ACT

Pierre Lague & Romain Lecleire

Compte Rendu

TP3

Abstract

Le de ce troisème TP est de "concrétiser" les notions de propriété NP et de réduction polynomiale. Un travail rédactionnel de démonstration ainsi que d'implémentation est demandé pour familiariser les étudiants avec la méthode utilisée. Le langage de programmation utilisé est `Python 3` et `Java`.

Section 1: Avant implémentation : définition d'une propriété NP

En se rapportant au cours et aux définitions données dans le sujet de TP, une propriété NP est elle qu'il existe un polynome Q et un algorithme A à deux entrées et a valeurs booléenne tels que :

$$L = \{u/\exists c, A(c,u) = \text{Vrai}, |c| \leq Q(|u|)\}$$

On a alors à définir ce qu'est un certificat (dans l'equation précédente c). **Un certificat (c) est une structure de données respectant des contraintes qui restent à vérifier par un algorithme de vérification (A).**

Dans le cadre du problème de BinPacking, nous avons décidé d'implémenter un certificat sous la forme d'un tableau de tuples de la forme $[(index\_obj1, index\_sac1), ..., (index\_objx, index\_sacx)]$. Selon la définition le taille d'un certificat doit être bornée polynomialement par la taille des données en entrée.

Détaillons alors nos données d'entrée (ce qui suit est issu du sujet du TP) :

- n : un nombre d'objets
- $x_1, ..., x_n$ - n entiers, les poids des objets
- c : la capacité d'un sac (entière)
- k : le nombre de sacs

En termes de taille mémoire, nous avons l'équivalent :

- taille de n : $O(log(n))$ bits
- taille des $x_1, ..., x_n$: $O(n * log(c))$ bits car c est la valeur maximale de x dans le pire des cas.
- taille de c : $O(log(c))$ bits
- taille de k : $O(log(k))$ bits.

A noter que nos certificats ne peuvent pas dépendre de k car ils doivent être indépendants de la données d'entrée. Un certificat sera valable si sa taille est inférieure à la taille des donnés d'entrée, soit si

$$|c| \leq |n| + |x| + |c| + |k|$$

En choisissant un certificat prenant la forme d'un tableau de tuples avec des éléments ne dépendant que du nombre d'objets et pas du nombre de sacs. Sachant que nous avons 2 éléments par tuples, la taille de notre certificat est alors égale à : $|c| = 2n * log(c)$. En la comparant à la taille de nos données d'entrée on obtient :

$$O(n * log(c)) \leq O(n * log(c)) \equiv O(n * log(c)) \leq O(n * log(c)) + O(log(n)) + O(log(c)) + O(log(k))$$

La taille de notre certificat est donc bien bornée polynomialement par la taille de nos données d'entrée. En se rapportant à la définition de la classe NP, il faut un algorithme de vérification du certificat qui retournera $Vrai$ si ce certificat est une solution à notre problème. L'algorithme est le suivant :

Génération aléatoire d'un certificat

On nous demande ensuite de proposer un algorithme qui nous permettra de générer de façon aléatoire un certificat en ne prenant en entrée que le nombre de sac et le nombre d'objets. L'algorithme est le suivant :

Algorithm 1 Vérification du certificat du problème BinPack

```
function VERIFICATION_DE_CERTIFICAT(s, c, e)   ▷ Algorithme de vérification du certificat du problème BinPack
    tab ← tableau de zéros de longueur |s|
    for i ← 0 to |s| − 1 do
        index_bag ← s[i][0]
        index_object ← s[i][1]
        tab[index_bag] += e[index_object]
    end for
    for i ← 0 to |tab| − 1 do
        if tab[i] > c then
            print(i)
            return False
        end if
    end for
    return True
end function
```

Algorithm 2 Génération de certificat aléatoire

```
function MAKERANDOMCERTIFICAT(nb_objets, nb_pack)   ▷ Fonction qui génère un certificat aléatoire
    certificat ← liste vide
    for i ← 0 to nb_objets − 1 do
        append(certificat, (random.randint(0, nb_pack-1), i))
    end for
    return certificat
end function
```

Algorithme du British Museum

Nous proposons l'implémentation suivante pour l'algorithme du british museum. L'ordre sur lequel on se base pour ordonner les sacs est un ordre en "base n" comme un ordre en base x pour les chiffres normaux, ici, on itère nos sacs modulo le nombre de sacs :

