TP 3 - Les propriétés NP, les réductions polynomiales

Objectif : Le but du TP est de "concrétiser" les notions de propriété NP et de réduction polynomiale.

A faire : Le TP a trois sections, une sur la notion de NP, une sur celle de réduction polynomiale, une dernière (sans implémentation) sur le lien optimisation/décision. Le TP est à rendre sous forme d'une archive contenant le code et un rapport avec les réponses aux questions et vos choix d'implémentation. Vous disposez d'exemples de données pour tester.

1 Qu'est-ce qu'une propriété NP ?

On va travailler sur le problème de la mise en sachets. Le problème est de placer un certain nombre d'objets dans un certain nombre de sacs. Les objets sont de poids divers, chaque sac acceptant une charge maximale -la même pour tous dans cette version. La propriété *BinPack* associée est définie formellement comme suit :

BinPack

Donnée :

n –un nb d'objets

$x_1, \cdots, x_n$ – n entiers, les poids des objets

c –la capacité d'un sac (entière)

k –le nombre de sacs

Sortie :

Oui, si il existe une mise en sachets possibles, i.e. :

$aff : [1..n] \rightarrow [1..k]$ –à chaque objet, on attribue un numéro de sac

tq $\sum_{i/aff(i)=j} x_i \leq c$, pour tout numéro de sac j, $1 \leq j \leq k$. – aucun sac n'est trop chargé

Non, sinon

Exemple : soit $n = 5$, $x_1 = 3, x_2 = 2, x_3 = 4, x_4 = 3, x_5 = 3$; si $c = 5$, il y a une solution pour $k = 4$ mais pas pour $k = 3$; si $c = 4$, il faut au moins $k = 5$ pour avoir une solution.

Avant d'implémenter !

Q 1. *La propriété est NP.*

NP

L est dit NP si il existe un polynôme Q, et un algorithme polynomial A à deux entrées et à valeurs booléennes tels que :

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

Définir une notion de certificat.

Comment pensez-vous l'implémenter ?

Quelle sera la taille d'un certificat ? La taille des certificats est-elle bien bornée polynomialement par rapport à la taille de l'entrée ?

Proposez un algorithme de vérification associé. Est-il bien polynomial ?

Q 2. *NP = Non déterministe polynomial*

Q 2.1. *Génération aléatoire d'un certificat.*

Proposez un algorithme de génération aléatoire de certificat, i.e. qui prend en entrée une instance de BinPack, ou seulement le nombre d'objets et le nombre de sacs, et génère aléatoirement un certificat de façon à ce que chaque certificat ait une probabilité non nulle d'être généré.

Votre algorithme génère-t-il de façon uniforme les certificats, i.e. tous les certificats ont-ils la même probabilité d'apparaître ?

Q 2.2. Quel serait le schéma d'un algorithme non-déterministe polynomial pour le problème ?

Q 3. *$NP \subset EXPTIME$*

Q 3.1. Pour n et k fixés, combien de valeurs peut-prendre un certificat ?

Q 3.2. *Enumération de tous les certificats*

Une méthode classique pour énumérer les certificats (soit définir un itérateur sur les certificats) consiste à s'appuyer sur un ordre total sur les certificats associés à un problème. Donc, on définit le certificat de départ comme "le plus petit" pour l'ordre, et la notion de successeur qui permet de passer d'un certificat au suivant sauf pour le "maximal". Quel ordre proposez-vous ?

Q 3.3. *L'algorithme du British Museum*

Comment déduire de ce qui précède un algorithme pour tester si le problème a une solution ? Quelle complexité a cet algorithme ?

Implémentation

Q 4. Implémenter les notions et algorithmes évoqués ci-dessus.

On devra donc être capable de lire une instance du problème *BinPack*, lire une proposition de certificat, vérifier si un certificat est valide, vérifier si le problème a une solution en essayant tous les certificats, "vérifier aléatoirement" si le problème a une solution en générant aléatoirement un certificat.

A la fin du sujet est proposé un embryon d'architecture en Java : vous pouvez en choisir une autre ou/et utiliser d'autres langages (C, CAML, HASKELL, ..). Vous veillerez à documenter votre code, à respecter l'esprit et à faciliter le test en respectant le schéma ci-dessous.

Pour tester, on pourra donc avoir un programme qui lit l'instance du problème dans un fichier et :

. en mode "vérification" propose à l'utilisateur de saisir un certificat et vérifie sa validité.

. en mode "non-déterministe", génère aléatoirement un certificat, le teste et retourne Faux si il n'est pas valide, "Vrai" sinon (en affichant éventuellement la valeur du certificat).

. en mode "exploration exhaustive" génère tous les certificats jusqu'à en trouver un valide, si il en existe un et retourne Faux si il n'en existe pas de valide -la propriété n'est donc pas vérifiée- , "Vrai" sinon (en affichant éventuellement la valeur du certificat trouvé).

Par exemple pour Java, l'usage pourra être : java testBinPack <files> <mode> <nbsachets> avec comme modes (au moins) -ver, -nd, -exh.

Attention ! Ne pas utiliser la version "exploration exhaustive" sur des problèmes de grande taille !

