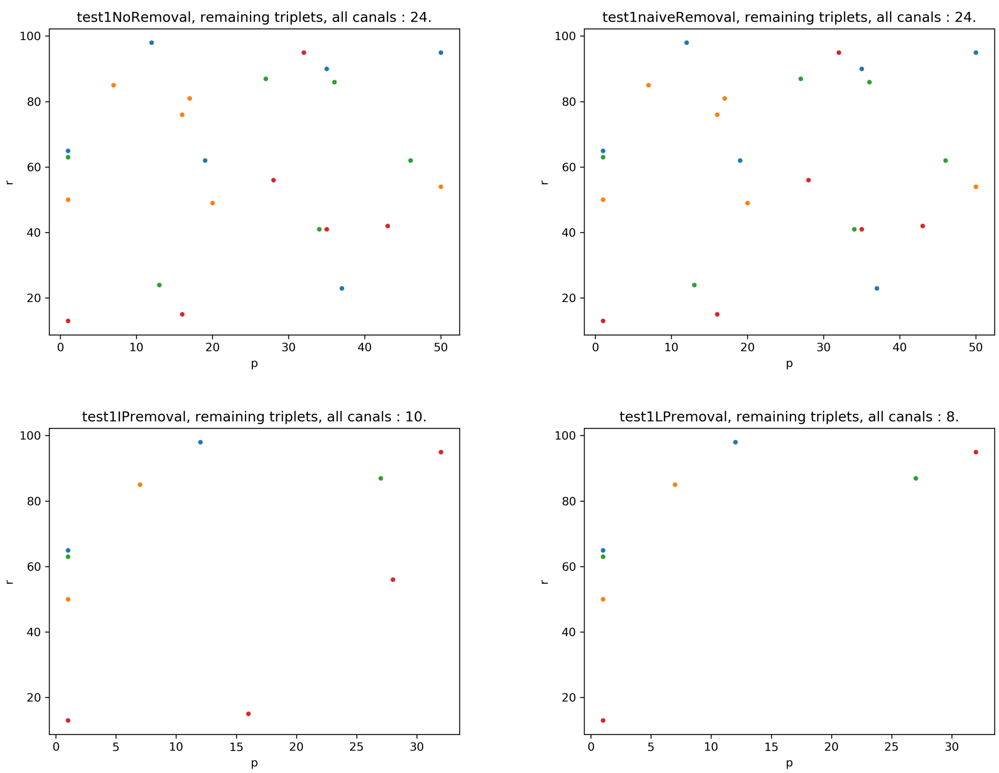
l'application de la fonction $Cross$ étant de complexité linéaire $O(KM)$. Enlever les triplets prend également un temps linéaire (parcours de liste).

En conclusion, on a atteint une complexité en $O(NKMlog(KM))$.

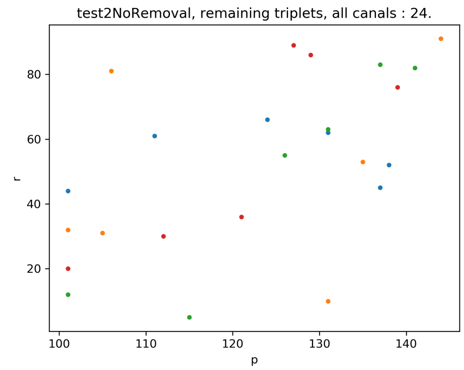
3.4 Question 5).

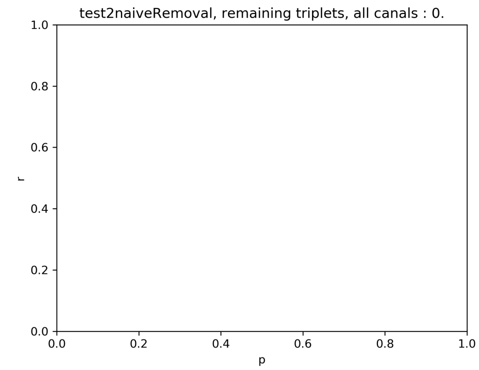
Les points de même couleurs font partis du même channel.