Algorithm 3 Création du certificat suivant (BM)

```
function CREATION_CERTIFICAT_SUIVANT_BM(nombre_objets, nombre_sacs, certificat)
    if certificat est None then
        certificat ← liste vide
        for i ← 0 to nombre_objets − 1 do
            append(certificat, (1, i))
        end for
    else
        end_of_increment ← False
        i ← 0
        while (not end_of_increment) or (i == nombre_objets) do
            if i ≥ nombre_objets then                    ▷ Pour éviter le dépassement de l'index
                Return certificat
            end if
            if certificat[i][0] ≠ nombre_sacs then                    ▷ Incrémente le sac
                certificat[i] ← (certificat[i][0] + 1, certificat[i][1])
                end_of_increment ← True
            else          ▷ Si le sac est à la valeur maximale, réinitialise à 1 et incrémente le sac suivant
                certificat[i] ← (1, certificat[i][1])
                i ← i + 1
            end if
        end while
    end if
    Return certificat
end function
```

Section 2: Reductions polynomiales

Q1 Problème "Partition"

- Input :
  - n : un nombre d'entiers
  - $x_1 ... x_n$: les entiers en question
- Output :
  - Oui : s'il existe un sous-ensemble J de $\{x_1 ... x_n\}$ tel que la somme des $x_i \in J$ soit exactement égal à la moitié de la somme des x_i :

$$\sum_{i \in J} x_i = \frac{\sum_{i=1}^{n} x_i}{2}$$

  - Non : si ce n'est pas le cas.

Montrons que le probème "Parition" se réduit polynomialement en "BinPack"

Si on considère les poids des objets de BinPack comme étant les entiers de Parition, alors :

$$x_{i_{bin}} = x_{i_{part}}$$

et

$$n_{bin} = n_{part}$$

.

On considère également la capacité maximale C comme étant la moitié de la somme des x_i de BinPack :

$$\frac{\sum_{i=1}^{n} x_i}{2}$$

Enfin, le nombre de sacs k est de 2, car on créer 2 partitions en tout. Il est donc possible de coder les données de "Partition" dans le problème "BinPack".

Si I = Part(x_i, n) est une instance positive de "Partition", alors il existe un sous-ensemble $J \in [x_1...x_n]$ tel que $\sum_{i \in J} x_i = \frac{\sum_{i \notin J} x_i}{2}$. Alors la somme des poids des objets dans le sac J et le sac $\neg J$ est inférieure ou égale à C, car

$$\begin{aligned} &\sum_{i \in J} x_i \\ &= \frac{\sum_{i \notin J} x_i}{2} \\ &\leq \frac{\sum_{i=1}^{n} x_i}{2} \end{aligned}$$

Ainsi, I est aussi une instance positive de "BinPack".

Si I = BP(x_i, n, C, 2) est une instance positive de "BinPack", il existe une répartition des n objets en 2 sacs de telle sorte que la somme des poids de ces objets soit égale à $C = \sum_{i=1} x_i$. Si on considère le premier sac comme J et le deuxième sac comme $\neg J$, alors

$$\begin{aligned} &\sum_{i \in J} x_i \\ &= \frac{\sum_{i \notin J} x_i}{2} \end{aligned}$$

car la capacité de chaque sac est égale à la moitié de la somme des poids de tous les objets. Ainsi, I est aussi une instance positive de "Partition".

La complexité de la réduction est en O(n) car la modification des données est telle qu'elle ne dépends que de n. Elle est donc bien polynomiale.

S'il existe une solution au problème Partition, alors il existe une solution au problème BinPack. L'inverse et la négative sont vrais également. La reduction est donc polynomiale.

Q1.1 Coder la réduction polynomiale de partition dans BinPack.

Algorithm 4 reductionPartition

```
1: function reductionPartition(nombre_objets, nombre_sacs, capacite, poids)
2:     capacite ← sum(poids)//2 + sum(poids)%2
3:     return nombre_objets, nombre_sacs, capacite, poids
4: end function
```

Q1.2 La propriété Partition est connue NP-complète. Qu'en déduire pour BinPack ?