2 Réductions polynomiales

Une première réduction très simple

Soit le problème de décision *Partition* défini par :

Partition

Donnée :

n –un nombre d'entiers

$x_1, \cdots, x_n$ –les entiers

Sortie :

Oui, si il existe un sous-ensemble de $[1..n]$ tel que la somme des x_i correspondants soit exactement la moitié de la somme des x_i, i.e. $J \subset [1..n]$, tel que $\sum_{i\in J} x_i = \sum_{i\notin J} x_i = \frac{\sum_{i=1}^{n} x_i}{2}$

Non, sinon

Q 1. Montrer que *Partition* se réduit polynomialement en *BinPack*.

Q 1.1.[*à coder*] Implémenter la réduction polynomiale de *Partition* dans *BinPack*. Vous pouvez tester avec les données fournies.

Q 1.2. La propriété *Partition* est connue $NP-$complète. Qu'en déduire pour *BinPack* ?

Q 1.3. Pensez-vous que *BinPack* se réduise polynomialement dans *Partition* ? Pourquoi ?

Une réduction un peu moins évidente

Soit maintenant *Sum* défini par :

Sum

Donnée :

n –un nombre d'entiers

$x_1, \cdots, x_n$ –les entiers

c –un entier cible

Sortie :

Oui, si il existe un sous-ensemble de $[1..n]$ tel que la somme des x_i correspondants soit exactement c, i.e. $J \subset [1..n]$, tel que $\sum_{i\in J} x_i = c$

Non, sinon.

Q 2. 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 ?

Q 3.[*à coder*] Montrer que *Sum* se réduit polynomialement en *Partition* et implémentez la réduction.

Composition de réductions

Q 4. En utilisant la réduction précédente, comment implémenter une réduction polynomiale de *Sum* dans

BinPack ?

Une dernière réduction

On considère maintenant le cas où les sacs peuvent êre de capacités différentes. La propriété de décision $BinPackDiff$ associée est définie formellement comme suit :

BinPackDiff

Donnée :

n –un nb d'objets

$x_1, \cdots, x_n$ – n entiers, les poids des objets

k –le nombre de sacs

$c_1, c_2, \ldots, c_k$ –les capacités des sacs (entières)

Sortie :

Oui, si il existe une mise en sachets possibles, i.e. :

$aff : [1..n] \rightarrow [1..k]$ –à chaque objet, on attribue un numéro de sac

tq $\sum_{i/aff(i)=j} x_i \leq c_j$, pour tout numéro de sac j, $1 \leq j \leq k$. – aucun sac n'est trop chargé

Non, sinon

Q 5. Proposer une réduction polynomiale de $BinPackDiff$ dans $BinPack$ (inutile de l'implémenter).

3 Optimisation versus Décision

Au problème de décision $BinPack$, on peut associer deux problèmes d'optimisation :

BinPackOpt1

Donnée :

n –un nb d'objets

$x_1, \cdots, x_n$ – n entiers, les poids des objets

c –la capacité d'un sac (entière)

Sortie : le nombre minimal de sachets nécessaires pour la mise en sachets.

BinPackOpt2

Donnée :

n –un nb d'objets

$x_1, \cdots, x_n$ – n entiers, les poids des objets

c –la capacité d'un sac (entière)

k –le nombre de sacs

Sortie : Une mise en sachets correcte qui minimise le nombre de sachets.

Q 1. Montrer que si $BinPackOpt1$ (resp. $BinPackOpt2$) était P, la propriété $BinPack$ le serait aussi ; qu'en déduire pour $BinPackOpt1$ (resp. $BinPackOpt2$) ?

Q 2. Montrer que si la propriété $BinPack$ était P, $BinPackOpt1$ le serait aussi.

Q 3. *Plus dur...* Montrer que si la propriété $BinPack$ était P, $BinPackOpt2$ le serait aussi.

Exemple d'architecture de code

Ce qui suit **n'est qu'un exemple** d'architecture de code en JAVA pour la partie 1 ; vous pouvez la modifier ou en choisir une autre, choisir un autre langage.

```
/*****************************************************************************************************
              EXEMPLE D'ARCHITECTURE DE CODE-  CE N'EST QU'UNE PROPOSITION
*****************************************************************************************************/
//La classe abstraite des problèmes de décision
abstract class PblDec{
   public PblDec(){};

   //retourne Vrai SSi le pb de décision a une solution
   abstract public boolean aUneSolution();
}

interface Certificat{
   //retournera vrai SSi le certificat est bien correct pour le pb auquel il est associé
    public boolean estCorrect();

   //pour pouvoir énumérer les certificats, on définit un ordre sur les certificats
   //le certificat prend la valeur suivante dans l'ordre choisi
   public void suivant();
   //le certificat est le dernier dans l'ordre choisi
   public boolean estDernier();
   //le certificat prend une valeur alétaoire
   public void alea();

   public void affiche();
}
....class PblBinPack extends PblDec
{
    private int nbObjets;
    private int poids[];
    private int cap;
    private int nbSacs;

   //....
    public boolean aUneSolution() {
       // essaie tous les certificats un à un jusqu'à en trouver un correct -si il existe ....
    }

    //Algo non déterministe
    //si il y a une solution, au moins une exécution doit retourner Vrai
    // sinon, toutes les exécutions doivent retourner Faux
    public boolean aUneSolutionNonDeterministe() {
        //génère alétaoirement un certificat et vérifie si il est correct ..
    }
}
...... class  CertificatBinPack implements Certificat{
  private PblBinPack pb; //le pb auquel est associé le certificat
...
}
```