Université de Lille 1 SCIENCES ET TECHNOLOGIES

Master 1 Informatique

TP ACT 3

Compte rendu de TP

Antoine CANDA
Theo VERSCHAEVE

Version 1.0
16 décembre 2016

Table des matières

Chapitre 1

Qu'est ce qu'une propriété NP ?

Question 1 :

Définition de la notion de certificat

Un certificat est une preuve de l'existence d'une solution au problème traité. Le certificat est une solution du problème en taille polynomial.

Implémentation possible d'un certificat

On peut implanter sous forme d'un tableau de taille n représentant chaque objet. On remplit chaque case avec un entier compris entre 1 et k représentant le sac dans lequel on place l'objet.

Taille d'un certificat par rapport à l'entrée

La taille sera n,avec n = au nombre d'objet de l'entrée, donc ce dernier est bien en taille polynomial.

Algorithme de vérification de certificat

tab[n] : le certificat
Poids[k] : le tableau représentant les sacs contenant des entiers représentant le poids du sac qui est initialisé a 0
Pour i=0 à n-1
  p = poids de l'objet i
  Poids[tab[i]-1] += p
fin pour
Pour i=0 à k-1
  Si Poids[i] > c
  retourne faux
  fin si
fin pour
retourne vrai

Complexité

Initialisation du tableau Poids à 0 = O(k). Ajout des poids de chaque objet au sac correspondant à sa position : O(n). Vérification du poids de chaque sac par rapport à la capacité maximale d'un sac : O(k).
On a une complexité en O(2k + n). Si on considère que l'instance du problème correspond au nombre de sac k et au nombre d'objet n alors on a une complexité qui est polynomiale.
On est également en présence d'un problème qui est bien NP car on a un certificat qui est en taille polynomial, ainsi qu'un algorithme de verification lui aussi en taille polynomial.

Question 2

Génération aléatoire d'un certificat :
Un entier n représentant le nombre d'objet
Un entier k représentant le nombre de sac
Crée un tableau tab de taille n initialise à 0 ;
Pour i=0 à n-1
  tab[i] = random(1..k) ;
fin pour
return tab ;
Chaque objet a une probabilité égale d'appartenir à l'un des k sacs.
Et on répète cette opération pour les n objets.
On a donc une probabilité égale d'avoir chaque certificat possible.

Question 3

1 :

$$k^n$$

2 :

Le certificat le plus petit serait le tableau contenant que des 1 pour chaque case.
Le certificat le plus grand serait celui contenant que des k.
On incrémenterait chaque case du tableau jusqu'à arriver à k.
Si le contenu de la case que l'on doit incrémenter est égale à k alors on incrémente la case à sa droite si elle existe et on repasse le contenu de la case précédente à 1.
On recommence cette boucle jusqu'à ce que chaque case égale k.

3 :

On utiliserait une boucle tant que avec à l'intérieur la génération d'un certificat. On teste ce certificat avec l'algorithme de vérification. Si ce certificat est valide, on retourne vrai car

on a une solution sinon on génère le certificat qui suit le précédent. Si après avoir généré le dernier certificat, on a toujours pas de solution alors on renvoie faux : on n'a pas trouvé de solution.
On est dans une complexité exponentielle : on doit tester tout les certificats possible et c'est un ordre de grandeur exponentielle.

Question 4

Voir code source fourni.
On a utilisé le langage java pour l'implémentation.
Utilisation, dans le dossier contenant l'exécutable, via la commande :

java -jar binpack.jar <file> <mode> <nbsac>

Les modes possibles sont :

- -nd : méthode non déterministe
- -exh : méthode exhaustive (British Museum)
- -ver : vérification d'un certificat entrée manuellement
- -rdP : résolution d'un problème Partition avec réduction dans BinPack, le paramètre <nbsac> peut être à 0
- -rdS : résolution d'un problème Sum avec réduction, le paramètre <nbsac> représente l'entier cible

Chapitre 2

Réductions polynomiales

Question 5

Les données du problème Partition sont un nombre n d'entier. Le but de ce problème est de déterminer si il existe une possibilité d'obtenir exactement la moitié de cette somme à partir de cette liste d'entier. On peut transformer ce problème en une instance de BinPack facilement.
En effet, si on calcul la somme de la liste d'entier présent puis que l'on divise ce résultat par deux, alors on a deux données qui sont importantes : le nombre de sac k qui sera égale à 2 et la capacité c qui vaudra la moitié de cette somme.
Si on peut placer les n objets dans les 2 sacs de capacité c alors il existe une solution au problème de partition.
Cette réduction est polynomial : on a une somme sur les entiers en entrée qui se fait en temps linéaire ainsi qu'une division en temps constant. On est donc en temps polynomial (linéaire même) sur l'entrée.

Question 5.1

Voir code source.
Pour exécuter : **java -jar binpack.jar <fichier> -rdP <nbSac>**

<nbSac> est une valeur quelconque et fichier un fichier au format de binpack (on ne lit pas le premier paramètre correspondant à la capacité), par exemple 0.

Question 5.2

Si la propriété Partition, définie comme étant NP-complet, se réduit en temps polynomiale dans le problème BinPack, c'est à dire qu'avec un algorithme polynomial, on transforme une instance de Partition en une instance de BinPack, alors BinPack est également NP-complet car on a montré que BinPack était un problème NP et vu que Partition se réduit en temps polynomial en une instance de BinPack alors BinPack est NP-dur. Une propriété NP et NP-dur est dite NP-Complète.

Question 5.3

Oui le problème BinPack peut se réduire en temps polynomial dans Partition car les deux sont des problèmes NP-complet et qu'il est possible de réduire un problème NP-complet dans un autre, de manière plus ou moins facile.