Selon les propriétés de la réduction polynomiale : soit L un problème dont la nature est inconnue et L' un problème NP-Complet; Si L' peut se réduire polynomialement dans L, alors L est NP-Complet. **Donc BinPack est NP-Complet.**

Q1.3 Pensez-vous que BinPack se réduise polynomialement dans Partition ? Pourquoi ?

Il n'est à priori pas possible de réduire polynomialement BinPack dans Partition. En effet, adapter les données de BinPack semble impossible pour qu'elles correspondent au problème Partition.

Q2 Entre Sum et Partition, lequel des deux problèmes peut être presque vu comme un cas particulier de l'autre ? Qu'en déduire en terme de réduction ?

- Input :
  - n : un nombre d'entiers
  - $x_1...x_n$: les entiers en question
  - c : un entier cible
- Output :
  - Oui : s'il existe un sous-ensemble J de $\{x_1...x_n\}$ tel que la somme des $x_i \in J$ soit exactement égal à c.
  - Non : si ce n'est pas le cas.

Entre Sum et Partition, Parition semble être le cas particulier de Sum où l'entier cible $c = \frac{\sum_{i=1} x_i}{2}$. **On en déduit que Sum va pouvoir se réduire polynomialement dans Partition.**

Q3 Montrer que Sum se réduit polynomialement en Partition et implémentez la réduction.

Il est possible de coder une instance de Partition en une instance de Sum :

- $n_{part} = n_{sum}$
- $x_{ipart} = x_{isum} + 2 * c_{sum} - \sum_{i=1} x_{i_{sum}}$

On ajoute à la fin de la liste des entiers la valeur :

$$2 * c_{sum} - \sum_{i=1} x_{i_{sum}}$$

de sorte à ce que la somme des entiers soit égale exactement à $2 * c_{sum}$. Il en résulterait alors une partition en deux ensembles de capacité égales à c_{sum}.

Soit $I = Sum(n, x_i, c)$ une instance positive de Sum, alors il existe un sous-ensemble J tel que la somme des $x_i \in J = c_{sum}$. Si on considère $J_{sum} = J_{part}$, $\neg J_{sum} = \neg J_{part}$ et $c_{sum} = \frac{\sum_{i \in J} x_i}{2}$ alors on retrouve une instance positive de Partition.

Soit $I = Part(n, x_i)$ une instance positive de Partition, alors il existe un sous-ensemble J tel que la somme des $x_i \in J = \frac{\sum_{i \in J} x_i}{2}$. Si on considère $J_{part} = J_{sum}$, $\neg J_{part} = \neg J_{sum}$ et $c = \frac{\sum_{i \in J} x_i}{2}$ alors on retrouve une instance positive de Sum où la somme des $x_i \in J$ doit être égale à la moitié de la somme des x_i.

La complexité de la réduction est en O(n) car la valeur ajoutée à la fin de la liste des objets ne dépends que de n. Elle est donc bien polynomiale.

S'il existe une solution au problème Partition, alors il existe une solution au problème Sum. L'inverse et la négative sont vrais également. La réduction est donc polynomiale.

Pour l'implémenter on peut utiliser cet algorithme qui à partir d'une instance de Sum donne une instance de Partition :

Algorithm 5 reductionSum

```
1: function reductionSum(n_sum, x_sum, c_sum)
2:     x_sum.append(2 * c_sum − ∑_{i=1} x_{i_sum})
3:     return n_sum, x_sum
4: end function
```

Q4 En utilisant la réduction précédente, comment implémenter une réduction polynomiale de Sum dans BinPack ?

On cherche une fonction de réduction permettant de résoudre le problème Sum en un problème BinPack. Pour cela, il faut adapter les données du problème Sum pour qu'elles soient compatibles et utilisables avec le problème BinPack.

- $n_{sum} = n_{bin}$
- $x_{isum} = x_{ibin}$ a la fin du tableau on ajoute la valeur suivante : $2 * c_{sum} - \sum_{i=1} x_{i_{sum}}$
- $k_{bin} = 2$
- $c_{sum} = c_{bin}$ (capacité d'un sac)

Donc résoudre le problème Sum revient à résoudre le problème BinPack : on cherche à savoir s'il existe un sac de capacité égale à la moitié de la somme des objets. Si oui, alors il existe une partition des objets en deux groupes de poids égaux. Si non, alors il n'existe de pas de répartition des objets en deux groupes de poids égaux.