3.4.1 Pour le fichier texte 1, on obtient :

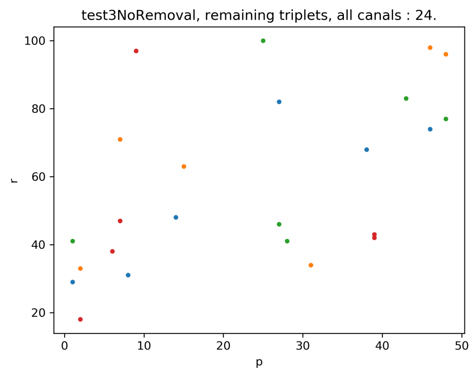


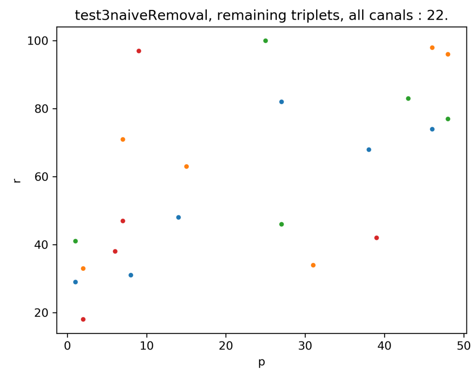
3.4.2 Pour le fichier texte 2, on obtient :

