ACT

TP3: Les propriétés NP, les réductions polynomiales

GIBIER François | MAZURE Antoine

1 Qu'est-ce qu'une propriété NP ?

Q1) Un certificat dans le problème BinPack est une répartition de n objets dans k sacs. Le certificat est valide si chaque objet i est affecté à un sac j et si la somme des poids des objets dans chaque sac ne dépasse pas la capacité c.

Autrement dit, il faut que:

$\forall s \in sacs, \sum_{o}^{o \in s} x[o] \leq c$

$\forall o \in objets, \exists! s \in sacs, \text{tq } o \in s$

Un certificat est representé par un tableau de taille n composé d'entiers de taille maximum log k bits. On a donc un certificat en $O(n \log k)$. Le certificat est donc bien borné polynomialement par la taille des données du problème.

```
def verif_certificat(int aff[n], int objets[n], int nb_sacs, int c) -> bool:
    Si taille(aff) != taille(objets):
        retourner Faux;

    int somme_poids_sacs[nb_sacs]; // initialisé à 0
    Pour i allant de 0 à n:
        somme_poids_sacs[aff[i]] += objets[i];

    Pour i allant de 0 à k:
        Si somme_poids_sacs[i] > c:
            retourner Faux

    retourner Vrai
```

Ici le certificat est donc le tableau aff, qui est une affectations de sac aux objets. La complexité de l'algorithme de vérification est donc $\theta(n + k)$, car on doit parcourir les objets puis les sacs pour voir qu'aucun des sacs n'est trop rempli.

Q2.1) Les certificats ont la même chance d'apparaître, cependant, les nombres sont pseudo aléatoires, mais à un instant t, on a la même chance d'obtenir n'importe quelle permutation d'objets.

Q2.2) Un algorithme non-déterministe polynomiale pour BinPack consiste à deviner une solution possible (affectation des objets aux sacs) et à vérifier en temps polynomial que cette solution satisfait les contraintes de capacité et de nombre de sacs.

```
certificate = generate_certificate(objets, k, c)
```

`verify_certificate(certificate, objets, k, c)`

Q3.1) On a le nombre d'arrangements avec répétition des n objets parmi k sacs, soit k^n certificats, car on a k choix possibles pour n objets donc on a n multiplications successives de k.

Q3.2) On peut utiliser l'ordre lexicographique en base k pour énumérer les certificats; Pour 2 objets et 3 sacs (donc en base 3), les certificats seront [0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0] [2,1], [2,2].

Q3.3) L'algorithme du British Museum est le fait de tester toutes les possibilités une à une en partant de la plus petite dans l'espoir de trouver une solution. Ici, les possibilités sont les certificats, on va donc tester tous les certificats dans l'ordre lexicographique jusqu'à trouver un certificat valide, ou pas. Cet algorithme est en $O((n+k) * k^n)$, car l'algorithme de vérification est en $\theta(n+k)$ et on le fait maximum k^n fois.

2 Réductions polynomiales

Q1) Le problème Partition peut être vu comme une version simplifiée de BinPack avec deux sacs (k=2), chacun ayant une capacité égale à la moitié de la somme totale des entiers. Si il existe une configuration pour laquelle les objets peuvent être placés dans ces deux sacs, alors l'instance de Partition est positive.

Q1.1) Voir *partition* dans main.c.

Q1.2) Si Partition est connu comme NP-Complet, cela veut dire qu'il est également NP-dur, et donc que tout problème NP est réductible polynomialement dans Partition. Comme Partition se réduit polynomialement en BinPack, cela veut dire que BinPack est au plus aussi difficile que Partition, donc il est NP-dur. De plus, on a montré qu'il existe un algorithme de vérification polynomial pour BinPack. BinPack est donc NP et NP-dur, il est donc lui aussi NP-Complet.

Q1.3) Non, BinPack n'est pas réductible polynomialement en Partition car pour $k > 2$, cela ne marcherait pas, on aurait que 2 sous-ensembles traités, cela montrerait seulement qu'il existe ou non une solution pour $k = 2$. Partition n'est pas assez générale pour que BinPack ne se réduise en Partition. Dans le cas ou Partition renvoyait les deux sous-ensembles (si une solution existe), on pourrait réduire BinPack quand k est une puissance de 2, on pourrait récursivement divisé les sous-ensembles en k sacs.

Q2) Partition est un cas particulier de Sum, quand la somme des éléments est égale à c * 2.

Cela signifie qu'un algorithme qui résout Sum peut être utilisé pour résoudre Partition en choisissant la cible c comme étant la moitié de la somme des objets . Donc, le problème Partition se réduit en temps polynomial au problème Sum.

Q3) Pour le cas où la somme des objets est supérieure ou égale à c * 2, on peut directement renvoyer le résultat de partition. Sinon, on peut ajouter un objet

ayant pour valeur la différence de la somme des objets et 2 * c (sum_elements - 2 * c) à notre liste d'objets, ce qui nous fait retomber dans le cas sum_elements = 2 * c et donc on peut directement appliquer Partition.

Voir fonction *sum_red* dans main.c.

Q4) Comme Sum peut se réduire polynomialement en Partition, et Partition peut se réduire polynomialement en BinPack, par transitivité, Sum se réduit polynomialement en BinPack.

Sum -> Partition -> BinPack

Q5) Pour réduire polynomialement BinPackDiff en BinPack, on peut prendre comme c le maximum des poids des différents sacs. Puis, pour chacun des sacs, on va ajouter la différence qu'ils ont avec c.

Exemple: [1, 2, 8, 4, 6, 3] Poids des sacs: [13, 4, 6, 4] On prends donc c = 13, et on aura donc 4 sacs de poids 13. On ajoute ensuite les objets [9 (13 - 4), 7 (13 - 6), 9 (13 - 4)] à la liste des objets. Ces objets vont donc compenser le poids ajouté dans chacun des sacs.

On a: k = k c = max(poids_des_sacs) objets = [...objets, ...differences_des_poids_avec_c]

3 Optimisation versus Décision

Q1) Si BinPackOpt(1 ou 2) était P, il existerait un algorithme polynomial pour trouver le nombre minimal de sacs necessaires à la répartition des objets. On peut donc directement savoir si pour k sacs il est possible de répartir les objets. Et donc BinPack serait P.

Q2) On a maximum n (nombre d'objets) sacs, donc on peut seulement itérer de 1 à n pour trouver si une solution est possible. On serait donc en $O(n * p)$ avec p la complexité d'un algorithme polynomial (car BinPack est supposé P) ce qui est bien polynomial.

Q3) Si BinPack était P, cela voudrait dire qu'il existerait un algorithme pour trouver le nombre de sachets minimisant k. Avec le nombre minimal de sachets, on peut utiliser une heuristique de BinPacking comme "Best-Fit" qui serait en $O(n \log n)$.

Best-Fit agit comme ceci: On a k sachets, et on va essayer d'ajouter à chaque fois l'objet dans le sac le plus rempli jusqu'à ne plus avoir d'objets, ou que les sacs soient remplis.

La partie du problème faisant de BinPack un algorithme appartenant à la classe NP est le fait de déterminer le nombre minimal de sacs nécessaires à l'obtention d'une solution.

Donc si BinPack était P, alors BinPackOpt2 serait lui aussi P.