A priori, s'il existe une solution au problème BinPack, alors il existe une solution au problème Sum. L'inverse et la négative sont vrais également. La réduction est donc polynomiale car la réduction de BinPack ne dépends que la taille de n.

Q5 Proposer une réduction polynomiale de BinPackDiff dans BinPack (inutile de l'implémenter)

On cherche une fonction de réduction permettant de résoudre le problème BinPackDiff en un problème BinPack. Pour cela, il faut adapter les données du problème BinPackDiff pour qu'elles soient compatibles et utilisables avec le problème BinPack.

- $n_{BPDiff} = n_{BP}$
- $x_{i_{BPDiff}} = [\sum_{i=1} c_{i_{BP}}] - c_1, ..., [\sum_{i=1} c_{i_{BP}}] - c_k + [x_1, x_2, ..., x_n]$ (on ajoute le poids des objets à la fin du tableau)
- $k_{BPDiff} = k_{BP}$ (le nombre de sacs)
- $c_{i_{BPDiff}} = \sum_{i=1} c_{i_{BP}}$ (capacité d'un sac)

On choisit une capacité de sac significativement plus grande que les capacités fournies dans BinPackDiff. Dans ce contexte, la somme des capacités des sacs est utilisée, bien qu'il soit également possible de sélectionner une valeur plus élevée. Ensuite, un objet factice est ajouté à chaque sac de manière à ce que la capacité restante

du sac soit équivalente à celle spécifiée dans BinPackDiff. La valeur attribuée à cet objet factice dépend du choix de la capacité des sacs et doit être ajustée en conséquence :

$$\sum_{i=1}^{k}(ci) - ci$$

Résoudre le problème BinPackDiff revient à résoudre le problème BinPack suivant : **L'objectif est de déterminer si une solution existe en ajustant les données de manière à égaliser la capacité d'un sac avec la plus petite capacité parmi tous les sacs. Si cela est réalisable, une solution existe. Dans le cas contraire, il est probable qu'aucune solution n'existe.** La réduction est polynomiale car la modification des données ne dépends que de la taille de n.

Section 3: Optimisation versus Décision

Q1.1 Montrer que si BinPackOpt1 était P , la propriété BinPack le serait aussi ; qu'en déduire pour BinPackOpt1 ?

Si BinPackOpt1 était dans P, il existerait un algorithme A permettant de trouver la valeur minimale K de k. On aurait alors un algorithme polynomial pour résoudre BinPack : On appliquerait A et on testerait si la donnée k en entrée de BinPack est telle que $k \leq K$. Ainsi, il existerait un algorithme polynomial pour BinPackOpt1. Il existerait donc un algorithme polynomial pour un problème de décision NP-DUR : on aurait donc P = NP.

Donc le probleme d'optimisation BinPackOpt1 est NP-DUR

Q1.2 Montrer que si BinPackOpt2 était P , la propriété BinPack le serait aussi ; qu'en déduire pour BinPackOpt2 ?

Si BinPackOpt2 était dans P, il existerait un algorithme A pour trouver la valeur minimale K (le nombre de sachets dans la mise en sachets) de k. On aurait alors un algorithme polynomial pour résoudre BinPack : On appliquerait A et on testerait si la donnée k en entrée de BinPack est telle que $k \leq K$. Ainsi, il existerait un algorithme polynomial pour BinPackOpt2. Il existerait donc un algorithme polynomial pour un problème de décision NP-DUR : on aurait alors P = NP

Donc le probleme d'optimisation BinPackOpt2 est NP-DUR

Q2 Montrer que si la propriété BinPack était P , BinPackOpt1 le serait aussi.

Soit une instance I de BinPackOpt1 où nous devons minimiser la taille du plus grand bac. Transformons cette instance I en une instance I' de BinPack en ajoutant simplement un bac vide suffisamment grand. Appliquons l'algorithme A (qui résout BinPack) à l'instance I'. L'algorithme A donnera une solution à BinPack avec le même nombre de bacs que la taille minimale pour BinPackOpt1. La transformation et l'algorithme A peuvent être effectués en temps polynomial par définition. Ainsi, si nous pouvons résoudre BinPack en temps polynomial, nous pouvons résoudre BinPackOpt1 de manière efficace en utilisant la même approche. Cela montre que si BinPack est dans P, alors BinPackOpt1 le serait aussi.