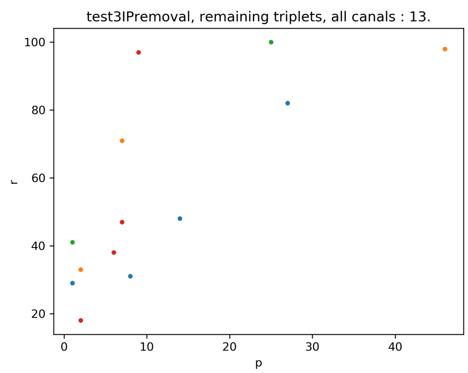


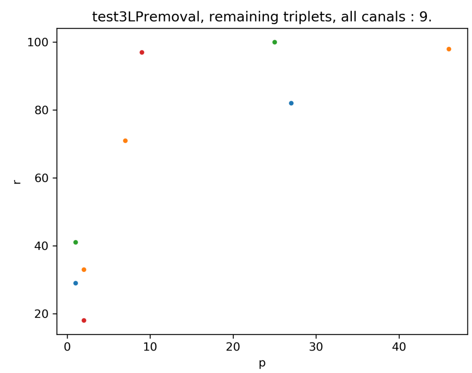


3.4.3 Pour le fichier texte 3, on obtient :

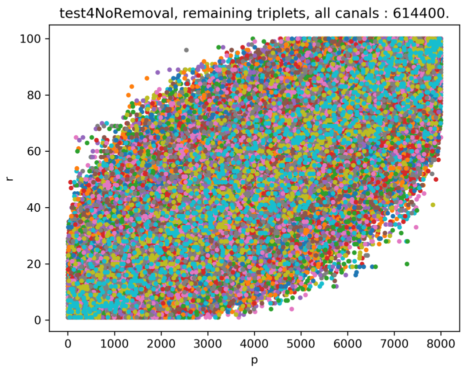


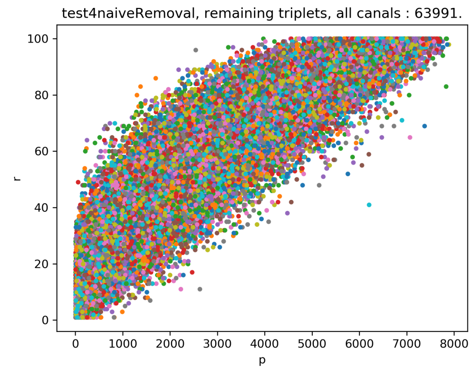


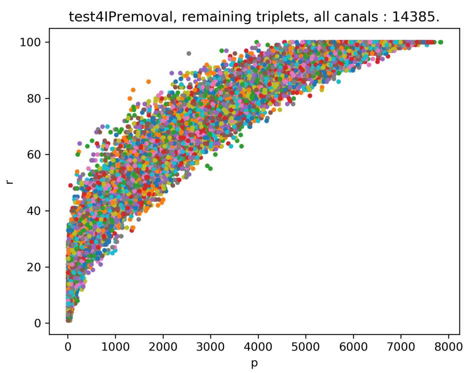


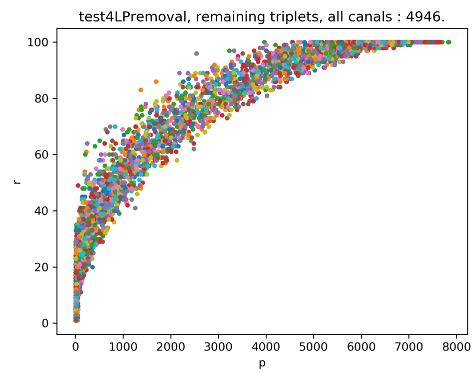


3.4.4 Pour le fichier texte 4, on obtient :

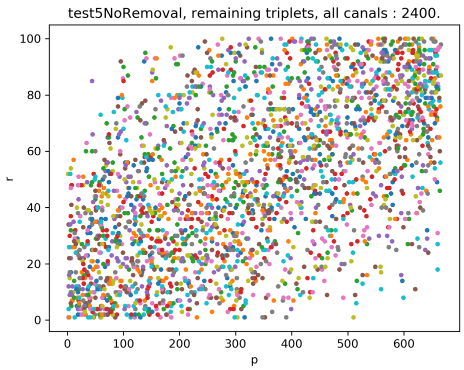


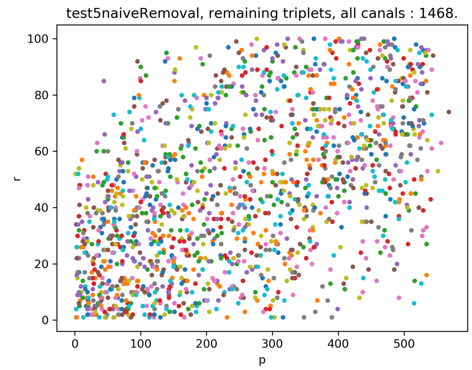


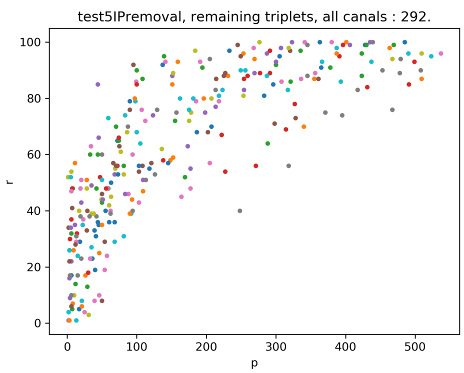


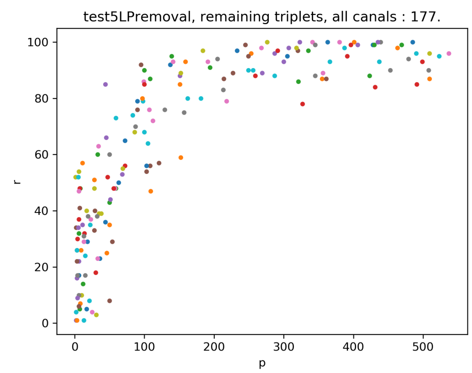


3.4.5 Pour le fichier texte 5, on obtient :









Remarques :

- Pour les fichier aux plus petits nombre de points, on peut voir que, pour un channel donné, il n'existe pas deux triplets verticalement ou horizontalement alignés. On remarque également que tous les triplets du fichier ont été enlevés car la puissance nécessaire pour ceux-ci étaient trop élevés.
- On a bien à la fin de cette deuxième étape de preprocessing, si t_1 et t_2 appartiennent au même channel alors : $p_1 > p_2 \Leftrightarrow r_1 > r_2$.
- Enfin, en comparant les résultats à la fin de l'IP-preprocessing et du LP-preprocessing on remarque bien, que pour un channel donné, on a sélectionné l'enveloppe concave supérieure.

4 Programmation linéaire et algorithme glouton

4.1 Question 6).

Étapes préalables :

- Sélection de tous les triplets dans une liste.
- Tri de ces triplets selon leur puissance.

Algorithme 3 Algorithme glouton

Input : la liste des triplets de chaque channel.
procédure GREEDYSOLVER
    Soit L, la liste contenant tous les triplets triés par p croissant.
    Soit *solution* un tableau de triplets
    $requiredPower \leftarrow 0$ // la puissance nécessaire pour la solution actuelle.
    tant que il reste des triplets non visités dans L et requiredPower ¡ P = puissance maximale délivrable **faire**
        Triplet t : triplet suivant dans la liste L.
        si *solutions* ne contient pas de triplet du channel n **alors**
            si $\sum_{i \in solution} p_i + p_t \leq P$ **alors**
                $requiredPower \leftarrow requiredPower + p_t$
                t est ajouté à *solutions* avec $x_t = 1$
            sinon
                t est ajouté à *solutions* avec x_t tel que $requiredPower + x_t p_t = P$
                $requiredPower \leftarrow P$
            fin si
        sinon
            si $\sum_{t_i \in solution} p_i + p_t - p_{n_{t_i} = n_t} \leq P$ **alors**
                $requiredPower \leftarrow requiredPower + p_t - p_{n_{t_i} = n_t}$
                t est ajouté à *solutions* avec $x_t = 1$ et remplace le précédent triplet.
            sinon
                t est ajouté à *solutions* avec x_t tel que $requiredPower + x_t p_t + (1 - x_t) p_{n_{t_i} = n_t} = P$
                $requiredPower \leftarrow P$
            fin si
        fin si
    fin tant que
fin procédure