Question 6

Le problème Partition peut être vu comme un cas particulier de Sum.
On en déduit donc qu'il est possible de réduire Sum dans Partition.
En effet, le cas de partition peut être vu comme le cas où l'entier cible du problème Sum est égale à la moitié de la somme des entiers de l'entrée.
L'idée est de faire en sorte qu'on détermine si on peut sommer l'entier cible dans la liste des entiers que l'on a.
La réduction en Partition n'a qu'une liste d'entier en paramètre et va chercher à calculer si on peut trouver une somme d'entier égale à la moitié de la somme totale des entiers de la liste.
Cela implique que l'on perde l'information de la cible dans le résultat : on ne passe qu'une liste d'entier. Et on ne peut pas se contenter de passer la liste d'entier que l'on a. De la même façon, il faut retrouver cette idée avec une division en deux de la somme totale. Une manière simple est de rajouter un entier avec la valeur de cette somme pour avoir une symétrie lors qu'on divisera pour le problème de partition.
Mais il y a le manque d'apparition de l'entier cible donc on pourrait l'ajouter à la somme.
Le problème est qu'il n'y a plus la symétrie que l'on souhaite avoir. Donc on pourrait ajouter cette fois ci deux fois la somme des entiers moins la valeur de l'entier cible.
On aurait donc cette information présente de manière indirecte deux fois avec les deux entiers.
Et pour l'obtenir, il est nécessaire de la "reproduire" avec les entiers de la liste initiale.
La somme de tout les entiers de la liste est égale à 4 fois la somme de la première liste.
Donc avec partition il faudra obtenir une somme égale à 2 fois la valeur de la liste initiale en considérant qu'on a un entier dont la valeur manquante pour atteindre cette moitié est la valeur de la cible.
Donc si on a la possibilité de faire cette somme avec la liste initiale, on valide notre problème en validant Partition.
Cette réduction est polynomiale car il s'agit de somme sur l'entrée. (1 somme de ces entiers en temps linéaire, puis une addition puis une multiplication et une soustraction).

Question 7

Voir code source et question 6.
Pour exécuter : **java -jar binpack.jar <fichier> -rdS <entierCible>**

<entierCible> remplace l'entrée <nbSac> utilisé pour le problème binpack.

Question 8

On peut réduire une instance de Sum dans Partition puis réduire cette instance de Partition dans BinPack.
On a donc une réduction de Sum dans BinPack à l'aide d'une composition de réduction.
Dans le code source, je réduis Sum dans partition qui est résolu dans une réduction en BinPack.

Question 9

On souhaite réduire BinPackDiff dans BinPack.
La différence entre les deux problèmes est simple : dans BinPackDiff, la capacité des sacs n'est pas fixe.
Il faut donc faire en sorte de retrouver cette information de taille variable dans BinPack.
Le problème BinPack est caractérisé par une liste d'objet ayant chacun un poids associé, un nombre de sac, une capacité de sac.
Une réduction possible pourrait être la suivante :
On garde le même nombre de sac dans BinPack que dans BinPackDiff. On fixe la capacité des sacs de BinPack avec la capacité du plus grand sac du problème BinPackDiff que l'on appelle cmax. On crée des nouveaux objets de poids cmax moins la capacité du sac[i] si le résultat est >0 (au plus n-1 nouveaux objets). On a ainsi une instance de Binpack.

L'algorithme serait le suivant :

```
cmax = 0 ;
Pour i=0 à n-1
  si c[i] > cmax
  cmax = c[i]
  fin si
fin pour
Pour i=0 à n-1
poids = cmax - c[i]
  Si(poids > 0)
  ajouteTabObjet(poids)
  fin si
fin pour
return PblBinPack pblBinPack = new PblBinPack(taille(tabObjet), tabObjet, cmax, k)
```

La réduction est en temps polynomial car on a deux boucles qui se font en temps linéaire sur les objets en entrées pour transformer l'instance de BinPackDiff en BinPack.

Chapitre 3

Optimisation versus Décision

Question 10

BinPackOpt1 est un cas particulier de BinPack qui minimise le nombre de sachet.
On peut par exemple commencer par déterminer un nombre k de sachet pour laquelle on a une mise en sachet possible et essayer d'avoir une mise en sachet correct pour k-1 jusqu'à avoir k-i sachet pour laquelle on ne peut plus avoir de mise en sachet possible.
On a donc le nombre de sachet minimale qui sera k-i+1. On peut donc ramener d'une certaine façon à un problème de décision.
Ce n'est probablement pas optimale mais c'est probablement possible.
Mais si on dit que l'on peut réduire ce problème en un problème de BinPack, alors si on peut le résoudre en temps polynomial, on peut résoudre également en temps polynomial BinPack or ce n'est pas le cas.
Donc on en déduit que cette version optimisée est également un problème NP (et plus particulièrement NP-complet car si on peut réduire en BinPack alors il est NP-dur et il est très certainement NP de base).
On peut dire la même chose pour BinPackOpt2 qui est similaire dans l'esprit.

Question 11

Si BinPack est P, comme il est possible de réduire BinPackOpt1 en une instance de BinPack, on aurait un algorithme de résolution de BinPackOpt1 en temps polynomial.
Donc BinPackOpt1 serait une propriété P.

Question 12

La stratégie BinPackOpt2 revient à faire un BinPack en utilisant une technique de placement si possible optimale (FirstFit ? BestFit...) et est donc réductible dans BinPack.
Or si BinPack est en temps polynomial alors on aurait aussi BinPackOpt2 en temps polynomial.