4.2 Question 7) Résultats de l'algorithme glouton

On obtient les résultats suivants :

$Test$	1	3	4	5
$Power$	78	99	15999	999
$UserRate$	365	371	9307	1523
$time(ms)$	0.10	0.17	4.17	0.39

Le test 2 ne donnant aucune solution (cf. question 5).

5 Algorithmes pour résoudre l'ILP

5.1 Question 8) Programmation dynamique

On note $R(n,p)$ le meilleur "user rate" pour le sous-problème correspondant à une puissance maximale de p et à l'utilisation des n premiers channels. On a l'équation : $R(n,p) = \underset{t/n_t=n}{max}\,(R(n-1, p-p_t) + r_t)$ L'algorithme 4 (ci-après) détaille cela.

La complexité temporelle de cet algorithme est $O(NPMK)$:

- Pour chaque channel : $O(N)$
- Pour tout $p \leq P$
- Pour tous les triplets du channel donné : $< KM = O(KM)$

La complexité spatiale de cet algorithme est $O(N^2P)$ pour le stockage des solutions successives.

5.2 Question 9) Programmation dynamique

5.3 Question 10) Branch-and Bound

L'idée de l'algorithme est la suivante :

1. On part du problème pour lequel on possède une liste des triplets pour chaque channel. C'est la racine.
2. On construit ensuite les branches à partir de cette racine. Les i-èmes branches sont créées en ajoutant à la solution temporaire de l'instance actuelle, chaque triplet du i-ème channel. On construit donc autant de nouvelles branches qu'il y a de triplets dans ce channel.
3. On calcule alors la borne supérieure pour chaque nouvelle instance.
4. On regarde alors l'instance qui possède la borne supérieure la plus élevée. Et on revient à l'étape 2, jusqu'à trouver une solution.

Les conditions d'arrêt ou d'obtention d'une solution sont détaillées ci-après.

L'algorithme BB se décompose en 3 principales fonctions :

- Une fonction *branch* créant les nouvelles branches à partir d'une instance du problème. Elle renvoie une pile (LIFO) contenant les nouvelles sous-instances du problème, triées par borne supérieure décroissante.
- Une fonction *bound* qui renvoie la borne supérieure d'une instance.
- Une fonction *solution*, récursive, qui renvoie la solution.

Il est nécessaire de commencer par créer une instance du système. Pour cela, on créé la classe *BBInstance*. Cette classe possède les champs suivants :

Algorithme 4 Algorithme DP

```
Input : la liste des triplets de chaque channel, trié à p croissant.
procédure dpProcessing
    optimalSolutions : tableau 3D de triplet tel que : optimalSolutions[n][p]
        contient la solution optimale pour le channen n et la puissance maximale p.
    objectiveFunction : tableau des "user rate" correspondant.

    pour n=1 à N-1 faire
        pour p = 0 à P faire
            listTriplets ← liste des triplets triés (tri déjà réalisé) à p croissant
                pour le channel n.
            optimalSolutions[n][p] ← Tableau de taille n + 1 de triplets nuls.
            objectiveFunction[n][p] ← 0.
            tant que il reste des triplets dans listTriplets faire
                Triplet t ← prochain triplet de listTriplets
                si n = 0 alors
                    currentObjectiveFunction (entier) ← r_t
                    si currentObjectiveFunction > objectiveFunction[0][p] alors
                        objectiveFunction[0][p] ← currentObjectiveFunction
                        optimalSolutions[0][p][0] ← t
                    fin si
                sinon
                    currentObjectiveFunction (entier) ← objectiveFunction[n−1][p − p_t] + r_t
                    si currentObjectiveFunction > objectiveFunction[n][p] alors
                        objectiveFunction[n][p] ← currentObjectiveFunction
                        pour chan=0 à n-1 faire
                            optimalSolutions[n][p][chan] ← optimalSolutions[n-1][p-p_t][chan]
                        fin pour
                        optimalSolutions[n][p][n] ← t
                    fin si
                fin si
            fin tant que
        fin pour
    fin pour
    retourne optimalSolutions[N][P] // la solution
fin procédure
```

- Triplet currentTriplet : le triplet choisit pour le channel i.
- int userRateSum : garde en mémoire la somme des "user rate" de la solution partielle.
- int upperBound
- int currentChannel : le channel actuel i
- int requiredPower : la puissance nécessaire pour l'instance actuelle.

On ne stocke pas les triplets "parents" de l'instance. On arrive en effet à les récupérer par le mécanisme de récursivité. Cette classe possède un constructeur $BBInstance(Triplett, intcurrentChannel, intrequiredPower, intuserRateSum)$.

On suppose aussi qu'on a une variable globale, un entier $maxUserRate$ intialisé à 0, qui stocke le $userrate$ de la solution actuelle (complète), 0 si on n'en a pas encore trouvée. On possède également une variable globale pour stocker la solution temporaire : $bbSolution$.

Enfin on stocke les nouvelles instances dans des piles LIFO (Last In First Out). En effet, on étudie en premier les instances placés en haut de la pile qui sont les plus prometteuses car elles sont la plus haute $upperBound$.

Il suffit alors de faire tourner l'algorithme $solution(root)$ pour obtenir la solution.

Les algorithmes 5, 6 et 7 détaillent cela.

Pour la complexité temporelle :

-

Algorithme 5 Algorithme BB - branch

Input : la liste des triplets. L'instance initiale $BBInstanceroot$.
procédure BRANCH(BBInstance bbInstance)
  $currentChannel \leftarrow currentChannel_{ChannelbbInstance} + 1$
  bbInstanceList : liste des instances qu'on va créer.
  tant que il y a dans triplets restants dans le channel actuel **faire**
    Triplet t
    newBBInstance $\leftarrow BBInstance(t, currentChannel, requiredPower + p_t, userRateSum + r_t)$
    Calcul de la nouvelle borne $newBBInstance$ avec la fonction $bound(newBBInstance)$.
    Ajouter newBBInstance à la list bbInstanceList.
  fin tant que
  $bbInstanceStack \leftarrow$ Tri de la liste par la borne supérieure des instances et ajout à la pile (LIFO).
fin procédure

Algorithme 6 Algorithme BB - bound

```
procédure BOUNDS(BBInstance bbInstance)
    Output : couple d'entiers.
    si parentInstance est nul alors // calcul initial des bornes (pour la racine)
        upperBound ← ∑_{n=1} N max_{k,m}(r_{(k,m,currentChannel)})
    sinon
        Soit A, l' "userrate" de la solution renvoyée par l'algorithme glouton
            à partir de la solution partielle.
        upperBound ← userRateSum+
    fin si
    retourne upperBound
fin procédure
```

Algorithme 7 Algorithme BB - Solution

```
procédure SOLUTION(BBInstance bbInstance)
    si requiredPoewer_{bbInstance} > P ou upperBound_{bbInstance} > maxUserRate
        alors
        retourne false
    sinon
        si upperBound > maxUserRate et currentChannel = N alors
            maxUserRate ← upperBound
            bbSolution[currentChannel] ← currentTriplet
            retourne true
        sinon
            si  on est au dernier channel  alors
                retourne false  // ce n'est pas une solution.
            sinon
                bbInstanceStack ← branch(bbInstance)
                Soit solutionFound un boolean. Initialisé à false.
                tant que bbInstanceStack est non vide faire
                    BBInstancechild ← l'instancequiestenhautdelapile
                    si solution(child) alors
                        si currentChannel = 0 alors
                            retourne true
                        fin si
                        solutionFound ← true
                        bbSolution[currentChannel] ← currentTriplet
                    fin si
                fin tant que
                retourne solutionFound
            fin si
        fin si
    fin si
fin procédure
```

5.4 Question 11)

On obtient les résultats suivants :

$Test$	1	3	4	5
$Power$	78	68	?	?
$UserRate$	365	350	?	?
$time(ms)$	9.7	1.21	?	?

L'algorithme, pour les fichiers 4 et 5, prend très longtemps. De plus, pour le fichier 4, des problèmes de mémoire se pose. Je n'ai pas encore réussi à resoudre ces différents problèmes.